



**Application Programming
and SQL Guide**



**Application Programming
and SQL Guide**

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 1183.

Sixth Edition, Softcopy Only (February 2008)

This edition applies to Version 8 of IBM DB2 Universal Database for z/OS (DB2 UDB for z/OS), product number 5625-DB2, and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

This and other books in the DB2 UDB for z/OS library are periodically updated with technical changes. These updates are made available to licensees of the product on CD-ROM and on the Web (currently at www.ibm.com/software/data/db2/zos/library.html). Check these resources to ensure that you are using the most current information.

© Copyright International Business Machines Corporation 1983, 2008. All rights reserved.

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

Contents

About this book	xxi
Who should read this book	xxi
Terminology and citations	xxi
How to read the syntax diagrams	xxii
Accessibility	xxiii
How to send your comments	xxiv

Summary of changes to this book	xxv

Part 1. Using SQL queries	1

Chapter 1. Retrieving data	3

Result tables	3
Data types	4
Selecting columns: SELECT	5
Selecting all columns: SELECT *	5
Selecting some columns: SELECT column-name	6
Selecting derived columns: SELECT expression	7
Eliminating duplicate rows: DISTINCT	7
Naming result columns: AS	7
Selecting rows using search conditions: WHERE	8
Putting the rows in order: ORDER BY	9
Specifying the sort key	10
Referencing derived columns	10
Summarizing group values: GROUP BY	11
Subjecting groups to conditions: HAVING	11
Merging lists of values: UNION	12
Using UNION to eliminate duplicates	13
Using UNION ALL to keep duplicates	13
Creating common table expressions: WITH	13
Using WITH instead of CREATE VIEW	14
Using common table expressions with CREATE VIEW	14
Using common table expressions when you use INSERT	15
Using recursive SQL	15
Accessing DB2 data that is not in a table	16
Using 15-digit and 31-digit precision for decimal numbers	16
Finding information in the DB2 catalog	17
Displaying a list of tables you can use	18
Displaying a list of columns in a table	18

Chapter 2. Working with tables and modifying data	19

Working with tables	19
Creating your own tables: CREATE TABLE	19
Working with temporary tables	21
Dropping tables: DROP TABLE	25
Working with views	25
Defining a view: CREATE VIEW	25
Changing data through a view	26
Dropping views: DROP VIEW	27
Modifying DB2 data	27
Inserting rows: INSERT	27
Selecting values as you insert: SELECT FROM INSERT	31
Updating current values: UPDATE	36
Deleting rows: DELETE	37

Chapter 3. Joining data from more than one table	39
Inner join	40
Full outer join	41
Left outer join	42
Right outer join	43
SQL rules for statements containing join operations	44
Using more than one join in an SQL statement	45
Using nested table expressions and user-defined table functions in joins	46
Using correlated references in table specifications in joins	47
Chapter 4. Using subqueries	49
Conceptual overview	49
Correlated and uncorrelated subqueries	50
Subqueries and predicates	50
The subquery result table.	50
Tables in subqueries of UPDATE, DELETE, and INSERT statements	51
How to code a subquery	51
Basic predicate	51
Quantified predicate : ALL, ANY, or SOME.	51
IN keyword	52
EXISTS keyword	53
Using correlated subqueries	53
An example of a correlated subquery.	53
Using correlation names in references	54
Using correlated subqueries in an UPDATE statement	55
Using correlated subqueries in a DELETE statement.	56
Chapter 5. Using SPUFI to execute SQL from your workstation	59
Allocating an input data set and using SPUFI	59
Changing SPUFI defaults	62
# Changing SPUFI defaults - panel 2	65
Entering SQL statements	66
Using the ISPF editor	66
# Retrieving Unicode UTF-16 graphic data	67
Entering comments	67
Setting the SQL terminator character	67
# Controlling toleration of warnings	68
Processing SQL statements	68
When SQL statements exceed resource limit thresholds.	68
Browsing the output	69
Format of SELECT statement results	70
Content of the messages	71
Part 2. Coding SQL in your host application program	73
Chapter 6. Basics of coding SQL in an application program	77
Conventions used in examples of coding SQL statements	78
Delimiting an SQL statement	78
Declaring table and view definitions	79
Accessing data using host variables, variable arrays, and structures	79
Using host variables	80
Using host variable arrays	86
Using host structures	90
Checking the execution of SQL statements	91
Using the SQL communication area (SQLCA)	91
SQLCODE and SQLSTATE	92
The WHENEVER statement	93
Handling arithmetic or conversion errors	93
The GET DIAGNOSTICS statement	94

Calling DSNTIAR to display SQLCA fields	98

Chapter 7. Using a cursor to retrieve a set of rows 103

Accessing data by using a row-positioned cursor	103
Step 1: Declare the cursor	103
Step 2: Open the cursor	105
Step 3: Specify what to do at end-of-data	105
Step 4: Execute SQL statements	106
Step 5: Close the cursor	108
Accessing data by using a rowset-positioned cursor	108
Step 1: Declare the rowset cursor	108
Step 2: Open the rowset cursor	108
Step 3: Specify what to do at end-of-data for a rowset cursor	109
Step 4: Execute SQL statements with a rowset cursor	109
Step 5: Close the rowset cursor	113
Types of cursors	113
Scrollable and non-scrollable cursors.	113
Held and non-held cursors	122
Examples of using cursors	124

Chapter 8. Generating declarations for your tables using DCLGEN 131

Invoking DCLGEN through DB2I	132
Including the data declarations in your program	136
DCLGEN support of C, COBOL, and PL/I languages	136
Example: Adding a table declaration and host-variable structure to a library	138
Step 1. Specify COBOL as the host language	138
Step 2. Create the table declaration and host structure.	139
Step 3. Examine the results.	140

Chapter 9. Embedding SQL statements in host languages 143

Coding SQL statements in an assembler application	143
Defining the SQL communications area.	143
Defining SQL descriptor areas.	144
Embedding SQL statements	145
Using host variables	148
Declaring host variables.	148
Determining equivalent SQL and assembler data types	151
Determining compatibility of SQL and assembler data types.	155
Using indicator variables	156
Handling SQL error return codes.	157
Macros for assembler applications	158
Coding SQL statements in a C or C++ application	158
Defining the SQL communication area	158
Defining SQL descriptor areas.	159
Embedding SQL statements	160
Using host variables and host variable arrays.	161
Declaring host variables.	162
Declaring host variable arrays.	168
Using host structures.	173
Determining equivalent SQL and C data types	175
Determining compatibility of SQL and C data types	181
Using indicator variables and indicator variable arrays	182
Handling SQL error return codes.	184
Coding considerations for C and C++	186
Coding SQL statements in a COBOL application.	186
Defining the SQL communication area	186
Defining SQL descriptor areas.	187
Embedding SQL statements	187
Using host variables and host variable arrays.	191
Declaring host variables.	192

	Declaring host variable arrays	199
	Using host structures	205
	Determining equivalent SQL and COBOL data types	210
	Determining compatibility of SQL and COBOL data types	214
	Using indicator variables and indicator variable arrays	216
	Handling SQL error return codes	217
	Coding considerations for object-oriented extensions in COBOL	219
	Coding SQL statements in a Fortran application	220
	Defining the SQL communication area	220
	Defining SQL descriptor areas	220
	Embedding SQL statements	221
	Using host variables	223
	Declaring host variables	223
	Determining equivalent SQL and Fortran data types	225
	Determining compatibility of SQL and Fortran data types	227
	Using indicator variables	228
	Handling SQL error return codes	229
	Coding SQL statements in a PL/I application	230
	Defining the SQL communication area	230
	Defining SQL descriptor areas	230
	Embedding SQL statements	231
	Using host variables and host variable arrays	233
	Declaring host variables	234
	Declaring host variable arrays	237
	Using host structures	240
	Determining equivalent SQL and PL/I data types	241
	Determining compatibility of SQL and PL/I data types	245
	Using indicator variables and indicator variable arrays	246
	Handling SQL error return codes	247
	Coding SQL statements in a REXX application	249
	Defining the SQL communication area	249
	Defining SQL descriptor areas	250
	Accessing the DB2 REXX Language Support application programming interfaces	250
	Embedding SQL statements in a REXX procedure	252
	Using cursors and statement names	253
	Using REXX host variables and data types	254
	Using indicator variables	257
	Setting the isolation level of SQL statements in a REXX procedure	258
	Chapter 10. Using constraints to maintain data integrity	259
	Using check constraints	259
	Check constraint considerations	259
	When check constraints are enforced	260
	How check constraints set CHECK-pending status	260
	Using referential constraints	261
	Parent key columns	261
	Defining a parent key and a unique index	262
	Defining a foreign key	264
	Referential constraints on tables with multilevel security with row-level granularity	266
	Using informational referential constraints	267
	Chapter 11. Using DB2-generated values as keys	269
	Using ROWID columns as keys	269
	Defining a ROWID column	269
	Direct row access	270
	Using identity columns as keys	270
	Defining an identity column	271
	Parent keys and foreign keys	272
	Using values obtained from sequence objects as keys	273
	Creating a sequence object	273

	Referencing a sequence object	274
	Keys across multiple tables	274

Chapter 12. Using triggers for active data 277

	Example of creating and using a trigger	277
	Parts of a trigger	279
	Trigger name	279
	Subject table.	279
	Trigger activation time	279
	Triggering event	279
	Granularity	280
	Transition variables	281
	Transition tables	282
	Triggered action	283
	Invoking stored procedures and user-defined functions from triggers.	285
	Passing transition tables to user-defined functions and stored procedures	286
	Trigger cascading	286
	Ordering of multiple triggers	287
	Interactions between triggers and referential constraints	288
	Interactions between triggers and tables that have multilevel security with row-level granularity	289
	Creating triggers to obtain consistent results	290

Part 3. Using DB2 object-relational extensions 293

Chapter 13. Introduction to DB2 object-relational extensions 295

Chapter 14. Programming for large objects 297

	Introduction to LOBs	297
	Declaring LOB host variables and LOB locators	300
	LOB materialization	305
	Using LOB locators to save storage	305
	Deferring evaluation of a LOB expression to improve performance	306
	Indicator variables and LOB locators	308
	Valid assignments for LOB locators	309
#	Avoiding character conversion for LOB locators	309

Chapter 15. Creating and using user-defined functions 311

	Overview of user-defined function definition, implementation, and invocation	311
	Example of creating and using a user-defined scalar function	312
	User-defined function samples shipped with DB2	313
	Defining a user-defined function	314
	Components of a user-defined function definition	314
	Examples of user-defined function definitions	316
	Implementing an external user-defined function.	318
	Writing a user-defined function	318
	Preparing a user-defined function for execution	350
	Testing a user-defined function	352
	Implementing an SQL scalar function	355
	Invoking a user-defined function	355
	Syntax for user-defined function invocation	355
	Ensuring that DB2 executes the intended user-defined function.	356
	Casting of user-defined function arguments	362
	What happens when a user-defined function abnormally terminates	363
	Nesting SQL statements	363
	Recommendations for user-defined function invocation	365

Chapter 16. Creating and using distinct types 367

	Introduction to distinct types	367
	Using distinct types in application programs	368

Comparing distinct types	368
Assigning distinct types	369
Using distinct types in UNIONS	371
Invoking functions with distinct types	371
Combining distinct types with user-defined functions and LOBs	372

Part 4. Designing a DB2 database application 377

Chapter 17. Planning for DB2 program preparation 381

Planning to process SQL statements	383
Planning to bind	384
Binding DBRMs with packages and plans	384
Planning for changes to your application	386

Chapter 18. Planning for concurrency 393

Definitions of concurrency and locks	393
Effects of DB2 locks	394
Suspension	394
Timeout	394
Deadlock	395
Basic recommendations to promote concurrency.	397
Recommendations for database design	398
Recommendations for application design	399
Aspects of transaction locks	402
The size of a lock	402
The duration of a lock	404
The mode of a lock	404
The object of a lock	407
Options for tuning locks.	408
Bind options.	408
Isolation overriding with SQL statements	421
The LOCK TABLE statement	422
Access paths	424
LOB locks	425
Relationship between transaction locks and LOB locks	425
Hierarchy of LOB locks	426
LOB and LOB table space lock modes	426
LOB lock and LOB table space lock duration	427
Instances when LOB table space locks are not taken	428
The LOCK TABLE statement for LOBs	428

Chapter 19. Planning for recovery 431

Unit of work in TSO batch and online	432
Unit of work in CICS.	432
Unit of work in IMS online programs	433
Planning ahead for program recovery: Checkpoint and restart	435
When are checkpoints important?	436
Checkpoints in MPPs and transaction-oriented BMPs	436
Checkpoints in batch-oriented BMPs	437
Specifying checkpoint frequency	438
Unit of work in DL/I and IMS batch programs	438
Commit and rollback coordination	438
Restart and recovery in IMS batch	439
Using savepoints to undo selected changes within a unit of work	439

Chapter 20. Planning to access distributed data 441

Planning for DRDA and DB2 private protocol access	441
Advantages of DRDA access	442
Moving from DB2 private protocol access to DRDA access	442

Bind processes for DRDA and DB2 private protocol access	444
Precompiler and bind options for DRDA access	445
Coding methods for distributed data	448
Using three-part table names to access distributed data	448
Using explicit CONNECT statements to access distributed data	450
Coordinating updates to two or more data sources	452
Working without two-phase commit.	452
Update restrictions on servers that do not support two-phase commit	453
Forcing update restrictions by using CONNECT (Type 1).	453
Maximizing performance for distributed data	454
Coding efficient queries	454
Maximizing LOB performance in a distributed environment.	454
Using bind options to improve performance for distributed applications	456
Using block fetch in distributed applications	458
Limiting the number of DRDA network transmissions.	461
Limiting the number of rows returned to DRDA clients	464
Working with distributed data.	465
SQL limitations at dissimilar servers.	465
Executing long SQL statements in a distributed environment	466
Retrieving data from ASCII or Unicode tables	466
Accessing data with a scrollable cursor when the requester is down-level	467
Accessing data with a rowset-positioned cursor when the requester is down-level	467
Maintaining data currency by using cursors	467
Copying a table from a remote location.	467
Transmitting mixed data.	467

Part 5. Developing your application 469

Chapter 21. Preparing an application program to run 471

Steps in program preparation	472
Step 1: Process SQL statements	473
Step 2: Compile (or assemble) and link-edit the application	494
Step 3: Bind the application	495
Step 4: Run the application.	508
Using JCL procedures to prepare applications	512
Available JCL procedures	512
Including code from SYSLIB data sets	513
Starting the precompiler dynamically	514
An alternative method for preparing a CICS program	516
Using JCL to prepare a program with object-oriented extensions	517
Using ISPF and DB2 Interactive	518
DB2I help	518
DB2I Primary Option Menu	518
DB2 Program Preparation panel	521
DB2I Defaults Panel 1	526
DB2I Defaults Panel 2	527
Precompile panel	528
Bind/Rebind/Free selection panel	531
Bind Package panel	533
Bind Plan panel	536
Rebind Package panel	539
Rebind Trigger Package panel	540
Rebind Plan panel.	542
Free Package panel	544
Free Plan panel.	545
The Defaults for Bind or Rebind Package or Plan panels	546
System Connection Types panel	550
Panels for entering lists of values.	552
Program Preparation: Compile, Link, and Run panel	553
The Run panel	555

Chapter 22. Testing an application program 557

Establishing a test environment	557
Designing a test data structure	557
Filling the tables with test data	559
Testing SQL statements using SPUFI.	560
Debugging your program	560
Debugging programs in TSO	560
Debugging programs in IMS	561
Debugging programs in CICS	562
Locating the problem.	566
Analyzing error and warning messages from the precompiler	567
SYSTEM output from the precompiler	567
SYSPRINT output from the precompiler	568

Chapter 23. Processing DL/I batch applications. 573

Planning to use DL/I batch applications	573
Features and functions of DB2 DL/I batch support.	573
Requirements for using DB2 in a DL/I batch job	574
Authorization	574
Program design considerations	574
Address spaces	574
Commits	575
SQL statements and IMS calls	575
Checkpoint calls	575
Application program synchronization	575
Checkpoint and XRST considerations	575
Synchronization call abends	576
Input and output data sets for DL/I batch jobs	576
DB2 DL/I batch input	576
DB2 DL/I batch output	578
Preparation guidelines for DL/I batch programs	578
Precompiling	578
Binding	578
Link-editing	579
Loading and running	579
Restart and recovery	580
JCL example of a batch backout	581
JCL example of restarting a DL/I batch job	581
Finding the DL/I batch checkpoint ID	582

Part 6. Additional programming techniques 583

Chapter 24. Coding dynamic SQL in application programs 593

Choosing between static and dynamic SQL	593
Flexibility of static SQL with host variables	594
Flexibility of dynamic SQL	594
Limitations of dynamic SQL	594
Dynamic SQL processing	594
Performance of static and dynamic SQL	595
Caching dynamic SQL statements	596
Using the dynamic statement cache	597
# Using the statement cache table	599
Keeping prepared statements after commit points	599
Limiting dynamic SQL with the resource limit facility	601
Writing an application to handle reactive governing	602
Writing an application to handle predictive governing.	602
Using predictive governing and down-level DRDA requesters	603
Using predictive governing and enabled requesters.	603
Choosing a host language for dynamic SQL applications	603
Dynamic SQL for non-SELECT statements.	603

	Dynamic execution using EXECUTE IMMEDIATE	604
	Dynamic execution using PREPARE and EXECUTE	605
I	Dynamic execution of a multiple-row INSERT statement	608
	Using DESCRIBE INPUT to put parameter information in an SQLDA	610
	Dynamic SQL for fixed-list SELECT statements	610
	Declaring a cursor for the statement name.	611
	Preparing the statement	611
	Opening the cursor	612
	Fetching rows from the result table	612
	Closing the cursor.	612
	Dynamic SQL for varying-list SELECT statements	613
	What your application program must do	613
	Preparing a varying-list SELECT statement	613
	Executing a varying-list SELECT statement dynamically	623
	Executing arbitrary statements with parameter markers	624
I	How bind options REOPT(ALWAYS) and REOPT(ONCE) affect dynamic SQL.	625
	Using dynamic SQL in COBOL	627
	Chapter 25. Using stored procedures for client/server processing.	629
	Introduction to stored procedures	629
	An example of a simple stored procedure	630
	Setting up the stored procedures environment	634
	Defining your stored procedure to DB2.	635
	Refreshing the stored procedures environment (for system administrators)	639
	Moving stored procedures to a WLM-established environment (for system administrators)	640
	Writing and preparing an external stored procedure	641
	Language requirements for the stored procedure and its caller	641
	Calling other programs	642
	Using reentrant code	642
	Writing a stored procedure as a main program or subprogram	643
	Restrictions on a stored procedure	646
	Using COMMIT and ROLLBACK statements in a stored procedure	646
	Using special registers in a stored procedure	647
	Accessing other sites in a stored procedure	649
	Writing a stored procedure to access IMS databases	650
	Writing a stored procedure to return result sets to a DRDA client	650
	Preparing a stored procedure	652
	Binding the stored procedure	653
	Writing a REXX stored procedure	654
	Writing and preparing an SQL procedure	657
	Comparison of an SQL procedure and an external procedure	658
	Statements that you can include in a procedure body	659
#	Declaring and using variables, parameters, and conditions in an SQL procedure	661
	Parameter style for an SQL procedure	662
	Terminating statements in an SQL procedure	662
	Handling SQL conditions in an SQL procedure	663
	Examples of SQL procedures	667
	Preparing an SQL procedure	669
	Writing and preparing an application to use stored procedures.	681
	Forms of the CALL statement	682
	Authorization for executing stored procedures	683
	Linkage conventions	684
	Using indicator variables to speed processing	703
	Declaring data types for passed parameters	703
	Writing a DB2 UDB for z/OS client program or SQL procedure to receive result sets	708
	Accessing transition tables in a stored procedure	714
	Calling a stored procedure from a REXX procedure	714
	Preparing a client program	718
	Running a stored procedure	719
	How DB2 determines which version of a stored procedure to run	720
	Using a single application program to call different versions of a stored procedure	720

	Running multiple stored procedures concurrently	722
#	Multiple instances of a stored procedure	722
	Accessing non-DB2 resources	723
	Testing a stored procedure	725
	Debugging the stored procedure as a stand-alone program on a workstation	725
	Debugging with the Debug Tool and IBM VisualAge COBOL	725
	Debugging an SQL procedure or C language stored procedure with the Debug Tool and C/C++ Productivity Tools for z/OS	726
	Debugging with Debug Tool for z/OS interactively and in batch mode	727
	Using the MSGFILE run-time option	728
	Using driver applications	729
	Using SQL INSERT statements	729

Chapter 26. Tuning your queries 731

	General tips and questions	731
	Is the query coded as simply as possible?	731
	Are all predicates coded correctly?	731
	Are there subqueries in your query?	732
	Does your query involve aggregate functions?	733
	Do you have an input variable in the predicate of an SQL query?	734
	Do you have a problem with column correlation?	734
	Can your query be written to use a noncolumn expression?	734
	Can materialized query tables help your query performance?	734
	Does the query contain encrypted data?	735
	Writing efficient predicates	735
	Properties of predicates	735
	Predicates in the ON clause	738
	General rules about predicate evaluation	739
	Order of evaluating predicates.	739
	Summary of predicate processing.	740
	Examples of predicate properties	745
	Predicate filter factors	746
	Column correlation	752
	DB2 predicate manipulation	755
	Predicates with encrypted data	759
	Using host variables efficiently	759
	Changing the access path at run time	760
	Rewriting queries to influence access path selection	763
	Writing efficient subqueries.	766
	Correlated subqueries	766
	Noncorrelated subqueries	767
	Conditions for DB2 to transform a subquery into a join	768
	Subquery tuning	770
	Using scrollable cursors efficiently	771
	Writing efficient queries on tables with data-partitioned secondary indexes.	772
	Special techniques to influence access path selection	774
	Obtaining information about access paths	775
	Fetching a limited number of rows: FETCH FIRST <i>n</i> ROWS ONLY	775
	Minimizing overhead for retrieving few rows: OPTIMIZE FOR <i>n</i> ROWS	776
	Favoring index access	778
#	Using a subsystem parameter to control outer join processing	779
	Using the CARDINALITY clause to improve the performance of queries with user-defined table function references	779
	Reducing the number of matching columns	780
	Creating indexes for efficient star-join processing	782
	Rearranging the order of tables in a FROM clause	784
	Updating catalog statistics	785
	Using a subsystem parameter	786

Chapter 27. Using EXPLAIN to improve SQL performance 789

Obtaining PLAN_TABLE information from EXPLAIN	790
Creating PLAN_TABLE	791
Populating and maintaining a plan table	798
Reordering rows from a plan table	799
Asking questions about data access	800
Is access through an index? (ACCESSTYPE is I, I1, N or MX)	801
Is access through more than one index? (ACCESSTYPE=M)	801
How many columns of the index are used in matching? (MATCHCOLS=n)	802
Is the query satisfied using only the index? (INDEXONLY=Y)	802
Is direct row access possible? (PRIMARY_ACCESSTYPE = D)	803
Is a view or nested table expression materialized?	806
Was a scan limited to certain partitions? (PAGE_RANGE=Y)	806
What kind of prefetching is expected? (PREFETCH = L, S, D, or blank)	807
Is data accessed or processed in parallel? (PARALLELISM_MODE is I, C, or X)	807
Are sorts performed?	807
Is a subquery transformed into a join?	808
When are aggregate functions evaluated? (COLUMN_FN_EVAL)	808
How many index screening columns are used?	808
Is a complex trigger WHEN clause used? (QBLOCKTYPE=TRIGGR)	809
Interpreting access to a single table	809
Table space scans (ACCESSTYPE=R PREFETCH=S)	809
Index access paths.	810
UPDATE using an index	815
Interpreting access to two or more tables (join)	815
Definitions and examples of join operations	815
Nested loop join (METHOD=1)	818
Merge scan join (METHOD=2)	820
Hybrid join (METHOD=4)	821
Star join (JOIN_TYPE='S')	823
Interpreting data prefetch	830
Sequential prefetch (PREFETCH=S)	830
Dynamic prefetch (PREFETCH=D)	831
List prefetch (PREFETCH=L)	831
Sequential detection at execution time	832
Determining sort activity	834
Sorts of data.	834
Sorts of RIDs	835
The effect of sorts on OPEN CURSOR	835
Processing for views and nested table expressions	836
Merge	836
Materialization	837
Using EXPLAIN to determine when materialization occurs	839
Using EXPLAIN to determine UNION activity and query rewrite	840
Performance of merge versus materialization	842
Estimating a statement's cost	842
Creating a statement table	843
Populating and maintaining a statement table	845
Retrieving rows from a statement table.	845
The implications of cost categories	846
Chapter 28. Parallel operations and query performance	847
Comparing the methods of parallelism	848
Enabling parallel processing	850
When parallelism is not used	851
Interpreting EXPLAIN output	852
A method for examining PLAN_TABLE columns for parallelism	852
PLAN_TABLE examples showing parallelism.	852
Tuning parallel processing	854
Disabling query parallelism	855
Chapter 29. Programming for the Interactive System Productivity Facility	857

Using ISPF and the DSN command processor	857
Invoking a single SQL program through ISPF and DSN	858
Invoking multiple SQL programs through ISPF and DSN.	859
Invoking multiple SQL programs through ISPF and CAF.	859

Chapter 30. Programming for the call attachment facility. 861

CAF capabilities and requirements	861
CAF capabilities	861
CAF requirements.	863
How to use CAF	864
Summary of connection functions	866
Accessing the CAF language interface	867
General properties of CAF connections	868
CAF function descriptions	869
CONNECT: Syntax and usage.	871
OPEN: Syntax and usage	875
CLOSE: Syntax and usage	877
DISCONNECT: Syntax and usage	878
TRANSLATE: Syntax and usage	880
Summary of CAF behavior.	881
Sample scenarios	882
A single task with implicit connections	882
A single task with explicit connections	883
Several tasks	883
Exit routines from your application	883
Attention exit routines	883
Recovery routines	884
Error messages and dsnttrace	884
CAF return codes and reason codes	884
Program examples for CAF.	885
Sample JCL for using CAF	886
Sample assembler code for using CAF	886
Loading and deleting the CAF language interface	886
Connecting to DB2 for CAF	886
Checking return codes and reason codes for CAF	888
Using dummy entry point DSNHLI for CAF	890
Variable declarations for CAF	891

Chapter 31. Programming for the Resource Recovery Services attachment facility . . 893

RRSAF capabilities and requirements	893
RRSAF capabilities	893
RRSAF requirements	894
How to use RRSAF	896
Summary of connection functions	896
Implicit connections	897
Accessing the RRSAF language interface	898
General properties of RRSAF connections	900
Summary of RRSAF behavior	902
RRSAF function descriptions	903
Register conventions	903
Parameter conventions for function calls	904
IDENTIFY: Syntax and usage	904
SWITCH TO: Syntax and usage	907
SIGNON: Syntax and usage	909
AUTH SIGNON: Syntax and usage	914
CONTEXT SIGNON: Syntax and usage.	918
SET_ID: Syntax and usage	922
SET_CLIENT_ID: Syntax and usage	923
CREATE THREAD: Syntax and usage	926
TERMINATE THREAD: Syntax and usage.	928

TERMINATE IDENTIFY: Syntax and usage	930
TRANSLATE: Syntax and usage	931
RRSAF connection examples	932
Example of a single task.	932
Example of multiple tasks	933
Example of calling SIGNON to reuse a DB2 thread.	933
Example of switching DB2 threads between tasks	933
RRSAF return codes and reason codes	934
Program examples for RRSAF.	935
Sample JCL for using RRSAF	935
Loading and deleting the RRSAF language interface	935
Using dummy entry point DSNHLI for RRSAF	935
Connecting to DB2 for RRSAF.	936
Chapter 32. CICS-specific programming techniques.	939
Controlling the CICS attachment facility from an application	939
Improving thread reuse	939
Detecting whether the CICS attachment facility is operational	939
 Chapter 33. WebSphere MQ with DB2	941
WebSphere MQ messages	941
WebSphere MQ message handling	941
# WebSphere MQ functions and stored procedures	944
Commit environment for AMI-based DB2 MQ functions and stored procedures	948
# DB2 MQ tables.	949
# Converting applications to use the MQI functions	958
How to use WebSphere MQ functions	959
# Asynchronous messaging in DB2 UDB for z/OS and OS/390	966
# MQListener in DB2 for OS/390 and z/OS.	967
# Configuring and running MQListener in DB2 UDB for OS/390 and z/OS	968
# Configuring MQListener tasks.	971
# MQListener error processing	972
# Creating a sample stored procedure to use with MQListener	973
# MQListener examples	974
Chapter 34. Using DB2 as a web services consumer and provider	977
DB2 as a web services consumer	977
The SOAPHTTPV and SOAPHTTPC user-defined functions.	977
# The SOAPHTTPNV and SOAPHTTPNC user-defined functions	978
SQLSTATEs for DB2 as a web services consumer	979
DB2 as a web services provider	980
Chapter 35. Programming techniques: Questions and answers	983
Providing a unique key for a table	983
Scrolling through previously retrieved data	983
Using a scrollable cursor	983
Using a ROWID or identity column	984
Scrolling through a table in any direction	985
Updating data as it is retrieved from the database	986
Updating previously retrieved data	986
Updating thousands of rows	986
Retrieving thousands of rows	987
Using SELECT *	987
Optimizing retrieval for a small set of rows	987
Adding data to the end of a table	988
Translating requests from end users into SQL statements.	988
Changing the table definition	988
Storing data that does not have a tabular format	989
Finding a violated referential or check constraint	989

Part 7. Appendixes 991

Appendix A. DB2 sample tables 993

Activity table (DSN8810.ACT)	993
Department table (DSN8810.DEPT)	994
Employee table (DSN8810.EMP)	996
Employee photo and resume table (DSN8810.EMP_PHOTO_RESUME)	999
Project table (DSN8810.PROJ)	1000
Project activity table (DSN8810.PROJACT)	1001
Employee to project activity table (DSN8810.EMPPROJACT)	1002
Unicode sample table (DSN8810.DEMO_UNICODE)	1003
Relationships among the sample tables	1004
Views on the sample tables	1004
Storage of sample application tables	1009
Storage group	1010
Databases	1010
Table spaces	1010

Appendix B. Sample applications 1013

Types of sample applications	1013
Using the sample applications	1015
TSO	1016
IMS	1018
CICS	1018

Appendix C. Running the productivity-aid sample programs 1019

Running DSNTIAUL	1020
Running DSNTIAD	1024
Running DSNTPE2 and DSNTPE4	1026

Appendix D. Programming examples. 1031

Sample COBOL dynamic SQL program	1031
Pointers and based variables	1031
Storage allocation	1031
Example	1032
Sample dynamic and static SQL in a C program	1043
Sample DB2 REXX application	1047
Sample COBOL program using DRDA access	1061
Sample COBOL program using DB2 private protocol access	1069
Examples of using stored procedures	1075
Calling a stored procedure from a C program	1075
Calling a stored procedure from a COBOL program	1079
Calling a stored procedure from a PL/I program	1082
C stored procedure: GENERAL	1083
C stored procedure: GENERAL WITH NULLS	1085
COBOL stored procedure: GENERAL	1088
COBOL stored procedure: GENERAL WITH NULLS	1091
PL/I stored procedure: GENERAL	1093
PL/I stored procedure: GENERAL WITH NULLS	1094

| Appendix E. Recursive common table expression examples. 1097

Appendix F. REBIND subcommands for lists of plans or packages 1103

Overview of the procedure for generating lists of REBIND commands	1103
Sample SELECT statements for generating REBIND commands	1103
Sample JCL for running lists of REBIND commands	1106

Appendix G. Reserved schema names and reserved words 1109

Reserved schema names	1109

Reserved words	1109
Appendix H. Characteristics of SQL statements in DB2 UDB for z/OS.	1113
Actions allowed on SQL statements	1113
SQL statements allowed in external functions and stored procedures	1116
SQL statements allowed in SQL procedures	1118
Appendix I. Program preparation options for remote packages	1123
Appendix J. DB2-supplied stored procedures	1127
WLM environment refresh stored procedure (WLM_REFRESH)	1129
Environment for WLM_REFRESH	1129
Authorization required for WLM_REFRESH.	1129
WLM_REFRESH syntax diagram	1130
WLM_REFRESH option descriptions	1130
Example of WLM_REFRESH invocation	1131
The CICS transaction invocation stored procedure (DSNACICS)	1131
Environment for DSNACICS	1132
Authorization required for DSNACICS	1132
DSNACICS syntax diagram	1132
DSNACICS option descriptions	1133
DSNACICX user exit routine	1135
Example of DSNACICS invocation	1137
DSNACICS output	1138
DSNACICS restrictions	1139
DSNACICS debugging	1139
# IMS transactions stored procedure (DSNAIMS)	1139
# Environment for DSNAIMS	1139
# Authorization required for DSNAIMS	1139
# DSNAIMS syntax diagram	1139
# DSNAIMS option descriptions	1140
# Examples of DSNAIMS invocation	1142
# Connecting to multiple IMS subsystems with DSNAIMS	1143
# The DB2 EXPLAIN stored procedure	1143
# Environment	1143
# Authorization required	1144
# DSNAEXP syntax diagram	1144
# DSNAEXP option descriptions	1144
# Example of DSNAEXP invocation	1145
# DSNAEXP output	1146
# Deprecated: Store an XML document from an MQ message queue in DB2 tables (DXXMQINSERT)	1146
Environment for DXXMQINSERT	1146
Authorization required for DXXMQINSERT	1146
DXXMQINSERT syntax diagram	1147
DXXMQINSERT option descriptions	1147
Example of DXXMQINSERT invocation	1147
DXXMQINSERT output	1148
# Deprecated: Store an XML document from an MQ message queue in DB2 tables (DXXMQSHRED)	1148
# Environment for DXXMQSHRED	1149
# Authorization required for DXXMQSHRED	1149
# DXXMQSHRED syntax diagram.	1149
# DXXMQSHRED option descriptions	1149
# Example of DXXMQSHRED invocation	1150
# DXXMQSHRED output.	1151
# Deprecated: Store a large XML document from an MQ message queue in DB2 tables (DXXMQINSERTCLOB)	1151
# Environment for DXXMQINSERTCLOB	1151
# Authorization required for DXXMQINSERTCLOB	1151
# DXXMQINSERTCLOB syntax diagram	1151
# DXXMQINSERTCLOB option descriptions	1152
# Example of DXXMQINSERTCLOB invocation	1152

# DXXMQINSERTCLOB output	1153
# Deprecated: Store a large XML document from an MQ message queue in DB2 tables (DXXMQSHREDCLOB)	1153
# Environment for DXXMQSHREDCLOB	1154
# Authorization required for DXXMQSHREDCLOB	1154
# DXXMQSHREDCLOB syntax diagram.	1154
# DXXMQSHREDCLOB option descriptions	1154
# Example of DXXMQSHREDCLOB invocation	1155
# DXXMQSHREDCLOB output.	1156
# Deprecated: Store XML documents from an MQ message queue in DB2 tables (DXXMQINSERTALL)	1156
# Environment for DXXMQINSERTALL	1156
# Authorization required for DXXMQINSERTALL	1156
# DXXMQINSERTALL syntax diagram	1156
# DXXMQINSERTALL option descriptions	1157
# Example of DXXMQINSERTALL invocation	1157
# DXXMQINSERTALL output	1158
# Deprecated: Store XML documents from an MQ message queue in DB2 tables (DXXMQSHREDALL)	1158
# Environment for DXXMQSHREDALL	1158
# Authorization required for DXXMQSHREDALL	1159
# DXXMQSHREDALL syntax diagram	1159
# DXXMQSHREDALL option descriptions	1159
# Example of DXXMQSHREDALL invocation	1160
# DXXMQSHREDALL output	1160
# Deprecated: Store large XML documents from an MQ message queue in DB2 tables (DXXMQSHREDALLCLOB)	1161
# Environment for DXXMQSHREDALLCLOB	1161
# Authorization required for DXXMQSHREDALLCLOB	1161
# DXXMQSHREDALLCLOB syntax diagram	1161
# DXXMQSHREDALLCLOB option descriptions	1161
# Example of DXXMQSHREDALLCLOB invocation	1162
# DXXMQSHREDALLCLOB output	1163
# Deprecated: Store large XML documents from an MQ message queue in DB2 tables (DXXMQINSERTALLCLOB)	1163
# Environment for DXXMQINSERTALLCLOB	1163
# Authorization required for DXXMQINSERTALLCLOB	1163
# DXXMQINSERTALLCLOB syntax diagram	1164
# DXXMQINSERTALLCLOB option descriptions	1164
# Example of DXXMQINSERTALLCLOB invocation	1165
# DXXMQINSERTALLCLOB output	1165
# Deprecated: Send XML documents to an MQ message queue (DXXMQGEN)	1165
# Environment for DXXMQGEN	1166
# Authorization required for DXXMQGEN	1166
# DXXMQGEN syntax diagram	1166
# DXXMQGEN option descriptions	1166
# Example of DXXMQGEN invocation	1168
# DXXMQGEN output	1169
# Deprecated: Send XML documents to an MQ message queue (DXXMQRETRIEVE)	1169
# Environment for DXXMQRETRIEVE	1169
# Authorization required for DXXMQRETRIEVE	1169
# DXXMQRETRIEVE syntax diagram	1169
# DXXMQRETRIEVE option descriptions	1170
# Example of DXXMQRETRIEVE invocation	1171
# DXXMQRETRIEVE output	1173
# Deprecated: Send large XML documents to an MQ message queue (DXXMQGENCLOB)	1173
# Environment for DXXMQGENCLOB	1173
# Authorization required for DXXMQGENCLOB	1173
# DXXMQGENCLOB syntax diagram	1173
# DXXMQGENCLOB option descriptions	1174
# Example of DXXMQGENCLOB invocation	1175
# DXXMQGENCLOB output	1176
# Deprecated: Send XML documents to an MQ message queue (DXXMQRETRIEVECLOB)	1176
# Environment for DXXMQRETRIEVECLOB	1176
# Authorization required for DXXMQRETRIEVECLOB	1177
# DXXMQRETRIEVECLOB syntax diagram	1177

#	DXXMQRETRIEVECLOB option descriptions	1177
#	Example of DXXMQRETRIEVECLOB invocation	1179
#	DXXMQRETRIEVECLOB output	1180
Appendix K. How to use the DB2 library		1181
Notices		1183
	Programming interface information.	1184
	Trademarks.	1185
Glossary		1187
Bibliography		1221
Index		X-1

About this book

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

Unless it is stated otherwise, this information assumes that DB2® is running in new-function mode (as opposed to compatibility mode or enabling-new-function mode).

Visit the following Web site for information about ordering DB2 books and obtaining other valuable information about DB2 UDB for z/OS:
<http://publib.boulder.ibm.com/infocenter/imzic>

Important

In this version of DB2 UDB 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. See Part 1 of *DB2 Utility Guide and Reference* for packaging details.

The DB2 Utilities Suite is designed to work with the DFSORT program, which you are licensed to use 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.

Who should read this book

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

Terminology and citations

In this information, DB2 Universal Database™ for z/OS® is referred to as "DB2 UDB for z/OS." In cases where the context makes the meaning clear, DB2 UDB for z/OS is referred to as "DB2." When this information refers to titles of books in this library, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to IBM® *DB2 Universal Database for z/OS SQL Reference*.)

When referring to a DB2 product other than DB2 UDB 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.

#

OMEGAMON

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 or CICS Transaction Server for OS/390®.

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.

How to read the syntax diagrams

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

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

The ►— symbol indicates the beginning of a statement.

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

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

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

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

►—*required_item*—>◄

- Optional items appear below the main path.

►—*required_item*—
 └*optional_item*—>◄

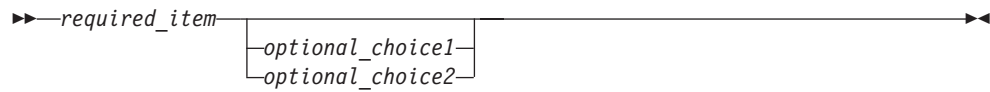
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

►—*required_item*—
 └*optional_item*—>◄

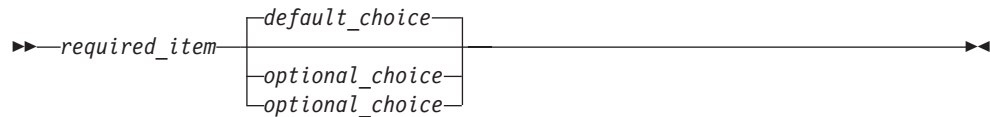
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.

►—*required_item*—
 └*required_choice1*—
 └*required_choice2*—>◄

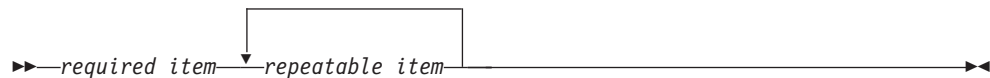
If choosing one of the items is optional, the entire stack appears below the main path.



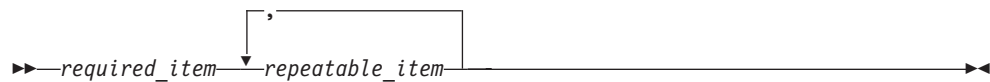
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



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



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



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

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

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products. The major accessibility features in z/OS products, including DB2 UDB for z/OS, enable users to:

- Use assistive technologies such as screen reader and screen magnifier software
- Operate specific or equivalent features by using only a keyboard
- Customize display attributes such as color, contrast, and font size

Assistive technology products, such as screen readers, function with the DB2 UDB for z/OS user interfaces. Consult the documentation for the assistive technology products for specific information when you use assistive technology to access these interfaces.

Online documentation for Version 8 of DB2 UDB for z/OS is available in the Information management software for z/OS solutions information center, which is an accessible format when used with assistive technologies such as screen reader

or screen magnifier software. The Information management software for z/OS solutions information center is available at the following Web site:
<http://publib.boulder.ibm.com/infocenter/dzichelp>

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 UDB for z/OS documentation. You can use the following methods to provide comments:

- Send your comments by e-mail 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 send comments from the Web. Visit the library Web site at:

www.ibm.com/software/db2zos/library.html

This Web site has a an online reader comment form that you can use to send comments.

- 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://publib.boulder.ibm.com/infocenter/db2zhhelp>.

Summary of changes to this book

The principal changes to this book are:

- Chapter 1, “Retrieving data,” on page 3 explains how to create and use common table expressions in SELECT, CREATE VIEW, and INSERT statements, and also describes how to use common table expressions to create recursive SQL.
- Chapter 2, “Working with tables and modifying data,” on page 19 explains how to select column values as you insert rows into a table by using the SELECT from INSERT statement.
- Chapter 6, “Basics of coding SQL in an application program,” on page 77 contains information on how to use:
 - Host variable arrays, and their indicator arrays, in a multiple-row INSERT statement (in a C or C++, COBOL, or PL/I program).
 - The GET DIAGNOSTICS statement to return diagnostic information about the last SQL statement that was executed (for example, information about input data errors during the execution of a multiple-row INSERT statement).
- Chapter 7, “Using a cursor to retrieve a set of rows,” on page 103 explains how to use:
 - Static and dynamic scrollable cursors.
 - A rowset-positioned cursor in a multiple-row FETCH statement (in a C or C++, COBOL, or PL/I program).
 - Positioned updates and deletes with a rowset-positioned cursor.
- Chapter 9, “Embedding SQL statements in host languages,” on page 143 contains information on how to declare host variable arrays (for C or C++, COBOL, and PL/I) for use with multiple-row INSERT and FETCH statements.
- Chapter 10, “Using constraints to maintain data integrity,” on page 259 describes informational referential constraints (not enforced by DB2), describes referential constraints on tables with multi-level security with row-level granularity, and explains how to maintain referential integrity when using data encryption.
- Chapter 11, “Using DB2-generated values as keys,” on page 269 is a new chapter that describes the use of ROWID columns for direct row access, identity columns as parent keys and foreign keys, and values generated from sequence objects as keys across multiple tables.
- Chapter 12, “Using triggers for active data,” on page 277 describes interactions between triggers and tables that use multi-level security with row-level granularity.
- Chapter 21, “Preparing an application program to run,” on page 471 describes the new SQL processing options:
 - CCSID, which specifies the CCSID in which the source program is written.
 - NEWFUN, which indicates whether to accept the syntax for DB2 Version 8 new functions.
 - For C programs, PADNSTR or NOPADNSTR, which indicates whether or not output host variables that are NUL-terminated strings are padded with blanks.

This chapter also describes how the CURRENT PACKAGE PATH special register is used in identifying the collection for packages at run time.
- Chapter 24, “Coding dynamic SQL in application programs,” on page 593 describes how to use a descriptor when you prepare and execute a multiple-row

| INSERT statement. This chapter also includes information about how bind
| option REOPT(ONCE) affects dynamic SQL statements.

- | • Chapter 25, “Using stored procedures for client/server processing,” on page 629
| describes how to invoke DSNTPSMP (the SQL Procedure Processor that prepares
| SQL procedures for execution) with the SQL CALL statement. This chapter also
| describes new SQL procedure statements and describes how to run multiple
| instances of the same stored procedure at the same time.
- | • Chapter 31, “Programming for the Resource Recovery Services attachment
| facility,” on page 893 contains information about using implicit connections to
| DB2 when applications include SQL statements.
- | • Chapter 33, “WebSphere MQ with DB2,” on page 941 is a new chapter that
| describes how to use DB2 WebSphere® MQ functions in SQL statements to
| combine DB2 database access with WebSphere MQ message handling.
- | • Appendix E, “Recursive common table expression examples,” on page 1097 is a
| new appendix that includes examples of using common table expressions to
| create recursive SQL in a bill of materials application.

Part 1. Using SQL queries

Chapter 1. Retrieving data	3
Result tables	3
Data types	4
Selecting columns: SELECT	5
Selecting all columns: SELECT *	5
Selecting some columns: SELECT column-name	6
Selecting derived columns: SELECT expression	7
Eliminating duplicate rows: DISTINCT	7
Naming result columns: AS	7
Selecting rows using search conditions: WHERE	8
Putting the rows in order: ORDER BY	9
Specifying the sort key	10
Referencing derived columns	10
Summarizing group values: GROUP BY	11
Subjecting groups to conditions: HAVING	11
Merging lists of values: UNION	12
Using UNION to eliminate duplicates	13
Using UNION ALL to keep duplicates	13
Creating common table expressions: WITH	13
Using WITH instead of CREATE VIEW	14
Using common table expressions with CREATE VIEW	14
Using common table expressions when you use INSERT	15
Using recursive SQL	15
Accessing DB2 data that is not in a table	16
Using 15-digit and 31-digit precision for decimal numbers.	16
Finding information in the DB2 catalog	17
Displaying a list of tables you can use	18
Displaying a list of columns in a table	18
 Chapter 2. Working with tables and modifying data	 19
Working with tables	19
Creating your own tables: CREATE TABLE	19
Identifying defaults.	19
Creating work tables	20
Creating a new department table	20
Creating a new employee table	21
Working with temporary tables.	21
Working with created temporary tables	22
Working with declared temporary tables.	23
Dropping tables: DROP TABLE.	25
Working with views	25
Defining a view: CREATE VIEW	25
Changing data through a view	26
Dropping views: DROP VIEW	27
Modifying DB2 data	27
Inserting rows: INSERT	27
Inserting a single row	28
Inserting rows into a table from another table	29
Other ways to insert data.	30
Inserting data into a ROWID column.	30
Inserting data into an identity column	30
Selecting values as you insert: SELECT FROM INSERT	31
Result table of the INSERT operation	32
Selecting values when you insert a single row	32
Selecting values when you insert data into a view	33
Selecting values when you insert multiple rows	33

	Result table of the cursor when you insert multiple rows	34
	What happens if an error occurs	35
	Updating current values: UPDATE	36
	Deleting rows: DELETE	37
	Deleting every row in a table	38
	Chapter 3. Joining data from more than one table	39
	Inner join	40
	Full outer join	41
	Left outer join	42
	Right outer join	43
	SQL rules for statements containing join operations	44
	Using more than one join in an SQL statement	45
	Using nested table expressions and user-defined table functions in joins	46
	Using correlated references in table specifications in joins	47
	Chapter 4. Using subqueries	49
	Conceptual overview	49
	Correlated and uncorrelated subqueries	50
	Subqueries and predicates	50
	The subquery result table.	50
	Tables in subqueries of UPDATE, DELETE, and INSERT statements	51
	How to code a subquery	51
	Basic predicate	51
	Quantified predicate : ALL, ANY, or SOME.	51
	Using the ALL predicate	52
	Using the ANY or SOME predicate	52
	IN keyword	52
	EXISTS keyword	53
	Using correlated subqueries	53
	An example of a correlated subquery.	53
	Using correlation names in references	54
	Using correlated subqueries in an UPDATE statement	55
	Using correlated subqueries in a DELETE statement.	56
	Using tables with no referential constraints.	56
	Using a single table.	56
	Using tables with referential constraints	56
	Chapter 5. Using SPUFI to execute SQL from your workstation.	59
	Allocating an input data set and using SPUFI	59
	Changing SPUFI defaults.	62
#	Changing SPUFI defaults - panel 2	65
	Entering SQL statements	66
	Using the ISPF editor	66
#	Retrieving Unicode UTF-16 graphic data	67
	Entering comments	67
	Setting the SQL terminator character	67
#	Controlling toleration of warnings	68
	Processing SQL statements	68
	When SQL statements exceed resource limit thresholds.	68
	Browsing the output	69
	Format of SELECT statement results	70
	Content of the messages	71

Chapter 1. Retrieving data

You can retrieve data using the SQL statement `SELECT` to specify a result table. This chapter describes how to interactively use `SELECT` statements to retrieve data from DB2 tables. It includes the following sections:

- “Result tables”
- “Data types” on page 4
- “Selecting columns: `SELECT`” on page 5
- “Selecting rows using search conditions: `WHERE`” on page 8
- “Putting the rows in order: `ORDER BY`” on page 9
- “Summarizing group values: `GROUP BY`” on page 11
- “Merging lists of values: `UNION`” on page 12
- “Creating common table expressions: `WITH`” on page 13
- “Accessing DB2 data that is not in a table” on page 16
- “Using 15-digit and 31-digit precision for decimal numbers” on page 16
- “Finding information in the DB2 catalog” on page 17

For more advanced topics on using `SELECT` statements, see Chapter 4, “Using subqueries,” on page 49, and Chapter 20, “Planning to access distributed data,” on page 441.

Examples of SQL statements illustrate the concepts that this chapter discusses. Consider developing SQL statements similar to these examples and then running them dynamically using `SPUFI` or DB2 Query Management Facility (DB2 QMF).

Result tables

The data retrieved through SQL 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: `SELECT` statement: The following `SELECT` statement retrieves the last name, first name, and phone number of employees in department D11 from the sample employee table:

```
SELECT LASTNAME, FIRSTNAME, PHONENO
FROM DSN8810.EMP
WHERE WORKDEPT = 'D11'
ORDER BY LASTNAME;
```

The result table looks similar to the following output:

LASTNAME	FIRSTNAME	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

Data types

When you create a DB2 table, you define each column to have a specific data type. The data type can be a built-in data type or a distinct type. This section discusses built-in data types. For information about distinct types, see Chapter 16, “Creating and using distinct types,” on page 367. 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, like a last name, into a column whose data type is numeric. Similarly, you cannot compare columns containing incompatible data types.

To better understand the concepts that are presented in this chapter, you must understand the data types of the columns to which an example refers. As shown in Figure 1, built-in data types have four general categories: datetime, string, numeric, and row identifier (ROWID).

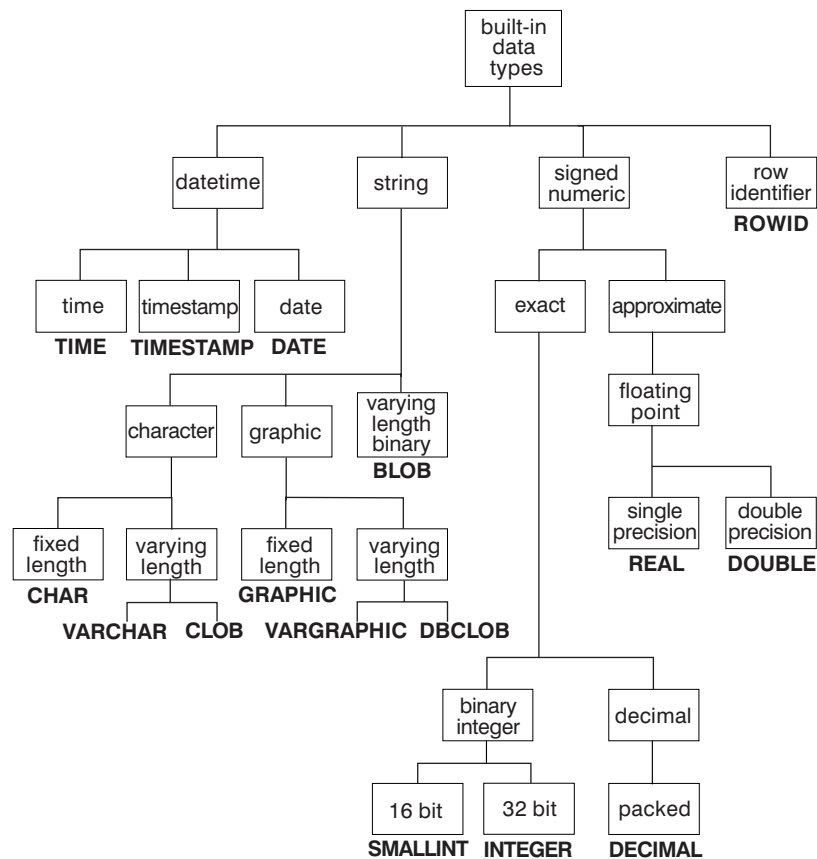


Figure 1. DB2 data types

For more detailed information about each data type, see Chapter 2 of *DB2 SQL Reference*.

Table 1 on page 5 shows whether operands of any two data types are compatible, Y (Yes), or incompatible, N (No). Numbers in the table, either as superscript of Y or N, or as a value in the column, indicates a note at the bottom of the table.

Table 1. Compatibility of data types for assignments and comparisons. Y indicates that the data types are compatible. N indicates no compatibility. For any number in a column, read the corresponding note at the bottom of the table.

Operands	Binary integer	Decimal number	Floating point	Character string	Graphic string	Binary string	Date	Time	Time-stamp	Row ID	Distinct type
Binary Integer	Y	Y	Y	N	N	N	N	N	N	N	2
Decimal Number	Y	Y	Y	N	N	N	N	N	N	N	2
Floating Point	Y	Y	Y	N	N	N	N	N	N	N	2
Character String	N	N	N	Y	Y ^{4,5}	N ³	1	1	1	N	2
Graphic String	N	N	N	Y ^{4,5}	Y	N	1,4	1,4	1,4	N	2
Binary String	N	N	N	N ³	N	Y	N	N	N	N	2
Date	N	N	N	1	1,4	N	Y	N	N	N	2
Time	N	N	N	1	1,4	N	N	Y	N	N	2
Timestamp	N	N	N	1	1,4	N	N	N	Y	N	2
Row ID	N	N	N	N	N	N	N	N	N	Y	2
Distinct Type	2	2	2	2	2	2	2	2	2	2	Y ²

Notes:

- The compatibility of datetime values is limited to assignment and comparison:
 - Datetime values can be assigned to string columns and to string variables, as explained in Chapter 2 of *DB2 SQL Reference*.
 - A valid string representation of a date can be assigned to a date column or compared to a date.
 - A valid string representation of a time can be assigned to a time column or compared to a time.
 - A valid string representation of a timestamp can be assigned to a timestamp column or compared to a timestamp.
- A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, DB2 supports assignments between a distinct type value and its source data type. For additional information, see Chapter 2 of *DB2 SQL Reference*.
- All character strings, even those with subtype FOR BIT DATA, are not compatible with binary strings.
- On assignment and comparison from Graphic to Character, the resulting length in bytes is 3 * (LENGTH(graphic string)), depending on the CCSIDs.
- Character strings with subtype FOR BIT DATA are not compatible with Graphic Data.

Selecting columns: SELECT

You have several options for selecting columns from a database for your result tables. This section describes how to select columns using a variety of techniques.

Selecting all columns: SELECT *

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.

Example: SELECT *: The following SQL statement selects all columns from the department table:

```
SELECT *
FROM DSN8810.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	ADMINSTRATION 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 this is not recommended; if you add a column to the table to which SELECT * refers, the program might reference columns for which you have not defined receiving host variables. For more information about host variables, see “Accessing data using host variables, variable arrays, and structures” on page 79.

If you list the column names in a static SELECT statement instead of using an asterisk, you can avoid the problem created by using 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 selects only the MGRNO and DEPTNO columns from the department table:

```
SELECT MGRNO, DEPTNO
FROM DSN8810.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.

Selecting derived columns: SELECT expression

You can select columns 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 DSN8810.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: AS.”

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 “Putting the rows in order: ORDER BY” on page 9.

Eliminating duplicate rows: DISTINCT

The DISTINCT keyword removes 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 administrating departments:

```
SELECT DISTINCT ADMRDEPT
FROM DSN8810.DEPT;
```

The result table looks similar to the following output:

```
ADMRDEPT
=====
A00
D01
E01
```

Naming result columns: AS

With the AS clause, you can name result columns in a SELECT statement. This is particularly useful for a column that is derived from an expression or a function. For syntax and more information about the AS clause, see Chapter 4 of *DB2 SQL Reference*.

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 DSN8810.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 DSN8810.EMP;
```

For more information about using the CREATE VIEW statement, see “Defining a view: CREATE VIEW” on page 25.

Example: UNION ALL with AS clause: You can use the AS clause 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 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, and 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
.		
.		
.		

For information about unions, see “Merging lists of values: UNION” on page 12.

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 DSN8810.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. For more information about using the GROUP BY clause, see “Summarizing group values: GROUP BY” on page 11.

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 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. See Chapter 16, “Creating and using distinct types,” on page 367 for more information about distinct types.

Table 2 lists the type of comparison, the comparison operators, and an example of how each type of comparison that you can use in a predicate in a WHERE clause.

Table 2. 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 or one value is equal to null	IS DISTINCT FROM	PHONENO IS DISTINCT FROM :PHONEHV
Similar to another value	LIKE	NAME LIKE '%SMITH%' 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. For more information about predicates, see Chapter 2 of <i>DB2 SQL Reference</i> .		

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* IS NOT DISTINCT FROM *value*
- NOT(*value* IS DISTINCT FROM *value*)

Both of these forms of the predicate create an expression where one value is equal to another value or both values are equal to null.

Putting the rows in order: ORDER BY

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 sections show you how to use the ORDER BY clause.

Specifying the sort key

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.

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 DSN8810.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 DSN8810.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

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 using a derived column name: The following SQL statement orders the selected information by total salary:

```
SELECT EMPNO, (SALARY + BONUS + COMM) AS TOTAL_SAL
FROM DSN8810.EMP
ORDER BY TOTAL_SAL;
```

Summarizing group values: GROUP BY

Use GROUP BY to group rows by the values of one or more columns or by the results of an expression. You can then apply aggregate functions to each group.

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 DSN8810.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 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 DSN8810.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 DSN8810.EMP
GROUP BY SUBSTR(WORKDEPT,1,1);
```

Subjecting groups to conditions: HAVING

Use HAVING to specify a search condition that each retrieved group must satisfy. The HAVING clause acts like a WHERE clause for groups, and contains the same kind of search conditions you specify in a WHERE clause. The search condition in the HAVING clause tests properties of each group rather than properties of individual rows in the group.

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 DSN8810.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: GROUP BY” on page 11. 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 you only want results from 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 DSN8810.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 and OR, and you can use NOT for any predicate of a search condition.

Merging lists of values: UNION

Using the UNION keyword, you can combine two or more SELECT statements to
 # form a single result table. When DB2 encounters the UNION keyword, it processes
 # each SELECT statement to form an interim result table, and then combines the
 # interim result table of each statement. If you use UNION to combine two columns
 # with the same name, the result table inherits that name. If the two columns do not
 # have the same name, and you do not use labels, the column in the result set is

unnamed. When you use the UNION statement, the SQLNAME field of the
SQLDA contains the unqualified name or label of the column, or a string of length
zero if the name or label does not exist.

Using UNION to eliminate duplicates

You can use UNION to eliminate duplicates when merging lists of values obtained from several tables.

Example: UNION clause: You can obtain a combined list of employee numbers that includes both of the following:

- People in department D11
- People whose assignments include projects MA2112, MA2113, and AD3111.

The following SQL statement gives a combined result table containing employee numbers in ascending order with no duplicates listed:

```
SELECT EMPNO
  FROM DSN8810.EMP
 WHERE WORKDEPT = 'D11'
UNION
SELECT EMPNO
  FROM DSN8810.EMPPROJACT
 WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
 ORDER BY EMPNO;
```

If you have an ORDER BY clause, it must appear after the last SELECT statement that is part of the union. In this example, the first column of the final result table determines the final order of the rows.

Using UNION ALL to keep duplicates

If you want to keep duplicates in the final result table of a UNION, specify the optional keyword ALL after the UNION keyword.

Example: UNION ALL clause: The following SQL statement gives a combined result table containing employee numbers in ascending order, and includes duplicate numbers:

```
SELECT EMPNO
  FROM DSN8810.EMP
 WHERE WORKDEPT = 'D11'
UNION ALL
SELECT EMPNO
  FROM DSN8810.EMPPROJACT
 WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
 ORDER BY EMPNO;
```

Creating common table expressions: WITH

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 for the SELECT, INSERT, and CREATE VIEW statements.

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.

A common table expression can be used 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 desired 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

Using WITH instead of CREATE VIEW

Using the WITH clause to create a common table expression saves you the overhead of needing to create and drop a regular view that you only need to use once. Also, during statement preparation, DB2 does not need to access the catalog for the view, which saves you additional overhead.

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 maximum total pay based on the total pay for each department.

```
WITH DTOTAL (deptno, totalpay) AS
    (SELECT deptno, sum(salary+bonus)
     FROM DSN8810.EMP
     GROUP BY deptno)
SELECT deptno
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:

```
DEPTNO
=====
D11
```

Using common table expressions with CREATE VIEW

You can use common table expressions before a fullselect in a CREATE VIEW statement. The common table expression must be placed immediately inside of the statement. This 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 (deptno) AS
    WITH DTOTAL (deptno, totalpay) AS
        (SELECT deptno, sum(salary+bonus)
```



```

FROM DSN8810.EMP
GROUP BY deptno)
SELECT deptno
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:

```

DEPTNO
=====
A00
D11
D21

```

Using common table expressions when you use INSERT

You can use common table expressions before a fullselect in an INSERT statement. The common table expression must be placed immediately inside of the statement.

Example: Using a WITH clause in an INSERT statement: The following example illustrates the use of a common table expression in an INSERT statement.

```

INSERT INTO vital_mgr (mgrno)
WITH VITALDEPT (deptno, se_count) AS
  (SELECT deptno, count(*)
   FROM DSN8810.EMP
   WHERE job = 'senior engineer'
   GROUP BY deptno)
SELECT d.manager
FROM DSN8810.DEPT d, VITALDEPT s
WHERE d.deptno = s.deptno
AND s.se_count > (SELECT AVG(se_count)
FROM VITALDEPT);

```

The fullselect in the previous example uses the result table for VITALDEPT to find the manager's number for departments that have a greater than average number of senior engineers. The manager's number is then inserted into the vital_mgr table.

Using recursive SQL

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*. Queries that use recursion are useful in applications like bill of materials applications, network planning applications, and reservation systems.

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 it's FROM clause
 - Not include aggregate functions, a GROUP BY clause, or a HAVING clause
- The column names must be specified following the table name of the common table expression

- The data types, lengths, and CCSIDs of the column names from the common table expression that are referenced in the iterative fullselect must match
- The UNION statements must be UNION ALL
- Outer joins must not be part of any recursion cycle
- Subquery must not be part on any recursion cycle

It is possible to introduce an infinite loop when developing a recursive common table expression. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. A warning is issued if one of the following 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 Appendix E, “Recursive common table expression examples,” on page 1097 for examples of bill of materials applications that use recursive common table expressions.

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 as follows:

- Set the contents of a host variable to the value of an expression by using the SET host-variable assignment statement.
EXEC SQL SET :hvranda1 = RAND(:hvranda);
- Use the VALUES INTO statement to return the value of an expression in a host variable.
EXEC SQL VALUES RAND(:hvranda)
INTO :hvranda1;
- Select the expression from the DB2-provided EBCDIC table, named SYSIBM.SYSDUMMY1, which consists of one row.
EXEC SQL SELECT RAND(:hvranda)
INTO :hvranda1
FROM SYSIBM.SYSDUMMY1;

Using 15-digit and 31-digit precision for decimal numbers

DB2 allows 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. *s* is a number between one and nine and represents the minimum scale to be used for division operations.

- The installation option for DECIMAL ARITHMETIC on panel DSNTIP4 is DEC31, D31.s, or 31; 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 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 “Using DYNAMICRULES to specify behavior of dynamic SQL statements” on page 502 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 install option DECIMAL ARITHMETIC is DEC31 or 31. *s* is a number between one and nine and represents the minimum scale to be used for division operations. See “Step 1: Process SQL statements” on page 473 for information about precompiling and for a list of all precompiler options.

Recommendation: Choose DEC31 or D31.s to reduce the chance of overflow, or when dealing with a precision greater than 15 digits. *s* is a number between one and nine and represents the minimum scale to be used for division operations.

Avoiding decimal arithmetic errors: 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. This specification reduces the probability of errors for statements that are embedded in the program. *s* is a number between one and nine and represents the minimum scale to be used for division operations.

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.

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.

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.

Finding information in the DB2 catalog

The following examples show you how to access the DB2 system catalog tables to list the following objects:

- The tables that you can access
- The column names of a table

The contents of the DB2 system catalog tables can be a useful reference tool when you begin to develop an SQL statement or an application program.

Displaying a list of tables you can use

The catalog table, SYSIBM.SYSTABAUTH, lists table privileges 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.

```
SELECT DISTINCT TCREATOR, TTNAME
  FROM SYSIBM.SYSTABAUTH
 WHERE GRANTEE IN (USER, 'PUBLIC', 'PUBLIC*') AND GRANTEETYPE = ' ';
```

In this query, the predicate GRANTEETYPE = ' ' selects authorization IDs.

If your DB2 subsystem uses an exit routine for access control authorization, you cannot rely on catalog queries to tell you the tables you can access. When such an exit routine is installed, both RACF and DB2 control table access.

Displaying a list of columns in a table

Another catalog table, SYSIBM.SYSCOLUMNS, describes every column of every table. Suppose you run the previous SQL statements to display a list of tables you can access and you now want to display information about table DSN8810.DEPT. To execute the following example, you must have the SELECT privilege on SYSIBM.SYSCOLUMNS.

```
SELECT NAME, COLTYPE, SCALE, LENGTH
  FROM SYSIBM.SYSCOLUMNS
 WHERE TBNAME = 'DEPT'
    AND TBCREATOR = 'DSN8810';
```

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 those column occupy in the base table, rather than the length of the LOB or ROWID data. To determine the maximum length of data for a LOB or ROWID column, include the LENGTH2 column in your query, as in the following example:

```
SELECT NAME, COLTYPE, LENGTH, LENGTH2
  FROM SYSIBM.SYSCOLUMNS
 WHERE TBNAME = 'EMP_PHOTO_RESUME'
    AND TBCREATOR = 'DSN8810';
```

Chapter 2. Working with tables and modifying data

This chapter discusses these topics:

- Creating your own tables: CREATE TABLE
- “Working with temporary tables” on page 21
- “Dropping tables: DROP TABLE” on page 25
- “Defining a view: CREATE VIEW” on page 25
- “Changing data through a view” on page 26
- “Dropping views: DROP VIEW” on page 27
- “Inserting rows: INSERT” on page 27
- “Selecting values as you insert: SELECT FROM INSERT” on page 31
- “Updating current values: UPDATE” on page 36
- “Deleting rows: DELETE” on page 37

See *DB2 SQL Reference* for more information about working with tables and data.

Working with tables

This section discusses how to work with tables. As you work with tables, you might need to create new tables, copy existing tables, add columns, add or drop referential and check constraints, drop the tables you are working with, or make any number of changes.

Creating your own tables: CREATE TABLE

Use the CREATE TABLE statement to create a table. 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);
```

The preceding CREATE statement has the following elements:

- CREATE TABLE, which names the table PRODUCT.
- 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)). For more information about data types, see “Data types” on page 4.
 - Optionally, a default value. See “Identifying defaults.”
 - Optionally, a referential constraint or check constraint. See “Using referential constraints” on page 261 and “Using check constraints” on page 259.

You must separate each column description from the next with a comma, and enclose the entire list of column descriptions in parentheses.

Identifying defaults

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, zero is the default value.
 - For fixed-length strings, blank is the default value.
 - For variable-length strings, including LOB strings, the empty string (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
 - USER, which specifies the value of the USER special register at the time that an INSERT statement assigns a default value to the column in the row that is being inserted
 - CURRENT SQLID, which specifies the value of the CURRENT SQLID special register at the time that an INSERT statement assigns a default value to the column in the row that is being inserted
 - The name of a cast function that casts a default value (of a built-in data type) to the distinct type of a column

Creating work tables

Before testing SQL statements that insert, update, and delete rows, you should create *work tables* (duplicates of the DSN8810.EMP and DSN8810.DEPT tables), so that the original sample tables remain intact. This section shows how to create two work tables and how to fill a work table with the contents of another table.

Each example shown in this chapter assumes that you logged on using your own authorization ID. The authorization ID qualifies the name of each object 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 DSN8810.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.

Creating a new department table

Use the following statements to create a new department table called YDEPT, modeled after the existing table, DSN8810.DEPT, and an index for YDEPT:

```
CREATE TABLE YDEPT
  LIKE DSN8810.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 DSN8810.DEPT to your own YDEPT work table.

```
INSERT INTO YDEPT
  SELECT *
  FROM DSN8810.DEPT;
```

For information about using the INSERT statement, see “Inserting rows: INSERT” on page 27.

Creating a new employee table

You can use the following statements to create a new employee table called YEMP.

```
CREATE TABLE YEMP
  (EMPNO      CHAR(6)          PRIMARY KEY NOT NULL,
   FIRSTNAME  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 statement ALTER TABLE. If you want to change a table name after you create it, use the statement RENAME TABLE.

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 drop a column from a table.

For more information about changing a table definition by using ALTER TABLE, see Part 2 (Volume 1) of *DB2 Administration Guide*. For other details about the ALTER TABLE and RENAME TABLE statements, see Chapter 5 of *DB2 SQL Reference*.

Working with temporary tables

When you need a table only for the duration of an application process, you can create a temporary table. There are two kinds of temporary tables:

- 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).
- DB2 does no locking (for created temporary tables) or limited locking (for declared temporary tables).

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. For more information, see “Writing a stored procedure to return result sets to a DRDA client” on page 650. The following sections provide more details on created temporary tables and declared temporary tables.

Working with created temporary tables

You create the *definition* of a created temporary table using the SQL statement CREATE GLOBAL TEMPORARY TABLE.

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 using the LIKE clause:

```
CREATE GLOBAL TEMPORARY TABLE TEMPPROD LIKE PROD;
```

The SQL statements in the previous examples create identical definitions, even though table PROD contains two columns, DESCRIPTION and CURDATE, that are defined as NOT NULL WITH DEFAULT. Unlike the PROD sample table, the DESCRIPTION and CURDATE columns in the TEMPPROD table are defined as NOT NULL and do not have defaults, because created temporary tables do not support non-null default values.

After you run one of the two CREATE statements, the definition of TEMPPROD exists, but no instances of the table exist. To drop the definition of TEMPPROD, you must run the following statement:

```
DROP TABLE TEMPPROD;
```

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

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 HOLD 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 is run, 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 defined WITH HOLD, is open when the COMMIT statement is run. In either case, DB2 drops the instance of TEMPPROD when the application ends.

Working with declared temporary tables

You create an instance of a declared temporary table using the SQL statement DECLARE GLOBAL TEMPORARY TABLE. 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 create a special database and table spaces for them. You do that by running the CREATE DATABASE statement with the AS TEMP clause, and then creating segmented table spaces in that database. A DB2 subsystem can have only one database for declared temporary tables, but that database can contain more than one table space. There must be at least one table space with a 8-KB page size in the TEMP database to declare a temporary table.

Example: The following statements create a database and table space for declared temporary tables:

```
CREATE DATABASE DTTDB AS TEMP;
CREATE TABLESPACE DTTTS IN DTTDB
SEGSIZE 4;
```

You can define a declared temporary table in any of the following ways:

- 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 or created temporary table that you copy has identity columns, you can specify that the corresponding columns in the declared temporary table are also identity columns. Do that by specifying the INCLUDING IDENTITY COLUMN ATTRIBUTES clause when you define the declared temporary table.

- Use a fullselect to choose specific columns from 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. Do that by specifying the INCLUDING IDENTITY COLUMN ATTRIBUTES clause when you define the declared temporary table.

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 underlying the view.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
AS (SELECT * FROM PROVIEW)
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 populate the declared temporary table using INSERT statements, modify the table using searched or positioned UPDATE or DELETE statements, and query the table using SELECT statements. You can also 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 there is a held cursor 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.

Dropping tables: DROP 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 associated with that table. You also lose all authorities granted on the table.

For more information about the DROP statement, see Chapter 5 of *DB2 SQL Reference*.

Working with views

This section discusses how to use CREATE VIEW and DROP VIEW to control your views of existing 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.

Defining a view: CREATE VIEW

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 and, in most cases, is interchangeable with a table. Using views can 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 created with the following statement shows each department manager's name with the department data in the DSN8810.DEPT table.

```
CREATE VIEW VDEPTM AS
  SELECT DEPTNO, MGRNO, LASTNAME, ADMRDEPT
    FROM DSN8810.DEPT, DSN8810.EMP
   WHERE DSN8810.EMP.EMPNO = DSN8810.DEPT.MGRNO;
```

When a program accesses the data defined by a view, DB2 uses the view definition to return a set of rows the program can access with SQL statements. To see the departments 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 USER and CURRENT SQLID special registers in the CREATE VIEW statement. When referencing the view, DB2 uses the value of the USER or CURRENT SQLID 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. 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 UNION or UNION ALL operators, 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.

Changing data through a view

Some views are read-only; other views are subject to update or insert restrictions. (See Chapter 5 of *DB2 SQL Reference* for more information about read-only views.)

If a view does not have update restrictions, some additional considerations include:

- 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 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 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 specifies WITH CHECK.

Dropping views: DROP VIEW

When you drop a view, you also drop all views that are defined on the following view. This SQL statement drops the VDEPTM view:

```
DROP VIEW VDEPTM;
```

Modifying DB2 data

This section discusses how to add or modify data in an existing table using the statements INSERT, UPDATE, and DELETE:

- “Inserting rows: INSERT”
- “Selecting values as you insert: SELECT FROM INSERT” on page 31
- “Updating current values: UPDATE” on page 36
- “Deleting rows: DELETE” on page 37

Inserting rows: INSERT

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. “Inserting a single row” on page 28 explains how to use the VALUES clause of the INSERT statement to add a single row of column values to a table.
- In an application program, specify arrays of column values to insert multiple rows into a table. “Inserting multiple rows of data from host variable arrays” on page 87 explains how to 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. “Inserting rows into a table from another table” on page 29 explains how to use the SELECT statement within an INSERT statement to add multiple rows to a table.

In each case, for every row 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, you can specify DEFAULT to tell DB2 to insert the default value for that column:

- Is nullable.
- Is defined with a default value.
- Has data type ROWID. ROWID columns always have default values.
- Is an identity column. Identity columns always have default values.

The values that you can insert into a ROWID column or an identity column depend on whether the column is defined with GENERATED ALWAYS or GENERATED BY DEFAULT. See “Inserting data into a ROWID column” on page 30 and “Inserting data into an identity column” on page 30 for more information.

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 because of 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 giving 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 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 YEMP has a foreign key, WORKDEPT, referencing the primary key, DEPTNO, in YDEPT, the value inserted for WORKDEPT (E31) must be a value of DEPTNO in YDEPT or null.

```
INSERT INTO YEMP
VALUES ('000400', 'RUTHERFORD', 'B', 'HAYES', 'E31', '5678', '1983-01-01',
'MANAGER', 16, 'M', '1943-07-10', 24000, 500, 1900);
```

Example: The following statement also inserts a row into the YEMP table. Because the unspecified columns allow nulls, DB2 inserts null values into the columns that you do not specify. Because YEMP has a foreign key, WORKDEPT, referencing the primary key, DEPTNO, in YDEPT, the value inserted for WORKDEPT (D11) must be a value of DEPTNO in YDEPT or null.

```
INSERT INTO YEMP
(EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, JOB)
VALUES ('000410', 'MILLARD', 'K', 'FILLMORE', 'D11', '4888', 'MANAGER');
```

Inserting rows into a table from another table

You can copy data 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 DSN8810.EMP into the newly created table:

```
INSERT INTO TELE
SELECT LASTNAME, FIRSTNME, PHONENO
FROM DSN8810.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 selected from the DSN8810.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 as well as phone number. The fullselect within the INSERT statement fills the DLIST table with data from rows selected from two existing tables, DSN8810.DEPT and DSN8810.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 DSN8810.DEPT, DSN8810.EMP
  WHERE DEPTNO = WORKDEPT;
```

Other ways to insert data

Besides using stand-alone INSERT statements, you can use the following two ways to insert data into a table:

- You can write an application program to prompt for and enter large amounts of data into a table. For details, see Part 2, “Coding SQL in your host application program,” on page 73.
- You can also use the DB2 LOAD utility to enter data from other sources. See Part 2 of *DB2 Utility Guide and Reference* for more information about the LOAD utility.

Inserting data into a ROWID column

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 “Using ROWID columns as keys” on page 269.

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;
```

Inserting data into an identity column

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 “Using identity columns as keys” on page 270

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;
```

Selecting values as you insert: SELECT FROM INSERT

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 section 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),
   HIREDATE 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, HIREDATE, 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. For an example of this method, see “Parent keys and foreign keys” on page 272.

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 columns that are based on expressions.
- 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. For information about triggers, see Chapter 12, “Using triggers for active data,” on page 277.

Example: Suppose 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 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. For information about using host variables and SELECT ... INTO, see “Using host variables ” on page 80.

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. 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, 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. For information about using cursors, see Chapter 7, “Using a cursor to retrieve a set of rows,” on page 103.

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 DSN8810.EMP_PHOTO_RESUME (EMPNO)
SELECT EMPNO FROM DSN8810.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 DSN8810.EMP_PHOTO_RESUME (EMPNO)
SELECT EMPNO FROM DSN8810.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 DSN8810.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. For information about the multiple-row INSERT statement, see “Inserting multiple rows of data from host variable arrays” on page 87.

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);
```

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. For information about cursor sensitivity, see “Using a scrollable cursor” on page 114.

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 EMPSAMP (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 EMPSAMP SET SALARY = SALARY + 500;
while (SQLCODE == 0) {
  EXEC SQL FETCH CS1 INTO :hv_salary;
  /* print fetch result */
  ...
}
```

The fetches that occur after the update processing return the rows that were generated during OPEN cursor processing. However, 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. For information about held cursors, see “Held and non-held cursors” on page 122.

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 DSN8810.EMP_PHOTO_RESUME (EMPNO)
                        SELECT EMPNO FROM DSN8810.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: When you set a savepoint prior to opening the cursor and then roll back to that savepoint, all of the insertions are undone. For information about savepoints and ROLLBACK processing, see “Using savepoints to undo selected changes within a unit of work” on page 439.

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 DSN8810.EMP_PHOTO_RESUME (EMPNO)
                        SELECT EMPNO FROM DSN8810.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 DSN8810.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 EMP_SAMP (NAME, SALARY)
SELECT FIRSTNAME || MIDINIT || LASTNAME,
SALARY + 10000.00
FROM DSN8810.EMP)
```

The addition of 10000.00 causes a decimal overflow to occur, and no rows are inserted into the EMP_SAMP 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 EMP_SAMP (NAME, SALARY)
SELECT FIRSTNAME || MIDINIT || LASTNAME, SALARY + 10000.00
FROM DSN8810.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;
```

Updating current values: UPDATE

To change the data in a table, use the UPDATE statement. You can also use the UPDATE statement to remove a value from a row's column (without removing the row) by changing the column's value to null.

Example: Suppose an employee relocates. To update several items of the employee's data in the YEMP work table to reflect the move, you can execute:

```
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 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
An expression can be any of the following items:

- A column
- A constant
- A fullselect that returns a scalar
- A host variable
- A special register

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 you want to update.

If you omit the WHERE clause, DB2 updates **every row** in the table or view with the values 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 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 and bonus for employee 000190 to the average salary and minimum bonus for all employees.

```
UPDATE YEMP
  SET (SALARY, BONUS) =
    (SELECT AVG(SALARY), MIN(BONUS)
     FROM EMP)
  WHERE EMPNO = '000190';
```

Deleting rows: DELETE

You can use the DELETE statement to remove entire rows from a table. The DELETE statement removes zero or more rows of a table, depending on how many rows satisfy the search condition you specify in the WHERE clause. If you omit a WHERE clause from a DELETE statement, DB2 removes **all the rows** from the table or view you have named. The DELETE statement does not remove specific columns from the row.

You can use DELETE to remove all rows from a created temporary table or declared temporary table. However, you can use DELETE with a WHERE clause to remove only selected rows from a declared temporary table.

This 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 host 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).

Deleting every row in a table

The DELETE statement is a powerful statement that deletes **all** rows of a table unless you specify a WHERE clause to limit it. (With segmented table spaces, deleting all rows of a table is very fast.) For example, the following 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. By comparison, using DROP TABLE drops all views and authorizations, which can invalidate plans and packages. For information about the DROP statement, see “Dropping tables: DROP TABLE” on page 25.

Chapter 3. 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.

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.

The examples in this section use the following two tables to show various types of joins:

The PARTS table			The PRODUCTS table		
PART	PROD#	SUPPLIER	PROD#	PRODUCT	PRICE
=====	=====	=====	=====	=====	=====
WIRE	10	ACWF	505	SCREWDRIIVER	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			

Figure 2 illustrates how these two tables can be combined using the three outer join functions.

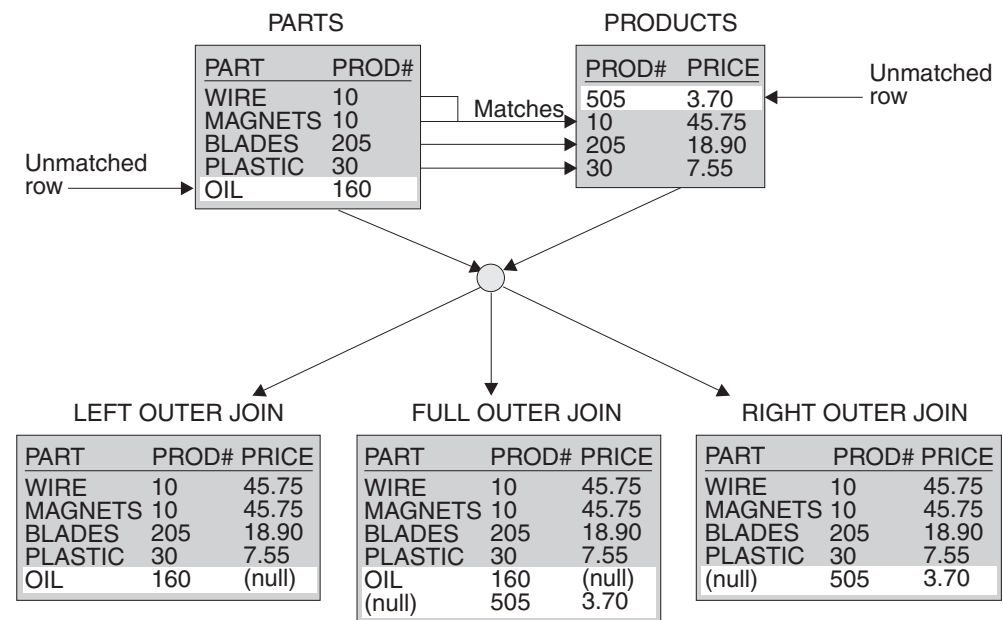


Figure 2. Three outer joins from the PARTS and PRODUCTS tables

The result table contains data 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. But, 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.

Inner join

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. See Chapter 4 of *DB2 SQL Reference* for the complete syntax of a join condition.

In the simplest type of inner join, the join condition is *column1=column2*.

Example: You can join the PARTS and PRODUCTS tables 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

Notice 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.
An *outer join*, however, includes rows where 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 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;
```

In either case, 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: In the following example, **A** indicates the first instance of table DSN8810.PROJ and **B** indicates the second instance of this table. The join condition is such that the value in column PROJNO in table DSN8810.PROJ A must be equal to a value in column MAJPROJ in table DSN8810.PROJ B.

The following SQL statement joins table DSN8810.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 DSN8810.PROJ A, DSN8810.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.

Full outer join

The clause 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.

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:

```
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 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 41 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

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: 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

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: 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 containing join operations

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 containing 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

Using more than one join in an SQL statement

Using more than one join: You can join more than two tables. Suppose 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 DSN8810.EMP, DSN8810.PROJ, DSN8810.DEPT
    WHERE EMPNO = RESPEMP
      AND WORKDEPT = DSN8810.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.

Using more than one join type: You can use more than one join type in the FROM clause. Suppose that you want a result table that shows employees whose last name begins with 'S' or a letter after 'S', their department names, and the projects that they are responsible for, if any. You can use the following SELECT statement:

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
  FROM DSN8810.EMP INNER JOIN DSN8810.DEPT
    ON WORKDEPT = DSN8810.DEPT.DEPTNO
  LEFT OUTER JOIN DSN8810.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'.
2. Join the intermediate result table with the project table on the employee number, keeping the rows with no matching employee number in the project table.
3. Process the select list in the final result table, leaving only four columns.

Using 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 use:

- A nested table expression, which is a fullselect enclosed in parentheses and followed by a correlation name
- A user-defined table function, which is a user-defined function that returns a table

Example of using a nested table expression as the right operand of a join: The following query contains a fullselect as the right operand of a left outer join with the PROJECTS table. The correlation name is TEMP.

```
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;
```

The following statement is the nested table expression:

```
(SELECT PART,
  COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
  PRODUCTS.PRODUCT
 FROM PARTS FULL OUTER JOIN PRODUCTS
  ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
```

Example of using correlated references: In the following example, the correlation name that is used for the nested table expression is CHEAP_PARTS. The 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	SCREWDRIER
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 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.

```
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 '30' is 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: You can join the results of a user-defined table function with a table, just as you can join two tables. For example, suppose CVTPRICE is a table function that converts the prices in the PRODUCTS table to the currency 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#;
```

Using correlated references in table specifications in joins

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 preceded by the keyword TABLE
- The argument of a table function

For more information about correlated references, see “Using correlation names in references” on page 54.

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)), then 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.

Chapter 4. Using 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.

This chapter presents the following sections:

- “Conceptual overview”
- “How to code a subquery” on page 51
- “Using correlated subqueries” on page 53

Conceptual overview

Suppose that you want a list of the employee numbers, names, and commissions of all employees working 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 DSN8810.EMP
WHERE EMPNO

:
```

But you cannot proceed because the DSN8810.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 DSN8810.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 surrounding the subquery is called the *outer SELECT*.

```
SELECT EMPNO, LASTNAME, COMM
FROM DSN8810.EMP
WHERE EMPNO IN
  (SELECT EMPNO
   FROM DSN8810.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 DSN8810.EMPPROJECT
 WHERE PROJNO = 'MA2111');
```

The result is in an *interim result table*, similar to the one shown 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 DSN8810.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. The subquery executes only once, if the subquery is the same for every row or group.

This kind of subquery is *uncorrelated*. In the previous query, for example, the content of the subquery is the same for every row of the table DSN8810.EMP.

Subqueries that vary in content from row to row or group to group are *correlated* subqueries. For information about correlated subqueries, see “Using correlated subqueries” on page 53. All of the following information that precedes the section about correlated subqueries applies to both correlated and uncorrelated subqueries.

Subqueries and predicates

A subquery is always part of a predicate. The predicate is of the form:
operand operator (subquery)

The predicate can be part of a WHERE or HAVING clause. A WHERE or HAVING clause can include predicates that contain subqueries. A predicate containing 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 DSN8810.EMP
WHERE SALARY =
(SELECT AVG(SALARY)
FROM DSN8810.EMP);
```

```
SELECT EMPNO, LASTNAME
FROM DSN8810.EMP
WHERE (SALARY, BONUS) IN
      (SELECT AVG(SALARY), AVG(BONUS)
       FROM DSN8810.EMP);
```

Except for a subquery of a basic predicate, the result table can contain more than one row. For more information, see “Basic predicate .”

Tables in subqueries of UPDATE, DELETE, and INSERT statements

The following rules apply to a table that is used in a subquery for an UPDATE, DELETE, or INSERT statement:

- When you use a subquery in an INSERT statement, the subquery can use the same table as the INSERT statement.
- When you use a subquery in a *searched* UPDATE or DELETE statement (an UPDATE or DELETE that does not use a cursor), the subquery can use the same table as the UPDATE or DELETE statement.
- When you use a subquery in a *positioned* UPDATE or DELETE statement (an UPDATE or DELETE that uses a cursor), the subquery cannot use the same table as the UPDATE or DELETE statement.

How to code a subquery

You can specify a subquery in either a WHERE or HAVING clause by using:

- A basic predicate
- A quantified predicate: ALL, ANY, or SOME
- The IN keyword
- The EXISTS keyword

Basic predicate

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.

Example: 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 DSN8810.EMP
WHERE EDLEVEL >
      (SELECT AVG(EDLEVEL)
       FROM DSN8810.EMP);
```

Quantified predicate : 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.

If a subquery that returns one or more null values gives you unexpected results, see the description of quantified predicates in Chapter 2 of *DB2 SQL Reference*.

Using the 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 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.

Using the 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 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.

IN keyword

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.

Example: The following query returns the names of department managers:

```
SELECT EMPNO, LASTNAME
FROM DSN8810.EMP
WHERE EMPNO IN
  (SELECT DISTINCT MGRNO
   FROM DSN8810.DEPT);
```

EXISTS keyword

In the subqueries presented thus far, DB2 evaluates the subquery and uses the result as part of the WHERE clause of the outer SELECT. In contrast, when you use the keyword EXISTS, DB2 simply 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.

Example: 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 DSN8810.EMP
WHERE EXISTS
  (SELECT *
   FROM DSN8810.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 you specify does not exist; that is, you can code the following clause: WHERE NOT EXISTS (SELECT ...);

Using correlated subqueries

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. You can also write a subquery that DB2 re-evaluates when it examines a new row (in a WHERE clause) or group of rows (in a HAVING clause) as it executes the outer SELECT. This is called a *correlated subquery*.

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 DSN8810.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. A query that does this follows:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM DSN8810.EMP X
WHERE EDLEVEL >
      (SELECT AVG(EDLEVEL)
       FROM DSN8810.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 DSN8810.EMP. At any time during the execution of the query, X designates the row of DSN8810.EMP to which the WHERE clause is being applied.

Consider what happens when the subquery executes for a given row of DSN8810.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 DSN8810.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 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

Using correlation names in references

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 "Using nested table expressions and user-defined table functions in joins" on page 46. 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 DSN8810.EMP AS X
WHERE EDLEVEL >
      (SELECT AVG(EDLEVEL)
       FROM DSN8810.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

When you use a correlated subquery in an UPDATE statement, the correlation name refers to the rows you are updating. For example, when all activities of a project must complete before September 2004, your department considers that project to be a priority project. You can use the following SQL statement to evaluate the projects in the DSN8810.PROJ table, and write a 1 (a flag to indicate PRIORITY) in the PRIORITY column (a column you have added to DSN8810.PROJ for this purpose) for each priority project:

```
UPDATE DSN8810.PROJ X
SET PRIORITY = 1
WHERE DATE('2004-09-01') >
      (SELECT MAX(ACENDATE)
       FROM DSN8810.PROJACT
       WHERE PROJNO = X.PROJNO);
```

As DB2 examines each row in the DSN8810.PROJ table, it determines the maximum activity end date (the ACENDATE column) for all activities of the project (from the DSN8810.PROJACT table). If the end date of each activity associated with the project is before September 2004, the current row in the DSN8810.PROJ table qualifies and DB2 updates it.

Using correlated subqueries in a DELETE statement

When you use a correlated subquery in a DELETE statement, the correlation name represents the row you delete. DB2 evaluates the correlated subquery once for each row in the table that is named in the DELETE statement to decide whether or not 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 half a person's time or less. The department then deletes the rows for that project from the DSN8810.PROJ table. In the examples in this section, PROJ and PROJACT are independent tables; that is, they are separate tables with no referential constraints defined on them.

```
DELETE FROM DSN8810.PROJ X
  WHERE .5 >
    (SELECT SUM(ACSTAFF)
     FROM DSN8810.PROJACT
     WHERE PROJNO = X.PROJNO);
```

To process this statement, DB2 determines for each project (represented by a row in the DSN8810.PROJ table) whether or not the combined staffing for that project is less than 0.5. If it is, DB2 deletes that row from the DSN8810.PROJ table.

To continue this example, suppose DB2 deletes a row in the DSN8810.PROJ table. You must also delete rows related to the deleted project in the DSN8810.PROJACT table. To do this, use:

```
DELETE FROM DSN8810.PROJACT X
  WHERE NOT EXISTS
    (SELECT *
     FROM DSN8810.PROJ
     WHERE PROJNO = X.PROJNO);
```

DB2 determines, for each row in the DSN8810.PROJACT table, whether a row with the same project number exists in the DSN8810.PROJ table. If not, DB2 deletes the row in DSN8810.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, the last delete rule in the path to that table must be RESTRICT or NO ACTION if 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 DSN8810.DEPT THIS
  WHERE NOT DEPTNO =
    (SELECT WORKDEPT
     FROM DSN8810.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. The deletion involves the table that is
referred to in the subquery (DSN8810.EMP is a dependent table of DSN8810.DEPT)
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. See
“Materialization” on page 837 for more information about materialization.

Chapter 5. Using SPUFI to execute SQL from your workstation

This chapter explains how to enter and execute SQL statements at a TSO workstation by using the SPUFI (SQL processor using file input) facility. This chapter contains the following sections:

- “Allocating an input data set and using SPUFI”
- “Changing SPUFI defaults” on page 62
- “Entering SQL statements ” on page 66
- “Processing SQL statements ” on page 68
- “When SQL statements exceed resource limit thresholds” on page 68
- “Browsing the output ” on page 69

You can execute most of the interactive SQL examples shown in Part 1, “Using SQL queries,” on page 1 by following the instructions provided in this chapter and using the sample tables shown in Appendix A, “DB2 sample tables,” on page 993. The instructions assume that ISPF is available to you.

You can use the TSO PROFILE command to control whether message IDs are
displayed. To view message IDs, type TSO PROFILE MSGID on the ISPF command
line. To suppress message IDs, type TSO PROFILE NOMSGID.

Allocating an input data set and using SPUFI

Before you use SPUFI, you should allocate an input data set, if one does not already exist. This data set will contain one or more SQL statements that you want to execute. For information on ISPF and allocating data sets, see *z/OS ISPF User's Guide Volumes 1 and 2*.

To use SPUFI, select SPUFI from the DB2I Primary Option Menu as shown in Figure 152 on page 519

The SPUFI panel then displays as shown in Figure 3 on page 60.

From then on, when the SPUFI panel displays, 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 displays, as needed. Values that you do not change remain in effect.

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 3. The SPUFI panel filled in

Fill out the SPUFI panel. You can access descriptions for each of the fields in the panel in the DB2I help system. See “DB2I help” on page 518 for more information about the DB2I help system. The following descriptions explain the information that you need to provide 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

Figure 3 on page 60 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

Allows 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 62 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 described in “Entering SQL statements ” on page 66.)

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. For a list of those SQL statements, see Appendix H, “Characteristics of SQL statements in DB2 UDB for z/OS,” on page 1113.

8 AUTOCOMMIT

To make changes to the DB2 data permanent, leave **Y(YES)** on line 8. Specifying **Y** makes SPUFI issue **COMMIT** if all statements execute successfully. If all statements do not execute successfully, SPUFI issues a **ROLLBACK** statement, which deletes changes already made to the file (back to the last commit point). For information about the **COMMIT** and **ROLLBACK** functions, see “Unit of work in TSO batch and online” on page 432 or Chapter 5 of *DB2 SQL Reference*.

If you specify **N**, DB2 displays the SPUFI **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. SPUFI 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. For more information, see “Format of **SELECT** statement results ” on page 70.

10 CONNECT LOCATION

Specify the name of the database server, if applicable, to which you want to submit SQL statements. SPUFI then issues a type 2 **CONNECT** statement to this server.

SPUFI 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.

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.

Changing SPUFI defaults

When you finish with the SPUFI panel, press the ENTER key. If you specified YES on line 5 of the SPUFI panel, the next panel you see is the SPUFI Defaults panel. SPUFI provides default values the first time you use SPUFI, 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. Figure 4 shows the initial default values.

DSNE345I 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)
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)
#Output data set characteristics:
6 RECORD LENGTH ... ==> 4092 (LRECL= logical record length)
7 BLOCKSIZE ==> 4096 (Size of one block)
8 RECORD FORMAT.... ==> VB (RECFM= F, FB, FBA, V, VB, or VB)
9 DEVICE TYPE..... ==> SYSDA (Must be a DASD unit name)

#Output format characteristics:
#10 MAX NUMERIC FIELD ==> 33 (Maximum width for numeric field)
#11 MAX CHAR FIELD .. ==> 80 (Maximum width for character field)
#12 COLUMN HEADING .. ==> NAMES (NAMES, LABELS, ANY, or BOTH)
#13 FOR BIT DATA ==> ASIS (ASIS or HEX)

#PRESS: ENTER to process END to exit HELP for more information

Figure 4. The SPUFI defaults panel

| If you want to change the current default values, specify new values in the fields
| of the panel. All fields must contain a value. The DB2I help system contains
| detailed descriptions of each of the fields of the CURRENT SPUFI DEFAULTS
| panel. The following descriptions explain the information you need to provide on
| the CURRENT SPUFI DEFAULTS panel.

1 SQL TERMINATOR

Allows you to specify the character that you use to end each SQL statement. You can specify any character *except* the characters listed in Table 3. A semicolon (;) is the default SQL terminator.

Table 3. Invalid special characters for the SQL terminator

Name	Character	Hexadecimal representation
blank		X'40'
comma	,	X'5E'

Table 3. Invalid special characters for the SQL terminator (continued)

Name	Character	Hexadecimal representation
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 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 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. See “Entering SQL statements ” on page 66 for details.

2 ISOLATION LEVEL

Allows you to specify the isolation level for your SQL statements. See “The ISOLATION option” on page 412 for more information.

3 MAX SELECT LINES

The maximum number of output lines 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 NO (the default) or YES to indicate whether SPUFI will continue to process an SQL statement after receiving SQL warnings:

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.

YES If a warning occurs when SPUFI executes an OPEN or FETCH for
a SELECT statement, SPUFI continues to process the SELECT
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 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 4092-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.

7 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.

8 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).

9 DEVICE TYPE

Allows you to 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.

10 MAX NUMERIC FIELD

The maximum width of a numeric value column in your output. Choose a value greater than 0. The default is 33. For more information, see "Format of SELECT statement results " on page 70.

11 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 80. For more information, see "Format of SELECT statement results " on page 70.

12 COLUMN HEADING

You can specify NAMES, LABELS, ANY, or BOTH for column headings.

- NAMES (default) uses column names only.
- LABELS 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, SPUFI

displays the contents of the AS clause in the heading, rather than the column name. You define column labels with LABEL statements.

13 FOR BIT DATA

Specify how SPUFI is to display the data from FOR BIT DATA columns.

ASIS SPUFI displays the data from FOR BIT DATA columns as it is stored. The default is ASIS.

If you specify ASIS when a graphic character set is in effect, SPUFI replaces all occurrences of the shift-out (X'0E') and shift-in (X'0F') characters in the output of the FOR BIT DATA columns with a substitution character of '.' (X'4B'). To avoid this substitution, specify HEX.

HEX SPUFI displays the data from FOR BIT DATA columns in hexadecimal format.

When you have entered your SPUFI options, press the ENTER key to continue. SPUFI then processes the next processing option for which you specified YES. If all other processing options are NO, SPUFI displays the SPUFI panel.

If you press the END key, you return to the SPUFI panel, but you lose all the changes you made on the SPUFI Defaults panel. If you press ENTER, SPUFI saves your changes.

Changing SPUFI defaults - panel 2

If you specify YES on line 5 of the SPUFI Defaults panel, the next panel that you see is the second SPUFI Defaults panel. Figure 5 shows the initial default values.

```
DSNESP07          CURRENT SPUFI DEFAULTS - PANEL 2          SSID: DSN
====>
DO NOT CHANGE THE FIELDS BELOW UNLESS DIRECTED BY THE DB2 SYSTEM ADMINISTRATOR
Change plans to be used by your SPUFI session:
 1 CS ISOLATION PLAN  ==> DSNESPC (Name of plan for CS isolation level)
 2 RR ISOLATION PLAN  ==> DSNESPC (Name of plan for RR isolation level)

Indicate warning message status:
 3 BLANK CCSID WARNING ==> YES      (Show warning if terminal CCSID is blank)

PRESS: ENTER to process    END to exit    HELP for more information
```

Figure 5. CURRENT SPUFI DEFAULTS - PANEL 2

Specify values for the following options on the CURRENT SPUFI DEFAULTS - PANEL 2 panel. All fields must contain a value. Using an invalid or incorrect plan name might cause SPUFI to experience operational errors or it might cause data contamination.


```

EDIT -----userid.EXAMPLES(XMP1) ----- COLUMNS 001 072
COMMAND INPUT ==> SAVE                                SCROLL ==> PAGE
***** TOP OF DATA *****
000100 SELECT LASTNAME, FIRSTNME, PHONENO
000200 FROM DSN8810.EMP
000300 WHERE WORKDEPT= 'D11'
000400 ORDER BY LASTNAME;
***** BOTTOM OF DATA *****

```

Figure 6. The edit panel: After entering an SQL statement

Pressing the END PF key saves the data set. You can save the data set *and* continue editing it by entering the SAVE command. Saving the data set after every 10 minutes or so of editing is recommended.

Figure 6 shows what the panel looks like if you enter the sample SQL statement, followed by a SAVE command.

You can bypass the editing step by resetting the EDIT INPUT processing option:

```
EDIT INPUT ... ==> NO
```


#

Retrieving Unicode UTF-16 graphic data

7SPUFI can be used to 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.

Entering comments

#

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. DB2 ignores everything to the right of the two hyphens.

Setting the SQL terminator character

Use the text --#SET TERMINATOR *character* in a SPUFI input data set as an instruction to SPUFI to interpret *character* as a statement terminator. A semicolon (;) is the default SQL terminator. You can specify any single-byte character *except* one of the characters that are listed in . The terminator that you specify overrides a terminator that you specified in option 1 of the CURRENT SPUFI DEFAULTS panel or in a previous --#SET TERMINATOR statement.

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 this:

```

CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#

```

Be careful to choose a character for the SQL terminator that is not used within the statement.

Controlling toleration of warnings

When you use SPUFI, you can control the toleration of warnings with the control statement TOLWARN:

--#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.

Example: 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
```

Processing SQL statements

SPUFI passes the input data set to DB2 for processing. DB2 executes the SQL statement in the input data set EXAMPLES(XMP1), and sends the output to the output data set *userid*.RESULT.

You can bypass the DB2 processing step by resetting the EXECUTE processing option:

```
EXECUTE ..... ==> NO
```

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. To interrupt DB2's processing, press the PA1 key and respond to the prompting message that asks you if you really want to stop processing. This cancels the executing SQL statement and returns you to the ISPF-PDF menu.

What happens to the output data set? This depends 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.

When SQL statements 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 the error time limit for predictive or reactive governing, SPUFI terminates processing of that SQL statement and all statements that follow in the SPUFI input data set. However, SPUFI displays a panel that lets you commit or roll back the previously uncommitted changes that you have made. That panel is shown in Figure 7 on page 69.

```

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 7. 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 a panel that lets you 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 Figure 8.

```

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 8. The resource limit facility warning panel

For information on the DB2 governor and how to set error and warning time limits, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Browsing the output

SPUFI formats and displays the output data set using the ISPF Browse program. Figure 9 on page 70 shows the output from the sample program. An output data set contains these 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 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 specified in the statement.

For all other types of SQL statements 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 DSN8810.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 9. Result data set from the sample problem

Format of SELECT statement results

The results of SELECT statements follow these rules:

- If a column’s numeric or character data cannot display completely:
 - Character values that are too wide truncate on the right.
 - Numeric values that are too wide display as asterisks (*).
 - For columns other than LOB columns, if truncation occurs, the output data set contains a warning message. Because LOB 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 column output.

You can change the amount of data displayed for numeric and character columns by changing values on the CURRENT SPUFI DEFAULTS panel, as described in “Changing SPUFI defaults” on page 62.

- A null value displays as a series of hyphens (-).
- A ROWID or BLOB column value displays in hexadecimal.
- A CLOB column value displays in the same way as a VARCHAR column value.

- A DBCLOB column value displays in the same way as a VARGRAPHIC column value.
- A heading identifies each selected column, and repeats at the top of each output page. The contents of the heading depend on the value you specified in field COLUMN HEADING of the CURRENT SPUFI DEFAULTS panel.

Content of the messages

Each 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 SPUFI made about the location of line numbers (if any) in your input data set.
- Some overall statistics:
 - Number of SQL statements processed
 - Number of input records read (from the input data set)
 - Number of output records 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 statement retrieved
 - Your UPDATE statement modified
 - Your INSERT statement added to a table
 - Your DELETE statement deleted from a table
- Which columns display truncated data because the data was too wide

Part 2. Coding SQL in your host application program

Chapter 6. Basics of coding SQL in an application program	77
Conventions used in examples of coding SQL statements	78
Delimiting an SQL statement	78
Declaring table and view definitions	79
Accessing data using host variables, variable arrays, and structures	79
Using host variables	80
Retrieving a single row of data into host variables	81
Updating data using values in host variables	82
Inserting data from column values that use host variables	83
Using indicator variables with host variables	83
Assignments and comparisons using different data types	85
Changing the coded character set ID of host variables	85
Using host variable arrays	86
Retrieving multiple rows of data into host variable arrays	87
Inserting multiple rows of data from host variable arrays	87
Using indicator variable arrays with host variable arrays	87
Using host structures	90
Retrieving a single row of data into a host structure	90
Using indicator variables with host structures	90
Checking the execution of SQL statements	91
Using the SQL communication area (SQLCA)	91
SQLCODE and SQLSTATE	92
The WHENEVER statement	93
Handling arithmetic or conversion errors	93
The GET DIAGNOSTICS statement	94
Retrieving statement and condition items	94
Data types for GET DIAGNOSTICS items	95
Calling DSNTIAR to display SQLCA fields	98
Defining a message output area	99
Possible return codes from DSNTIAR	100
Preparing to use DSNTIAR	100
A scenario for using DSNTIAR	100
 Chapter 7. Using a cursor to retrieve a set of rows.	 103
Accessing data by using a row-positioned cursor	103
Step 1: Declare the cursor	103
Step 2: Open the cursor	105
Step 3: Specify what to do at end-of-data	105
Step 4: Execute SQL statements	106
Using FETCH statements	106
Using positioned UPDATE statements	107
Using positioned DELETE statements	107
Step 5: Close the cursor	108
Accessing data by using a rowset-positioned cursor	108
Step 1: Declare the rowset cursor	108
Step 2: Open the rowset cursor	108
Step 3: Specify what to do at end-of-data for a rowset cursor	109
Step 4: Execute SQL statements with a rowset cursor	109
Using a multiple-row FETCH statement with host variable arrays	109
Using a multiple-row FETCH statement with a descriptor	109
Using rowset-positioned UPDATE statements	111
Using rowset-positioned DELETE statements	112
Number of rows in a rowset	113
Step 5: Close the rowset cursor	113
Types of cursors	113
Scrollable and non-scrollable cursors	113

Using a non-scrollable cursor	113
Using a scrollable cursor.	114
Comparison of scrollable cursors	118
Holes in the result table of a scrollable cursor	119
Held and non-held cursors	122
Examples of using cursors	124
Chapter 8. Generating declarations for your tables using DCLGEN	131
Invoking DCLGEN through DB2I	132
Including the data declarations in your program	136
DCLGEN support of C, COBOL, and PL/I languages	136
Example: Adding a table declaration and host-variable structure to a library	138
Step 1. Specify COBOL as the host language	138
Step 2. Create the table declaration and host structure.	139
Step 3. Examine the results	140
Chapter 9. Embedding SQL statements in host languages	143
Coding SQL statements in an assembler application	143
Defining the SQL communications area.	143
If you specify STDSQL(YES)	144
If you specify STDSQL(NO)	144
Defining SQL descriptor areas.	144
Embedding SQL statements	145
Using host variables	148
Declaring host variables	148
Determining equivalent SQL and assembler data types	151
Notes on assembler variable declaration and usage.	154
Determining compatibility of SQL and assembler data types.	155
Using indicator variables	156
Handling SQL error return codes.	157
Macros for assembler applications	158
Coding SQL statements in a C or C++ application	158
Defining the SQL communication area	158
If you specify STDSQL(YES)	159
If you specify STDSQL(NO)	159
Defining SQL descriptor areas.	159
Embedding SQL statements	160
Using host variables and host variable arrays.	161
Declaring host variables	162
Declaring host variable arrays.	168
Using host structures.	173
Determining equivalent SQL and C data types	175
Notes on C variable declaration and usage	179
Notes on syntax differences for constants	180
Determining compatibility of SQL and C data types	181
Using indicator variables and indicator variable arrays	182
Handling SQL error return codes.	184
Coding considerations for C and C++	186
Coding SQL statements in a COBOL application.	186
Defining the SQL communication area	186
If you specify STDSQL(YES)	186
If you specify STDSQL(NO)	186
Defining SQL descriptor areas.	187
Embedding SQL statements	187
Using host variables and host variable arrays.	191
Declaring host variables	192
Declaring host variable arrays.	199
Using host structures.	205
Determining equivalent SQL and COBOL data types	210
Notes on COBOL variable declaration and usage	212
Determining compatibility of SQL and COBOL data types	214

Using indicator variables and indicator variable arrays	216
Handling SQL error return codes	217
Coding considerations for object-oriented extensions in COBOL	219
Coding SQL statements in a Fortran application	220
Defining the SQL communication area	220
If you specify STDSQL(YES)	220
If you specify STDSQL(NO)	220
Defining SQL descriptor areas	220
Embedding SQL statements	221
Using host variables	223
Declaring host variables	223
Determining equivalent SQL and Fortran data types	225
Notes on Fortran variable declaration and usage	226
Notes on syntax differences for constants	227
Determining compatibility of SQL and Fortran data types	227
Using indicator variables	228
Handling SQL error return codes	229
Coding SQL statements in a PL/I application	230
Defining the SQL communication area	230
If you specify STDSQL(YES)	230
If you specify STDSQL(NO)	230
Defining SQL descriptor areas	230
Embedding SQL statements	231
Using host variables and host variable arrays	233
Declaring host variables	234
Declaring host variable arrays	237
Using host structures	240
Determining equivalent SQL and PL/I data types	241
Notes on PL/I variable declaration and usage	244
Determining compatibility of SQL and PL/I data types	245
Using indicator variables and indicator variable arrays	246
Handling SQL error return codes	247
Coding SQL statements in a REXX application	249
Defining the SQL communication area	249
Defining SQL descriptor areas	250
Accessing the DB2 REXX Language Support application programming interfaces	250
Embedding SQL statements in a REXX procedure	252
Using cursors and statement names	253
Using REXX host variables and data types	254
Determining equivalent SQL and REXX data types	254
Letting DB2 determine the input data type	254
Ensuring that DB2 correctly interprets character input data	255
Passing the data type of an input variable to DB2	256
Retrieving data from DB2 tables	256
Using indicator variables	257
Setting the isolation level of SQL statements in a REXX procedure	258
Chapter 10. Using constraints to maintain data integrity	259
Using check constraints	259
Check constraint considerations	259
When check constraints are enforced	260
How check constraints set CHECK-pending status	260
Using referential constraints	261
Parent key columns	261
Defining a parent key and a unique index	262
Incomplete definition	263
Recommendations for defining primary keys	263
Defining a foreign key	264
The relationship name	264
Indexes on foreign keys	264
The FOREIGN KEY clause in ALTER TABLE	265

Restrictions on cycles of dependent tables	265
Maintaining referential integrity when using data encryption	266
Referential constraints on tables with multilevel security with row-level granularity.	266
Using informational referential constraints.	267
Chapter 11. Using DB2-generated values as keys	269
Using ROWID columns as keys	269
Defining a ROWID column.	269
Direct row access	270
Using identity columns as keys	270
Defining an identity column	271
Parent keys and foreign keys	272
Using values obtained from sequence objects as keys	273
Creating a sequence object	273
Referencing a sequence object	274
Keys across multiple tables.	274
Chapter 12. Using triggers for active data	277
Example of creating and using a trigger	277
Parts of a trigger	279
Trigger name	279
Subject table.	279
Trigger activation time	279
Triggering event	279
Granularity	280
Transition variables	281
Transition tables	282
Triggered action	283
Trigger condition	283
Trigger body	283
Invoking stored procedures and user-defined functions from triggers.	285
Passing transition tables to user-defined functions and stored procedures	286
Trigger cascading	286
Ordering of multiple triggers	287
Interactions between triggers and referential constraints	288
Interactions between triggers and tables that have multilevel security with row-level granularity	289
Creating triggers to obtain consistent results	290

Chapter 6. Basics of coding SQL in an application program

Suppose you are writing an application program to access data in a DB2 database. When your program executes an SQL statement, the program needs to communicate with DB2. When DB2 finishes processing an SQL statement, DB2 sends back a return code, and your program should test the return code to examine the results of the operation.

To communicate with DB2, you need to perform the following actions:

- Choose a method for communicating with DB2. You can use one of the following methods:
 - Static SQL
 - Embedded dynamic SQL
 - Open Database Connectivity (ODBC)
 - JDBC application support
 - SQLJ application support

This book discusses embedded SQL. See Chapter 24, “Coding dynamic SQL in application programs,” on page 593 for a comparison of static and embedded dynamic SQL and an extended discussion of embedded dynamic SQL.

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. For more information about using both JDBC and SQLJ, see *DB2 Application Programming Guide and Reference for Java*.

- Delimit SQL statements, as described in “Delimiting an SQL statement” on page 78.
- Declare the tables that you use, as described in “Declaring table and view definitions” on page 79. (This is optional.)
- Declare the data items for passing data between DB2 and a host language, according to the host language rules described in Chapter 9, “Embedding SQL statements in host languages,” on page 143.
- Code SQL statements to access DB2 data. See “Accessing data using host variables, variable arrays, and structures” on page 79.

For information about using the SQL language, see Part 1, “Using SQL queries,” on page 1 and *DB2 SQL Reference*. Details about how to use SQL statements within application programs are described in Chapter 9, “Embedding SQL statements in host languages,” on page 143.

- Declare an SQL communications area (SQLCA). Alternatively, you can use the GET DIAGNOSTICS statement to provide diagnostic information about the last SQL statement that executed. See “Checking the execution of SQL statements” on page 91 for more information.

In addition to these basic requirements, you should also consider the following special topics:

- Cursors — Chapter 7, “Using a cursor to retrieve a set of rows,” on page 103 discusses how to use a cursor in your application program to select a set of rows and then process the set either one row at a time or one rowset at a time.
- DCLGEN — Chapter 8, “Generating declarations for your tables using DCLGEN,” on page 131 discusses how to use DB2’s declarations generator, DCLGEN, to obtain accurate SQL DECLARE statements for tables and views.

This section includes information about using SQL in application programs written in assembler, C, C++, COBOL, Fortran, PL/I, and REXX.

Conventions used in examples of coding SQL statements

The SQL statements shown in this section 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. For a description of the tables, see Appendix A, “DB2 sample tables,” on page 993.
- An SQL example does not necessarily show the complete syntax of an SQL statement. For the complete description and syntax of any of the statements described in this book, see Chapter 5 of *DB2 SQL Reference*.
- Examples do not take referential constraints into account. For more information about how referential constraints affect SQL statements, and examples of how SQL statements operate with referential constraints, see Chapter 2, “Working with tables and modifying data,” on page 19 and “Using referential constraints” on page 261.

Some of the examples vary from these conventions. Exceptions are noted where they occur.

Delimiting an SQL statement

For languages other than REXX, delimit an SQL statement in your program with the beginning keyword EXEC SQL and a statement terminator. The terminators for the languages that are described in this book are the following:

Language	SQL Statement Terminator
Assembler	End of line or end of last continued line
C and C++	Semicolon (;)
COBOL	END-EXEC.
Fortran	End of line or end of last continued line
PL/I	Semicolon (;)

For REXX, precede the statement with EXEC SQL. If the statement is in a literal string, enclose it in single or double quotation marks.

Example: Use EXEC SQL and END-EXEC. to delimit an SQL statement in a COBOL program:

```
EXEC SQL
    an SQL statement
END-EXEC.
```

Declaring table and view definitions

Before your program issues SQL statements that select, insert, update, or delete data, you should declare the tables and views that your program accesses. To do this, include an SQL DECLARE statement in your program.

You do not need to declare tables or views, but doing so offers advantages. One advantage is documentation. For example, the DECLARE statement specifies the structure of the table or view you are working with, and the data type of each column. You can refer to the DECLARE statement for the column names and data types in the table or view. Another advantage is that 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 do not correspond to the SQL DECLARE statements in your program.

One way to declare a table or view is to code a DECLARE statement in the WORKING-STORAGE SECTION or LINKAGE SECTION within the DATA DIVISION of your COBOL program. Specify the name of the table and list each column and its data type. When you declare a table or view, you specify DECLARE *table-name* TABLE regardless of whether the table-name refers to a table or a view.

For example, the DECLARE TABLE statement for the DSN8810.DEPT table looks like the following DECLARE statement in COBOL:

```
EXEC SQL
  DECLARE DSN8810.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.
```

As an alternative to coding the DECLARE statement yourself, you can use DCLGEN, the declarations generator that is supplied with DB2. For more information about using DCLGEN, see Chapter 8, “Generating declarations for your tables using DCLGEN,” on page 131.

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.

Accessing data using host variables, variable arrays, and structures

You can access data by using host variables, host variable arrays, and host structures within the SQL statements that you use in your application program.

A *host variable* is a data item that is declared in the host language for use within an SQL statement. Using host variables, you can:

- 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 in columns 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

A *host variable array* is a data array that is declared in the host language for use within an SQL statement. Using host variable arrays, you can:

- Retrieve data into host variable arrays for your application program's use
- Place data into host variable arrays to insert rows into a table

A *host structure* is a group of host variables that is referred to by a single name. You can use host structures in all host languages except REXX. Host structures are defined by statements of the host language. You can refer to a host structure in any context where you would 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.

This section describes:

- "Using host variables " on page 86
- "Using host variable arrays " on page 86
- "Using host structures " on page 90

Using host variables

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, as described in Chapter 9, "Embedding SQL statements in host languages," on page 143. You must declare the name of the host variable in the host program before you use it.

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 more performance suggestions, see Part 6, "Additional programming techniques," on page 583.

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 using dynamic SQL. See Chapter 24, "Coding dynamic SQL in application programs," on page 593 for more information.)

Host variables follow the naming conventions of the host language. A colon (:) must precede host variables that are used in SQL statements so DB2 can distinguish a variable name from a column name. A colon must **not** precede host variables outside of SQL statements.

For more information about declaring host variables, see the appropriate language section:

- **Assembler:** "Declaring host variables" on page 148
- **C and C++:** "Declaring host variables" on page 162
- **COBOL:** "Declaring host variables" on page 192

- **Fortran:** “Declaring host variables” on page 223
- **PL/I:** “Declaring host variables” on page 234
- **REXX:** “Using REXX host variables and data types” on page 254.

This section describes the following ways to use host variables:

- “Retrieving a single row of data into host variables”
- “Updating data using values in host variables” on page 82
- “Inserting data from column values that use host variables” on page 83
- “Using indicator variables with host variables” on page 83
- “Assignments and comparisons using different data types” on page 85
- “Changing the coded character set ID of host variables” on page 85

Retrieving a single row of data into host variables

You can use one or more host variables to specify a program data area that is to contain the column values of a retrieved row. The INTO clause of the SELECT statement names one or more host variables to contain the retrieved column values. The named variables correspond one-to-one with the list of column names in the SELECT statement.

If you do not know how many rows DB2 will return, or if you expect more than one row to return, you must use an alternative to the SELECT ... INTO statement. The DB2 *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. For information about using cursors, see Chapter 7, “Using a cursor to retrieve a set of rows,” on page 103.

Example: Retrieving a single row: Suppose you are retrieving the LASTNAME and WORKDEPT column values from the DSN8810.EMP table for a particular employee. You can define a host variable in your program to hold each column and then name the host variables with an INTO clause, as in the following COBOL example:

```
MOVE '000110' TO CBLEMPNO.
EXEC SQL
    SELECT LASTNAME, WORKDEPT
        INTO :CBLNAME, :CBLDEPT
        FROM DSN8810.EMP
        WHERE EMPNO = :CBLEMPNO
END-EXEC.
```

Note that 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. 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 DSN8810.EMP table.

You can use a host variable to specify a value in a search condition. For this example, you have defined a host variable CBLEMPNO 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.

If the SELECT ... INTO statement returns more than one row, an error occurs, and any data that is returned is undefined and unpredictable.

To prevent undefined and unpredictable data from being returned, you can use the FETCH FIRST 1 ROW ONLY clause to ensure that only one row is returned. For example:

```
EXEC SQL
    SELECT LASTNAME, WORKDEPT
    INTO :CBLNAME, :CBLDEPT
    FROM DSN8810.EMP
    FETCH FIRST 1 ROW ONLY
END-EXEC.
```

You can include an ORDER BY clause in the preceding example. This gives your application some control over which row is returned when you use a FETCH FIRST 1 ROW ONLY clause in a SELECT INTO statement.

```
EXEC SQL
    SELECT LASTNAME, WORKDEPT
    INTO :CBLNAME, :CBLDEPT
    FROM DSN8810.EMP
    ORDER BY LASTNAME
    FETCH FIRST 1 ROW ONLY
END-EXEC.
```

When you specify both the ORDER BY clause and the FETCH FIRST clause, ordering is done first and then the first row is returned. This means that the ORDER BY clause determines 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.

Example: Specifying expressions in the SELECT clause: When you specify a list of items in the SELECT clause, you can use 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:

```
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 DSN8810.EMP
    WHERE EMPNO = :PERSON
END-EXEC.
```

The following results have column headings that represent the names of the host variables:

EMP-NUM	PERSON-NAME	EMP-SAL	EMP-RAISE	EMP-TTL
000220	LUTZ	29840	4476	34316

Example: Specifying summary values in the SELECT clause: You can request summary values to be returned from aggregate functions. For example:

```
MOVE 'D11' TO DEPTID.
EXEC SQL
    SELECT WORKDEPT, AVG(SALARY)
    INTO :WORK-DEPT, :AVG-SALARY
    FROM DSN8810.EMP
    WHERE WORKDEPT = :DEPTID
END-EXEC.
```

Updating data using values in host variables

You can set or change values in a DB2 table to the value of host variables. To do this, use the host variable name in the SET clause of the UPDATE statement.

Example: Updating a single row: The following example changes an employee's phone number:

```
MOVE '4246' TO NEWPHONE.  
MOVE '000110' TO EMPID.  
EXEC SQL  
    UPDATE DSN8810.EMP  
        SET PHONENO = :NEWPHONE  
        WHERE EMPNO = :EMPID  
END-EXEC.
```

Example: Updating multiple rows: The following example gives the employees in a particular department a salary increase of 10%:

```
MOVE 'D11' TO DEPTID.  
EXEC SQL  
    UPDATE DSN8810.EMP  
        SET SALARY = 1.10 * SALARY  
        WHERE WORKDEPT = :DEPTID  
END-EXEC.
```

Inserting data from column values that use host variables

You can insert a single row of data into a DB2 table by using the INSERT statement with column values in the VALUES clause. A column value can be a host variable, a constant, or any valid combination of host variables and constants.

To insert multiple rows, you can use the form of the INSERT statement that selects values from another table or view. You can also use a form of the INSERT statement that inserts multiple rows from values that are provided in host variable arrays. For more information, see “Inserting multiple rows of data from host variable arrays” on page 87.

Example: The following example inserts a single row into the activity table:

```
EXEC SQL  
    INSERT INTO DSN8810.ACT  
        VALUES (:HV-ACTNO, :HV-ACTKWD, :HV-ACTDESC)  
END-EXEC.
```

Using indicator variables with host variables

Indicator variables are small integers that you can use to:

- 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 during assignment to a host variable
- Determine that a character value could not be converted during assignment to a host variable
- Determine the seconds portion of a time value that was truncated during assignment to a host variable

Retrieving data and testing the indicator variable: When DB2 retrieves the value of a column into a host variable, you can test the indicator variable that is associated with that host variable:

- If the value of the indicator variable is less than zero, the column value is null. The value of the host variable does not change from its previous value. If it is null because of a numeric or character conversion error, or an arithmetic expression error, DB2 sets the indicator variable to -2. See “Handling arithmetic or conversion errors” on page 93 for more information.

- If the indicator variable contains a positive integer, the retrieved value is truncated, and the integer is the original length of the string.
- If the value of the indicator variable is zero, the column value is nonnull. If the column value is a character string, the retrieved value is not truncated.

An error occurs if you do not use an indicator variable and DB2 retrieves a null value.

You can specify an indicator variable, preceded by a colon, immediately after the host variable. Optionally, you can use the word INDICATOR between the host variable and its indicator variable. Thus, the following two examples are equivalent:

```
EXEC SQL
  SELECT PHONENO
    INTO :CBLPHONE:INDNULL
    FROM DSN8810.EMP
   WHERE EMPNO = :EMPID
END-EXEC.
```

```
EXEC SQL
  SELECT PHONENO
    INTO :CBLPHONE INDICATOR :INDNULL
    FROM DSN8810.EMP
   WHERE EMPNO = :EMPID
END-EXEC.
```

You can then test INDNULL for a negative value. If it is negative, the corresponding value of PHONENO is null, and you can disregard the contents of CBLPHONE.

When you use a cursor to fetch a column value, you can use the same technique to determine whether the column value is null.

Inserting null values into columns by using host variable indicators: You can use an indicator variable to insert a null value from a host variable into a column. When DB2 processes INSERT and UPDATE 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.

For example, 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 in the UPDATE statement. For example:

```
EXEC SQL
  UPDATE DSN8810.EMP
    SET PHONENO = :NEWPHONE:PHONEIND
   WHERE EMPNO = :EMPID
END-EXEC.
```

When NEWPHONE contains a non-null value, set 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.
```

Testing for a null column value: 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.

To test whether a column has a null value, use the IS NULL predicate or the IS DISTINCT FROM predicate. For example, the following code does **not** select the employees who have no phone number:

```
MOVE -1 TO PHONE-IND.  
EXEC SQL  
    SELECT LASTNAME  
        INTO :PGM-LASTNAME  
        FROM DSN8810.EMP  
        WHERE PHONENO = :PHONE-HV:PHONE-IND  
END-EXEC.
```

You can use the IS NULL predicate to select employees who have no phone number, as in the following statement:

```
EXEC SQL  
    SELECT LASTNAME  
        INTO :PGM-LASTNAME  
        FROM DSN8810.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), you would need to 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 DSN8810.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 statement using the NOT form of the IS DISTINCT FROM predicate, as in the following statement:

```
EXEC SQL  
    SELECT LASTNAME  
        INTO :PGM-LASTNAME  
        FROM DSN8810.EMP  
        WHERE PHONENO IS NOT DISTINCT FROM :PHONE-HV:PHONE-IND  
END-EXEC.
```

Assignments and comparisons using different data types

For assignments and comparisons involving a DB2 column and a host variable of a different data type or length, you can expect conversions to occur. If you assign retrieved data to a host variable or compare retrieved data to a value in a host variable, see Chapter 2 of *DB2 SQL Reference* for the rules that are associated with these operations.

Changing the coded character set ID of host variables

All DB2 string data, other than BLOB data, has an encoding scheme and a coded character set ID (CCSID) associated with it. You can use the DECLARE VARIABLE statement to associate an encoding scheme and a CCSID with individual host variables. 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.

You can use the DECLARE VARIABLE statement in static or dynamic SQL applications. However, you cannot use the DECLARE VARIABLE statement to control the CCSID and encoding scheme of data that you retrieve or update using an SQLDA. See “Changing the CCSID for retrieved data” on page 619 for information on changing the CCSID in an SQLDA.

When you use a DECLARE VARIABLE statement in a program, put the DECLARE VARIABLE statement after the corresponding host variable declaration and before your first reference to that host variable.

Example: Using a DECLARE VARIABLE statement to change the encoding scheme of retrieved data: Suppose that you are writing a C program that runs on a DB2 UDB 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 CCSID UNICODE.

```
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 DECLARE VARIABLE statements like these:

```
EXEC SQL BEGIN DECLARE SECTION;
char hvpartnum[11];
EXEC SQL DECLARE :hvpartnum VARIABLE CCSID 1208;
sqldbcchar hvjpnname[11];
EXEC SQL DECLARE :hvjpnname VARIABLE CCSID 1200;
struct {
    short len;
    char d[30];
} hvengname;
EXEC SQL DECLARE :hvengname VARIABLE CCSID 1208;
EXEC SQL END DECLARE SECTION;
```

The BEGIN DECLARE SECTION and END DECLARE SECTION statements mark the beginning and end of a host variable declare section.

Using host variable arrays

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 that are described in Chapter 9, “Embedding SQL statements in host languages,” on page 143. 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.

For more information about declaring host variable arrays, see the appropriate language section:

- **C or C++:** “Declaring host variable arrays” on page 168
- **COBOL:** “Declaring host variable arrays” on page 199
- **PL/I:** “Declaring host variable arrays” on page 237

Assembler support for the multiple-row FETCH and INSERT statements is limited to the FETCH statement with the USING DESCRIPTOR clause and the dynamic INSERT statement using EXECUTE USING DESCRIPTOR. The DB2 precompiler does not recognize declarations of host variable arrays for Assembler; it recognizes these declarations only in C, COBOL, and PL/I.

This section describes the following ways to use host variable arrays:

- “Retrieving multiple rows of data into host variable arrays”
- “Inserting multiple rows of data from host variable arrays”
- “Using indicator variable arrays with host variable arrays”

Retrieving multiple rows of data into host variable arrays

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. For information about using rowset cursors, see “Accessing data by using a rowset-positioned cursor” on page 108.

Inserting multiple rows of data from host variable arrays

You can use a form of the INSERT 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.

Example: 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 DSN8810.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.

Using indicator variable arrays with host variable arrays

You can use indicator variable arrays with host variable arrays in the same way that you use indicator variables with host variables. For details, see “Using indicator variables with host variables” on page 83. An indicator variable array must have at least as many entries as its host variable array.

Retrieving data and using indicator arrays: When you retrieve data into a host variable array, if a value in its indicator array is negative, you can disregard the contents of the corresponding element in the host variable array. If a value in an indicator array is:

- 1 The corresponding row in the column that is being retrieved is null.
- 2 DB2 returns a null value because an error occurred in numeric conversion or in an arithmetic expression in the corresponding row.

-3 DB2 returns a null value because a hole was detected for the corresponding row during a multiple-row FETCH operation.

For information about the multiple-row FETCH operation, see “Step 4: Execute SQL statements with a rowset cursor” on page 109. For information about holes in the result table of a cursor, see “Holes in the result table of a scrollable cursor” on page 119.

Specifying an indicator array: You can specify an indicator variable array, preceded by a colon, immediately after the host variable array. Optionally, you can use the word INDICATOR between the host variable array and its indicator variable array.

Example: Suppose that you declare a scrollable rowset cursor by using the following statement:

```
EXEC SQL
  DECLARE CURS1 SCROLL CURSOR WITH ROWSET POSITIONING FOR
  SELECT PHONENO
  FROM DSN8810.EMP
END-EXEC.
```

For information about using rowset cursors, see “Accessing data by using a rowset-positioned cursor” on page 108.

The following two specifications of indicator arrays in the multiple-row FETCH statement are equivalent:

<pre>EXEC SQL FETCH NEXT ROWSET CURS1 FOR 10 ROWS INTO :CBLPHONE :INDNULL END-EXEC.</pre>	<pre>EXEC SQL FETCH NEXT ROWSET CURS1 FOR 10 ROWS INTO :CBLPHONE INDICATOR :INDNULL END-EXEC.</pre>

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.

Inserting null values by using indicator arrays: You can use a negative value in an indicator array to insert a null value into a column.

Example: 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:

```
/* Initialize each indicator array */
for (i=0; i<10; i++) {
  ind1[i] = 0;
  ind2[i] = 0;
  ind3[i] = -1;
}

EXEC SQL
  INSERT INTO DSN8810.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.

```
# Identifying errors during output host variable processing: Output host variable
# processing is the process of moving data that is retrieved from DB2 (such as from a
# FETCH) to an application. Errors that occur while processing output host variables
# do not affect the position of the cursor, and are usually caused by a problem in
# converting from one data type to another.
```

```
# Example: Suppose that an integer value of 32768 is fetched into a smallint host
# variable. The conversion might cause an error if you provide insufficient
# conversion information to DB2.
```

```
# If an indicator variable is provided during output host variable processing or if
# data type conversion is not required, a positive SQLCODE is returned for the row
# in most cases. In other cases where data conversion problems occur, a negative
# SQLCODE is returned 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 a positive SQLCODE is returned for the statement (SQLSTATE
# 01668, SQLCODE +354). To determine which rows cause errors when SQLCODE =
# +354, you can use GET DIAGNOSTICS.
```

```
# Example: Suppose that no indicator variables are provided for 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, a negative SQLCODE is recorded and processing
# continues until the 10 rows are fetched. When SQLCODE = +354 is returned for
# the statement, you can use GET DIAGNOSTICS to determine which errors occur
# for which rows. The following statement returns num_rows = 10 and num_cond =
# 3:
```

```
GET DIAGNOSTICS :num rows = ROW COUNT, :num cond = NUMBER;
```

To investigate the three conditions, use the following statements:

Statement A

```
GET DIAGNOSTICS CONDITION 3 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2 RETURNED SQLCODE, :row num = DB2 ROW NUMBER;
```

Output A

```
sqlstate = 22003
sqlcode  = -304
row num  = 5
```

Statement B

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2 RETURNED SQLCODE, :row num = DB2 ROW NUMBER;
```

Output B

```
sqlstate = 22003
sqlcode = -802
row num = 7
```

Statement C

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2 RETURNED SQLCODE, :row num = DB2 ROW NUMBER;
```

```
#
#
#      sqlstate = 01668
#      sqlcode  = +354
#      row_num  = 0

#
#      The fifth row has a data mapping error (-304) for column 1 and 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.
```

Using host structures

You can substitute a host structure for one or more host variables. You can also use indicator variables (or indicator structures) with host structures.

Retrieving a single row of data into a host structure

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 DSN8810.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 DSN8810.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. For more detailed information about coding a host structure in your program, see Chapter 9, “Embedding SQL statements in host languages,” on page 143. For more information about using DCLGEN and the restrictions that apply to the C language, see Chapter 8, “Generating declarations for your tables using DCLGEN,” on page 131.

Using indicator variables with host structures

You can define an *indicator structure* (an array of halfword integer variables) to support a host structure. You define indicator structures in the DATA DIVISION section of your COBOL program. If the column values your program retrieves into a host structure can be null, you can attach an indicator structure name to the host structure name. This allows DB2 to notify your program about each null value it returns to a host variable in the host structure. For example:

```
01 PEMP-ROW.
   10 EMPNO                PIC X(6).
   10 FIRSTNME.
      49 FIRSTNME-LEN      PIC S9(4) USAGE COMP.
      49 FIRSTNME-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, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, BIRTHDATE
  INTO :PEMP-ROW:EMP-IND
  FROM DSN8810.EMP
  WHERE EMPNO = :EMPNO
END-EXEC.

```

In this example, EMP-IND is an array containing six values, which you can test 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.

Because this example selects rows from the table DSN8810.EMP, some of the values in EMP-IND are always zero. The first four columns of each row are defined NOT NULL. In the preceding example, DB2 selects the values for a row of data into a host structure. You must use a corresponding structure for the indicator variables to determine which (if any) selected column values are null. For information on using the IS NULL keyword phrase in WHERE clauses, see “Selecting rows using search conditions: WHERE” on page 8.

Checking the execution of SQL statements

You can check the execution of SQL statements in various ways:

- By displaying specific fields in the SQLCA; see “Using the SQL communication area (SQLCA).”
- By testing SQLCODE or SQLSTATE for specific values; see “SQLCODE and SQLSTATE” on page 92.
- By using the WHENEVER statement in your application program; see “The WHENEVER statement” on page 93.
- By testing indicator variables to detect numeric errors; see “Handling arithmetic or conversion errors” on page 93.
- 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; see “The GET DIAGNOSTICS statement” on page 94.
- By calling DSNTIAR to display the contents of the SQLCA; see “Calling DSNTIAR to display SQLCA fields” on page 98.

Using the SQL communication area (SQLCA)

A program that includes SQL statements can have an area set apart for communication with DB2—an *SQL communication area* (SQLCA). 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. For details, see “SQLCODE and SQLSTATE” on page 92.
- 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 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. For details, see “Accessing data by using a rowset-positioned cursor” on page 108.
- 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.

See Appendix D of *DB2 SQL Reference* for a description of all the fields in the SQLCA.

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 allows an application program to check for errors in the same way for different IBM database management systems. See Appendix C of *DB2 Codes* for a complete list of possible SQLSTATE values.

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.

The WHENEVER statement

The WHENEVER statement causes DB2 to check the SQLCA and continue processing your program, or branch to another area in your program if an error, exception, or warning occurs. The condition handling area of your program can then examine SQLCODE or SQLSTATE to react specifically to the error or exception.

The WHENEVER statement is not supported for REXX. For information on REXX error handling, see “Embedding SQL statements in a REXX procedure” on page 252.

The WHENEVER statement allows 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.

Handling arithmetic or conversion errors

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.

The GET DIAGNOSTICS 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. For more information, see Chapter 5 of *DB2 SQL Reference*.

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.

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.

Restriction: If you issue a GET DIAGNOSTICS statement immediately following an SQL statement that uses private protocol access, DB2 returns an error.

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. For a description of available items and their data types, see “Data types for GET DIAGNOSTICS items” on page 95.

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 Figure 10, 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.

In the activity table, the ACTNO column is defined as SMALLINT. Suppose that

```
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 DSN8810.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);
}
```

Figure 10. Using GET DIAGNOSTICS to return the number of rows and conditions returned and condition information

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, or see *DB2 Codes* for information about SQLCODE -302.

Data types for GET DIAGNOSTICS items

Table 4 on page 96, Table 5 on page 97, and Table 6 on page 98 specify the data types for the statement, condition, and connection information items that you can request by using the GET DIAGNOSTICS statement. You must declare each target host variable with a data type that is compatible with the data type of the requested item.

Table 4. 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
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.	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 A, I, or S).	CHAR(1)
DB2_SQL_ATTR _CURSOR_TYPE	After an ALLOCATE or OPEN statement, this item indicates whether the cursor is 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 conditions items were discarded because of insufficient storage (Y or N).	CHAR(1)

Table 4. Data types for GET DIAGNOSTICS items that return statement information (continued)

Item	Description	Data type
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
ROW_COUNT	After DELETE, INSERT, UPDATE, 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.	DECIMAL(31,0)

Table 5. 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_ <i>n</i>	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

Table 5. Data types for GET DIAGNOSTICS items that return condition information (continued)

Item	Description	Data type
MESSAGE_TEXT	After any SQL statement, this item contains the message text associated with the SQLCODE.	VARCHAR(32672)
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 6. 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). For more information, see Chapter 5 of <i>DB2 SQL Reference</i> .	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)

For a complete description of the GET DIAGNOSTICS items, see Chapter 5 of *DB2 SQL Reference*.

Calling DSNTIAR to display SQLCA fields

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.

You can find the programming language-specific syntax and details for calling DSNTIAR on the following pages:

For Assembler programs, see page 157
 For C programs, see page 184
 For COBOL programs, see page 217
 For Fortran programs, see page 229
 For PL/I programs, see page 247

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.

Defining a message output area

The calling program must allocate enough storage in the message output area to hold all of the message text. 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.

Figure 11 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.

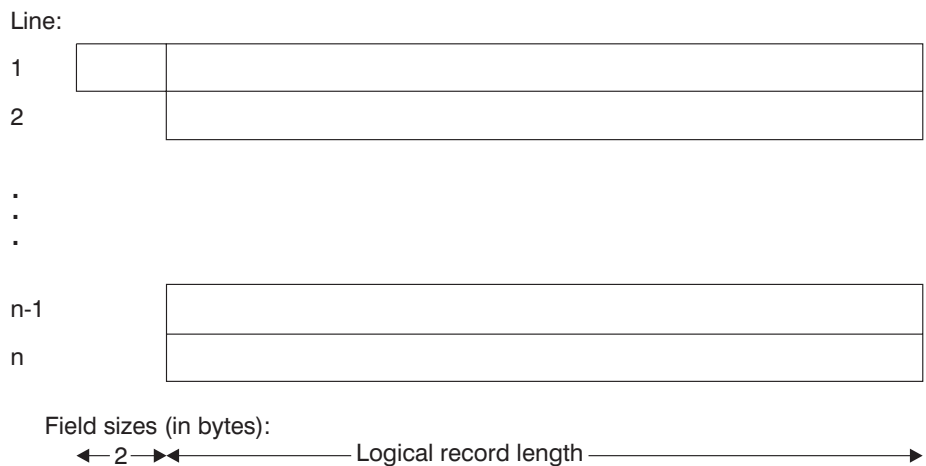


Figure 11. Format of the message output area

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

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.

Preparing to use DSNTIAR

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 install 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.

A scenario for using DSNTIAR

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 SQLCODEs 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
```

Chapter 7. Using a cursor to retrieve a set of rows

Use a *cursor* in an application program to retrieve rows from a table or from a result set that is returned by a stored procedure. This chapter explains how your application program can use a cursor to retrieve rows from a table. For information about using a cursor to retrieve rows from a result set that is returned by a stored procedure, see Chapter 25, “Using stored procedures for client/server processing,” on page 629.

When you execute a SELECT statement, you retrieve a set of rows. That set of rows is called the *result table* of the SELECT statement. In an application program, you can 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.”
- 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 108.

This chapter also includes the following sections:

- “Types of cursors” on page 113
- “Examples of using cursors” on page 124

Accessing data by using a row-positioned cursor

The basic steps in using a row-positioned cursor are:

1. Execute a DECLARE CURSOR statement to define the result table on which the cursor operates. See “Step 1: Declare the cursor.”
2. Execute an OPEN CURSOR to make the cursor available to the application. See “Step 2: Open the cursor” on page 105.
3. Specify what the program is to do when all rows have been retrieved. See “Step 3: Specify what to do at end-of-data” on page 105.
4. Execute multiple SQL statements to retrieve data from the table or modify selected rows of the table. See “Step 4: Execute SQL statements” on page 106.
5. Execute a CLOSE CURSOR statement to make the cursor unavailable to the application. See “Step 5: Close the cursor” on page 108.

Your program can have several cursors, each of which performs the previous steps.

Step 1: Declare the cursor

To define and identify a set of rows to be accessed with a 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. See Chapter 4 of *DB2 SQL Reference* for a complete list of clauses that you can use in the SELECT statement.

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 DSN8810.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 DSN8810.EMP X
    WHERE EXISTS
      (SELECT *
       FROM DSN8810.PROJ Y
       WHERE X.EMPNO=Y.RESPEMP
       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. Refer to Part 3 (Volume 1) of *DB2 Administration Guide* for a discussion of multilevel security with row-level granularity.

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 DSN8810.EMP X
    WHERE EXISTS
      (SELECT *
       FROM DSN8810.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 DSN8810.EMP X
WHERE EXISTS
  (SELECT *
   FROM DSN8810.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. For information about these options, see Table 64 on page 484. 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. The defining characteristics of a read-only result tables are described in greater detail in the discussion of DECLARE CURSOR in Chapter 5 of *DB2 SQL Reference*.

Step 2: Open the cursor

To tell DB2 that you are ready to process the first row of the result table, 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

For more information, see “The effect of sorts on OPEN CURSOR” on page 835.

Step 3: Specify what to do at end-of-data

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.
```

Your program must anticipate and handle an end-of-data whenever you use a cursor to fetch a row. For more information about the WHENEVER NOT FOUND statement, see “Checking the execution of SQL statements” on page 91.

Step 4: Execute SQL statements

You execute one of these SQL statements using the cursor:

- A FETCH statement
- A positioned UPDATE statement
- A positioned DELETE statement

Using FETCH 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. See “Scrollable and non-scrollable cursors” on page 113 for more information.

When you query a remote subsystem with FETCH, consider using block fetch for better performance. For more information see “Using block fetch in distributed applications” on page 458. Block fetch processes rows ahead of the current row. You cannot use a block fetch when you perform a positioned update or delete operation.

Using positioned UPDATE statements

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 DSN8810.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 INSERT 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.

Using positioned DELETE statements

After your program has executed a FETCH statement to retrieve the current row, you can use a positioned DELETE statement to delete that row. A example of a positioned DELETE statement looks like this:

```
EXEC SQL
  DELETE FROM DSN8810.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 INSERT 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.

Step 5: Close the cursor

If you finish processing the rows of the result table and want to use the cursor again, issue a CLOSE statement to close the cursor and then issue an OPEN statement to reopen the cursor. An example of a CLOSE statement looks like this:

```
EXEC SQL
  CLOSE C1
END-EXEC.
```

When you finish processing the rows of the result table, and the cursor is no longer needed, you can let DB2 automatically close the cursor when the current transaction terminates or when your program terminates.

Recommendation: To free the resources that are held by the cursor, close the cursor explicitly by issuing the CLOSE statement.

Accessing data by using a rowset-positioned cursor

The basic steps in using a rowset cursor are:

1. Execute a DECLARE CURSOR statement to define the result table on which the cursor operates. See “Step 1: Declare the rowset cursor.”
2. Execute an OPEN CURSOR to make the cursor available to the application. See “Step 2: Open the rowset cursor.”
3. Specify what the program is to do when all rows have been retrieved. See “Step 3: Specify what to do at end-of-data for a rowset cursor” on page 109.
4. Execute multiple SQL statements to retrieve data from the table or modify selected rows of the table. See “Step 4: Execute SQL statements with a rowset cursor” on page 109.
5. Execute a CLOSE CURSOR statement to make the cursor unavailable to the application. See “Step 5: Close the rowset cursor” on page 113.

Your program can have several cursors, each of which performs the previous steps.

Step 1: Declare the rowset cursor

To enable a cursor to fetch rowsets, 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 DSN8810.EMP
END-EXEC.
```

For restrictions that apply to rowset-positioned cursors and row-positioned cursors, see “Step 1: Declare the cursor” on page 103.

Step 2: Open the rowset cursor

To tell DB2 that you are ready to process the first rowset of the result table, 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 “Step 2: Open the cursor” on page 105.

Step 3: Specify what to do at end-of-data for a rowset cursor

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 Figure 21 on page 127.

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 “The GET DIAGNOSTICS statement” on page 94.

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 113.

Step 4: Execute SQL statements with a rowset cursor

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.

Using a multiple-row FETCH statement with host variable arrays

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.

Using a multiple-row FETCH statement with a descriptor

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.

Declare the SQLDA: 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;
};
```

Allocate the SQLDA: 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.

Set the fields in the SQLDA: 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->sqldaaid,"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->sqllen = 6;
# varptr->sqldata = (char *) hval1;
# 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->sqllen = 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->sqllen))->precision = 9;
# ((struct DECLEN *) &(varptr->sqllen))->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 $SQLN \times 44 + 16$, 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.
 - SQLEN 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.

For information about using the SQLDA in dynamic SQL, see Chapter 24, “Coding dynamic SQL in application programs,” on page 593. For a complete layout of the SQLDA and the descriptions given by the INCLUDE statement, see Appendix E of *DB2 SQL Reference*.

Open the cursor: You can open the cursor only after all of the fields have been set in the output SQLDA:

```
EXEC SQL OPEN C1;
```

Fetch the rows: 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.

Using rowset-positioned UPDATE statements

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

For information about restrictions for a positioned UPDATE, see “Using positioned UPDATE statements” on page 107.

Using the WHERE CURRENT OF clause: An example of a positioned UPDATE statement that uses the WHERE CURRENT OF clause is:


```

EXEC SQL
  UPDATE DSN8810.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.

Using the FOR ROW *n* OF ROWSET clause: An example of a positioned UPDATE statement that uses the FOR ROW *n* OF ROWSET clause is:

```

EXEC SQL
  UPDATE DSN8810.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.

Using rowset-positioned DELETE statements

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

For information about restrictions for a positioned DELETE, see “Using positioned DELETE statements” on page 107.

Using the WHERE CURRENT OF clause: An example of a positioned DELETE statement that uses the WHERE CURRENT OF clause is:

```

EXEC SQL
  DELETE FROM DSN8810.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.

Using the FOR ROW *n* OF ROWSET clause: An example of a positioned DELETE statement that uses the FOR ROW *n* OF ROWSET clause is:

```

EXEC SQL
  DELETE FROM DSN8810.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.

Number of rows in a rowset

The number of rows in a rowset is determined either explicitly or implicitly. 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 8 on page 118.

Step 5: Close the rowset cursor

If you finish processing the rows of the result table and want to use the cursor again, issue a CLOSE statement to close the cursor and then issue an OPEN statement to reopen the cursor.

When you finish processing the rows of the result table, and you no longer need the cursor, you can let DB2 automatically close the cursor when the current transaction terminates or when your program terminates.

Recommendation: To free the resources held by the cursor, close the cursor explicitly by issuing the CLOSE statement.

Types of cursors

You can declare cursors, both row-positioned and rowset-positioned, as scrollable or not scrollable, held or not held, and returnable or not returnable. The following sections discuss these characteristics:

- “Scrollable and non-scrollable cursors”
- “Held and non-held cursors” on page 122

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. For information about returnable cursors, see Chapter 25, “Using stored procedures for client/server processing,” on page 629.

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. For examples of FETCH statements that position a cursor for both rows and rowsets, see Table 8 on page 118.

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*.

Figure 12 shows a declaration for an insensitive scrollable cursor.

```
EXEC SQL DECLARE C1 INSENSITIVE SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8810.DEPT
  ORDER BY DEPTNO
END-EXEC.
```

Figure 12. Declaration for an insensitive scrollable row cursor

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.

Figure 13 shows a declaration for a sensitive static scrollable cursor.

```
EXEC SQL DECLARE C2 SENSITIVE STATIC SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8810.DEPT
  ORDER BY DEPTNO
END-EXEC.
```

Figure 13. Declaration for a sensitive static scrollable row cursor

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.

Figure 14 shows a declaration for a sensitive dynamic scrollable cursor.

```
EXEC SQL DECLARE C2 SENSITIVE DYNAMIC SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8810.DEPT
  ORDER BY DEPTNO
END-EXEC.
```

Figure 14. Declaration for a sensitive dynamic scrollable cursor

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, and DELETE statements (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.

Determining attributes of a cursor by checking the SQLCA: 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)

The number of rows in the result table of a cursor when the cursor position is after the last row (when SQLCODE is equal to +100). This field is not set for dynamic scrollable cursors.

SQLERRD(2)

The number of rows in the result table of a cursor when the cursor position is after the last row (when SQLCODE is equal to +100). This field is not set for dynamic scrollable cursors.

SQLERRD(3)

The number of rows in the result table of an INSERT when the SELECT statement of the cursor contains the INSERT statement.

If the OPEN statement executes with no errors or warnings, DB2 does not set SQLWARN0 when it sets SQLWARN1, SQLWARN4, or SQLWARN5. See Appendix D of *DB2 SQL Reference* for specific information about fields in the SQLCA.

Determining attributes of a cursor by using the GET DIAGNOSTICS

statement: 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 asensitive, insensitive, or sensitive to changes that are made by other processes (A, I, or S)

DB2_SQL_ATTR_CURSOR_TYPE

Indicates whether the cursor is declared static (S for INSENSITIVE or SENSITIVE STATIC) or dynamic (D for SENSITIVE DYNAMIC)

For more information about the GET DIAGNOSTICS statement, see “The GET DIAGNOSTICS statement” on page 94.

Retrieving rows with a scrollable cursor: 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. Table 7 on page 117 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 7. 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. ABSOLUTE and RELATIVE are described in greater detail in the discussion of FETCH in Chapter 5 of *DB2 SQL Reference*.

Example: To use the cursor that is declared in Figure 12 on page 114 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;
```

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 do that, 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.

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. For information about using a multiple-row FETCH statement, see “Using a multiple-row FETCH statement with host variable arrays” on page 109.

Table 8 on page 118 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 8. 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
Note: The FOR <i>n</i> ROWS clause and the ROWSET clause are described in greater detail in the discussion of FETCH in Chapter 5 of <i>DB2 SQL Reference</i> .	

Comparison of scrollable cursors

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.

Table 9 summarizes the sensitivity values and their effects on the result table of a scrollable cursor.

Table 9. 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 UPDATE and DELETE 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.

Holes in the result table of a scrollable cursor

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, which means that the result table does not shrink to fill the space of deleted rows or the space of rows that have been updated and no longer satisfy the search condition. You cannot access a delete hole or an update hole of a static cursor, although you can remove holes in specific situations; see “Removing a delete hole or an update hole” on page 121.

- 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 Figure 15 on page 120.

1
2
3
4
5

Figure 15. Values for COL1 of table A

Now suppose that you declare the following SENSITIVE STATIC scrollable cursor, 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 Figure 16.

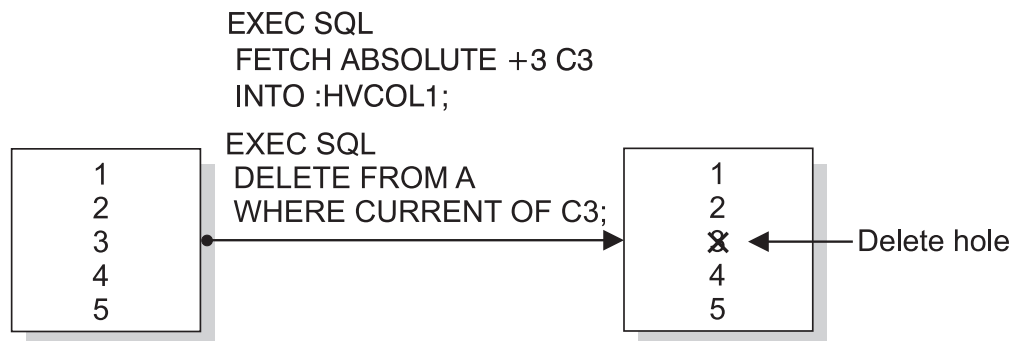


Figure 16. Creating a delete hole

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 Figure 17.

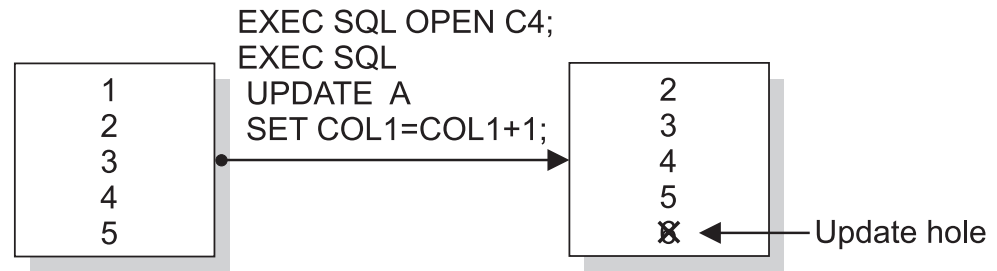


Figure 17. Creating an update hole

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.

Removing a delete hole or an update hole: You can remove a delete hole or an update hole in specific situations.

If you try to fetch from a delete hole, DB2 issues an SQL warning. If you try to update or delete a delete 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.

If you try to fetch from an update hole, DB2 issues an SQL warning. If you try to delete an update hole, DB2 issues an SQL error. However, you can convert an update hole back to a result table row by updating the row in the base table, as shown in Figure 18 on page 122. 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.

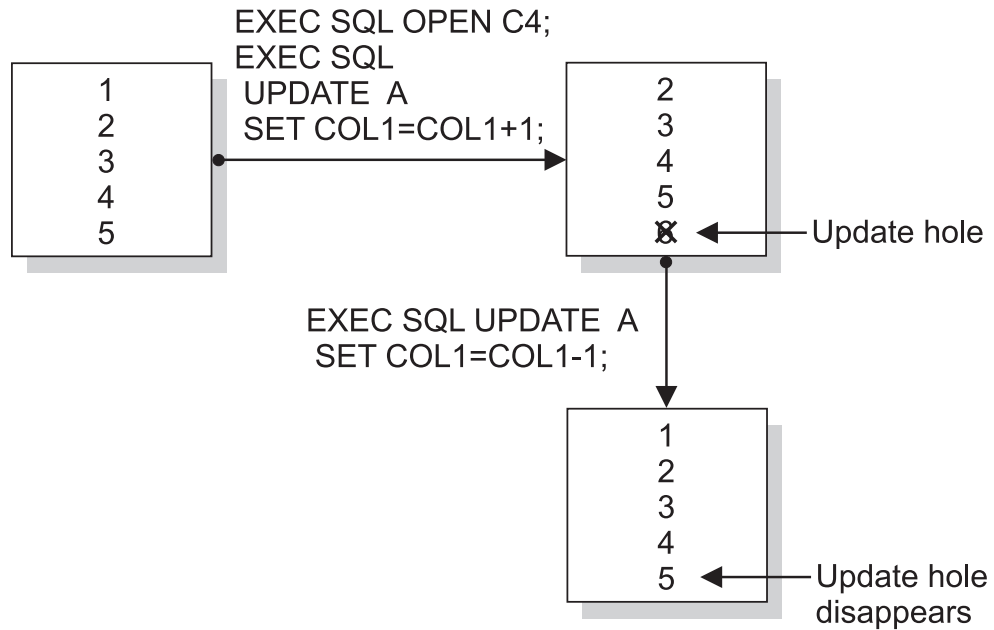


Figure 18. Removing an update hole

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.

Held and non-held cursors

When you declare a cursor, you tell DB2 whether you want the cursor to be held or not held by including or omitting the WITH HOLD clause. A held cursor, which is declared with the WITH HOLD clause, does not close after a commit operation. A cursor that is not held closes after a commit operation.

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.

The following cursor declaration causes the cursor to maintain its position in the DSN8810.EMP table after a commit point:

```
EXEC SQL
  DECLARE EMPLUPDT CURSOR WITH HOLD FOR
  SELECT EMPNO, LASTNAME, PHONENO, JOB, SALARY, WORKDEPT
```

```
FROM DSN8810.EMP  
WHERE WORKDEPT < 'D11'  
ORDER BY EMPNO  
END-EXEC.
```

Examples of using cursors

The examples in this section show the SQL statements that you include in a COBOL program to define and use cursors in the following ways:

- Non-scrollable cursor for row-positioned updates; see Figure 19 on page 125.
- Scrollable cursor to retrieve rows backward; see Figure 20 on page 126.
- Non-scrollable cursor for rowset-positioned updates; see Figure 21 on page 127.
- Scrollable cursor for rowset-positioned operations; see Figure 22 on page 128.

Figure 19 on page 125 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 DSN8810.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 DSN8810.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.

```

Figure 19. Performing cursor operations with a non-scrollable cursor

Figure 20 on page 126 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 DSN8810.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.

```

Figure 20. Performing cursor operations with a SENSITIVE STATIC scrollable cursor

Figure 21 on page 127 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 DSN8810.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 DSN8810.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.

```

Figure 21. Performing positioned update with a rowset cursor

Figure 22 on page 128 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 DSN8810.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.

```

Figure 22. Performing positioned update and delete with a sensitive rowset cursor (Part 1 of 2)

```

:
:
*****
* Branch back to fetch and process          *
* the next rowset.                         *
*****
:
:
*****
* Update row N in current rowset.          *
*****
UPDATE-ROW.
EXEC SQL
    UPDATE DSN8810.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 DSN8810.EMP
    FOR CURSOR EMPSET FOR ROW :N OF ROWSET
END-EXEC.
END-DELETE-ROW.
:
:
*****
* Close the cursor.                        *
*****
CLOSE-EMPSET.
EXEC SQL
    CLOSE EMPSET
END-EXEC.

```

Figure 22. Performing positioned update and delete with a sensitive rowset cursor (Part 2 of 2)

Chapter 8. Generating declarations for your tables using DCLGEN

DCLGEN, the declarations generator supplied with DB2, produces a DECLARE statement you can use in a C, COBOL, or PL/I program, so that you do not need to code the statement yourself. For detailed syntax of DCLGEN, see Part 3 of *DB2 Command Reference*.

DCLGEN generates a table 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's declaration, DB2 gets the relevant information from the DB2 catalog, which contains information about the table's definition and the definition of each column within the table. DCLGEN uses this information to produce a complete SQL DECLARE statement for the table or view and a corresponding PL/I, C structure declaration, or COBOL record description. You can use DCLGEN for table declarations only if the table you are declaring already exists.

You must use DCLGEN before you precompile your program. Supply the table or view name to DCLGEN before you precompile your program. To use the declarations generated by DCLGEN in your program, use the SQL INCLUDE statement. For more information about the INCLUDE statement, see Chapter 5 of *DB2 SQL Reference*.

DB2 must be active before you can use DCLGEN. You can start DCLGEN in several different ways:

- From ISPF through DB2I. Select the DCLGEN option on the DB2I Primary Option Menu panel.
- Directly from TSO. To do this, sign on to TSO, issue the TSO command DSN, and then issue the subcommand DCLGEN.
- From a CLIST, running in TSO foreground or background, that issues DSN and then DCLGEN.
- With JCL. Supply the required information, using JCL, and run DCLGEN in batch. In the *prefix.SDSNSAMP* library, sample jobs DSNTJ2C and DSNTJ2P show how to use JCL to run DCLGEN. For more information about the syntax of the DSN subcommand DCLGEN, see in *DB2 Command Reference*.

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 using the hex mode of ISPF edit.

This chapter includes the following sections:

- “Invoking DCLGEN through DB2I” on page 132
- “Including the data declarations in your program” on page 136
- “DCLGEN support of C, COBOL, and PL/I languages” on page 136
- “Example: Adding a table declaration and host-variable structure to a library” on page 138

Invoking DCLGEN through DB2I

The easiest way to start DCLGEN is through DB2I. Figure 23 shows the DCLGEN panel you reach by selecting the DCLGEN option on the DB2I Primary Option Menu panel. For more instructions on using DB2I, see “Using ISPF and DB2 Interactive” on page 518.

#####

```

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 23. DCLGEN panel

Fill in the DCLGEN panel as follows:

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:

'DON' 'S TABLE'

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.

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.

2 TABLE OWNER

Is the owner 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. If you specify this parameter, you must also specify a qualified name in the SOURCE TABLE NAME field. The value of the AT LOCATION field becomes a prefix for the table name on the SQL DECLARE statement, as follows:

location_name.owner_id.table_name

For example, for the location PLAINS_GA:

PLAINS_GA.CARTER.CROP_YIELD_89

The default is the local location name. This field applies to DB2 private protocol access only. The location you name must be another DB2 UDB for z/OS.

4 DATA SET NAME

Is the name of the data set you allocated to contain the declarations that DCLGEN produces. You must supply a name; no default exists.

The data set must already exist, be accessible to DCLGEN, and 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 knows what the host language is from the DB2I defaults panel.

For example, for library name LIBNAME(MEMBNAM), the name becomes:

userid.libname.language(membnam)

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 in the DATA SET NAME field, if the data set is password protected. It is not displayed on your terminal, and it is not recognized if you issued it from a previous session.

6 ACTION

Tells DCLGEN what 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.)

ADD indicates that an old version of the output does not exist and creates a new member with the specified data set name. This 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

Tells DCLGEN whether to include labels that are declared on any columns of the table or view as comments in the data declarations. (The SQL statement LABEL ON creates column labels to use as supplements to column names.) Use:

YES to include column labels.

NO to ignore column labels. This 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, then 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 you enter in this field do not fold 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.

DCLGEN accepts 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, apostrophes must enclose the prefix. If you use special characters, 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 you enter in this field do not fold 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

Tells DCLGEN whether to delimit DBCS table names and column names in the table declaration. Use:

- YES to enclose the DBCS table and column names with SQL delimiters.
- NO to not delimit the DBCS table and column names.

11 COLUMN SUFFIX

Tells DCLGEN whether to form field names by attaching the column name as a suffix to the value you specify in FIELD NAME PREFIX. 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.

The default is NO, which does not use the column name as a suffix and allows the value in FIELD NAME PREFIX to control the field names, if specified.

12 INDICATOR VARS

Tells DCLGEN whether 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:

For a C program: `short int Itable-name[n];`

For a COBOL program: `01 Itable-name PIC S9(4) USAGE COMP OCCURS n TIMES.`

For a PL/I program: `DCL Itable-name(n) BIN FIXED(15);`

n is the number of columns in the table. For example, suppose you define the following table:

```
CREATE TABLE HASNULLS (CHARCOL1 CHAR(1), CHARCOL2 CHAR(1));
```

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.
```

The default is NO, which does not generate an indicator variable array.

13 ADDITIONAL OPTIONS

Indicates whether to display the panel for additional DCLGEN options. The default is YES.

If you specified YES in the ADDITIONAL OPTIONS field, the ADDITIONAL DCLGEN OPTIONS panel is displayed, as shown in Figure 24.

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 24. ADDITIONAL DCLGEN OPTIONS panel

Fill in the ADDITIONAL DCLGEN OPTIONS panel as follows:

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 of 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 using the SQLCCSID option of the integrated DB2 coprocessor.

The choices are YES and NO. The default is NO.

If the table or view does not have FOR BIT DATA columns, DCLGEN does not generate this statement.

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.

If you are using an SQL reserved word as an identifier, you must edit the DCLGEN output in order to add the appropriate SQL delimiters.

DCLGEN produces output that is intended to meet the needs of most users, but occasionally, you will 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.

Including the data declarations in your program

Use the following SQL INCLUDE statement to place the generated table declaration and COBOL record description in your source program:

```
EXEC SQL
    INCLUDE member-name
END-EXEC.
```

For example, to include a description for the table DSN8810.EMP, code:

```
EXEC SQL
    INCLUDE DECEMP
END-EXEC.
```

In this example, DECEMP is a name of a member of a partitioned data set that contains the table declaration and a corresponding COBOL record description of the table DSN8810.EMP. (A COBOL record description is a two-level host structure that corresponds to the columns of a table's row. For information on host structures, see Chapter 9, "Embedding SQL statements in host languages," on page 143.) To get a current description of the table, use DCLGEN to generate the table's declaration and store it as member DECEMP in a library (usually a partitioned data set) just before you precompile the program.

DCLGEN support of C, COBOL, and PL/I languages

DCLGEN derives variable names from the source in the database. Table 10 lists the type declarations that DCLGEN produces for C, COBOL, and PL/I based on the corresponding SQL data types that are contained in the source tables. In Table 10, *var* represents variable names that DCLGEN provides.

Table 10. Declarations generated by DCLGEN

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)

Table 10. Declarations generated by DCLGEN (continued)

SQL data type ¹	C	COBOL	PL/I
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)
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)
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) ⁵	SQL TYPE IS DBCLOB_LOCATOR	USAGE SQL TYPE IS DBCLOB-LOCATOR	SQL TYPE IS DBCLOB_LOCATOR
BLOB(n) ⁵	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)
ROWID	SQL TYPE IS ROWID	USAGE SQL TYPE IS ROWID	SQL TYPE IS ROWID

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 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.

For more details about the DCLGEN subcommand, see Part 3 of *DB2 Command Reference*.

Example: Adding a table declaration and host-variable structure to a library

This example adds an SQL 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 DSN8810.VPHONE.
- The host-variable structure is for COBOL.
- The structure receives the default name DCLVPHONE.

Information that you must enter is in bold-faced type.

Step 1. Specify COBOL as the host language

Select option **D** on the ISPF/PDF menu to display the DB2I Defaults panel.

Specify **IBMCOB** as the application language, as shown in Figure 25, and press Enter.

```
DSNEOP01          DB2I DEFAULTS
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 DB2I JOB STATEMENT: (Optional if your site has a SUBMIT exit)
    ==> //USRT001A JOB (ACCOUNT),'NAME'
    ==> /*
    ==> /*
    ==> /*

PRESS: ENTER to process      END to cancel      HELP for more information
```

Figure 25. DB2I defaults panel—changing the application language

The COBOL Defaults panel is then displayed, as shown in Figure 26. Fill in the COBOL Defaults panel as necessary. Press Enter to save the new defaults, if any, and return to the DB2I Primary Option menu.

```
DSNEOP02          COBOL DEFAULTS
COMMAND ==> _

Change defaults as desired:

1  COBOL STRING DELIMITER ==>          (DEFAULT, ' or ")
2  DBCS SYMBOL FOR DCLGEN ==>          (G/N - Character in PIC clause)
```

Figure 26. The COBOL defaults panel. Shown only if the field APPLICATION LANGUAGE on the DB2I Defaults panel is IBMCOB.

#####

Fill in the fields as shown in Figure 27, and then press Enter.

Figure 27. DCLGEN panel—selecting source table and destination data set

DSNE905I EXECUTION COMPLETE, MEMBER VPHONEC ADDED

DB2 again displays the DCLGEN screen, as shown in Figure 29 on page 140. Press Enter to return to the DB2I Primary Option menu.


```

***** DCLGEN TABLE(DSN8810.VPHONE) ***
***** LIBRARY(SYSADM.TEMP.COBOL(VPHONEC)) ***
***** QUOTE ***
***** ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS ***
EXEC SQL DECLARE DSN8810.VPHONE TABLE
( LASTNAME          VARCHAR(15) NOT NULL,
  FIRSTNAME         VARCHAR(12) NOT NULL,
  MIDDLEINITIAL     CHAR(1) NOT NULL,
  PHONENUMBER       VARCHAR(4) NOT NULL,
  EMPLOYEENUMBER    CHAR(6) NOT NULL,
  DEPTNUMBER        CHAR(3) NOT NULL,
  DEPTNAME          VARCHAR(36) NOT NULL
) END-EXEC.
***** COBOL DECLARATION FOR TABLE DSN8810.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 EMPLOYEENUMBER      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 *****

```

Figure 30. DCLGEN results displayed in edit mode

Chapter 9. Embedding SQL statements in host languages

This chapter provides detailed information about using each of the following languages to write embedded SQL application programs:

- “Coding SQL statements in an assembler application”
- “Coding SQL statements in a C or C++ application” on page 158
- “Coding SQL statements in a COBOL application” on page 186
- “Coding SQL statements in a Fortran application” on page 220
- “Coding SQL statements in a PL/I application” on page 230.
- “Coding SQL statements in a REXX application” on page 249.

For each language, this chapter provides unique instructions or details about:

- Defining the SQL communications area
- Defining SQL descriptor areas
- Embedding SQL statements
- Using host variables
- Declaring host variables
- Declaring host variable arrays for C or C++, COBOL, and PL/I
- Determining equivalent SQL data types
- Determining if SQL and host language data types are compatible
- Using indicator variables or host structures, depending on the language
- Handling SQL error return codes

For information about reading the syntax diagrams in this chapter, see “How to read the syntax diagrams” on page xxii.

For information about writing embedded SQL application programs in Java, see *DB2 Application Programming Guide and Reference for Java*.

Coding SQL statements in an assembler application

This section helps you with the programming techniques that are unique to coding SQL statements within an assembler program.

Defining the SQL communications area

An assembler program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable, declared as a fullword integer
- An SQLSTATE variable, declared as a character string of length 5 (CL5)

Alternatively, you can include an SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define the SQLCODE or SQLSTATE variable or an SQLCA in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to the SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)

When you use the precompiler 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 declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within a BEGIN DECLARE SECTION and END DECLARE SECTION statement in your program declarations.

If you specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in an assembler program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA
```

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:

```
PROGAREA DSECT
      EXEC SQL INCLUDE SQLCA
```

As an alternative, you can create a separate storage area for the SQLCA and provide addressability to that area.

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix D of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:

- 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 ... USING DESCRIPTOR *descriptor-name*
- OPEN ... USING DESCRIPTOR *descriptor-name*
- PREPARE ... INTO *descriptor-name*

Unlike the SQLCA, a program can have more than one SQLDA in a program, and an SQLDA can have any valid name. You can code an SQLDA in an assembler program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA
```

You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the precompiler option TWOPASS. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix E of *DB2 SQL Reference* for a complete description of SQLDA fields.

Embedding SQL statements

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 DSN8810.DEPT          X
      SET MGRNO = :MGRNUM              X
      WHERE DEPTNO = :INTDEPT
```

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.

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: The precompiler option MARGINS allows you 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.

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.

Using host variables

You must explicitly declare each host variable before its first use in an SQL statement if you specify the precompiler option ONEPASS. If you specify the precompiler option TWOPASS, you must declare the host variable before its use in the statement DECLARE CURSOR.

You can precede the assembler statements that define host variables with the statement BEGIN DECLARE SECTION, and follow the assembler statements with the statement END DECLARE SECTION. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

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.

A colon (:) must precede all host variables in an SQL statement.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Declaring host variables

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.

Numeric host variables: Figure 31 shows the syntax for declarations of numeric host variables. The numeric *value* specifies the scale of the packed decimal variable. If *value* does not include a decimal point, the scale is 0.

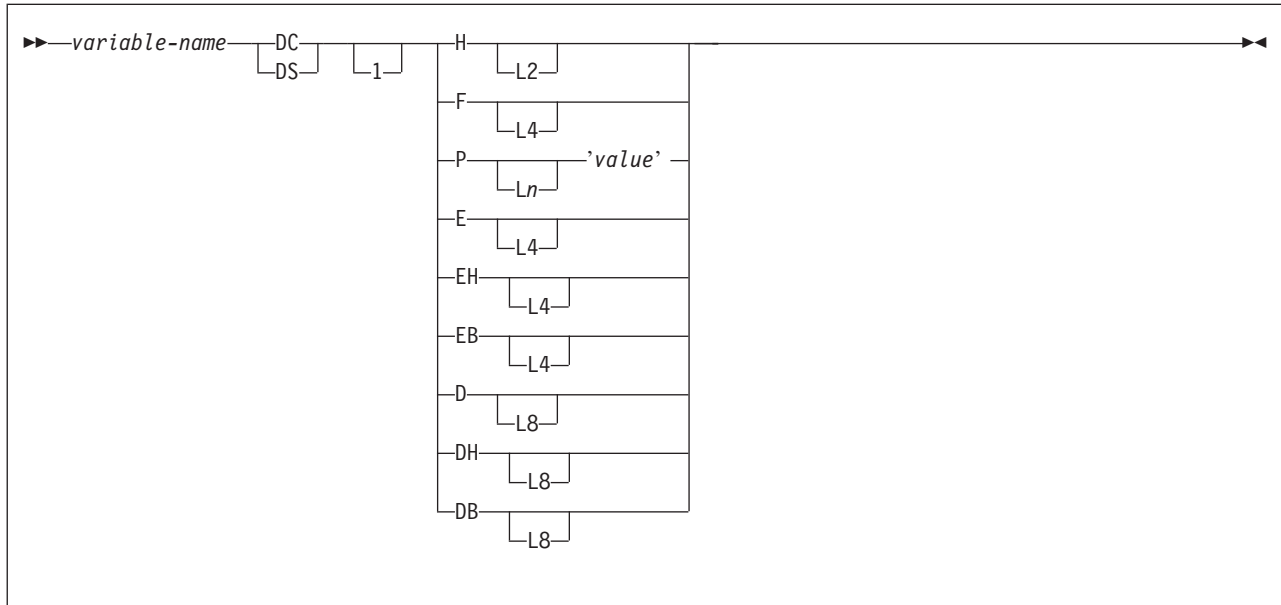


Figure 31. Numeric host variables

For floating-point data types (E, EH, EB, D, DH, and DB), DB2 uses the FLOAT precompiler option to determine whether the host variable is in IEEE binary floating-point or System/390® hexadecimal floating-point format. If the precompiler option is FLOAT(S390), you need to define your floating-point host variables as E, EH, D, or DH. If the precompiler option is FLOAT(IEEE), you need to define your floating-point host variables as EB or DB. DB2 converts all floating-point input data to System/390 hexadecimal floating-point before storing it.

Character host variables: The three valid forms for character host variables are:

- Fixed-length strings
- Varying-length strings
- CLOBs

The following figures show the syntax for forms other than CLOBs. See Figure 38 on page 151 for the syntax of CLOBs.

Figure 32 shows the syntax for declarations of fixed-length character strings.

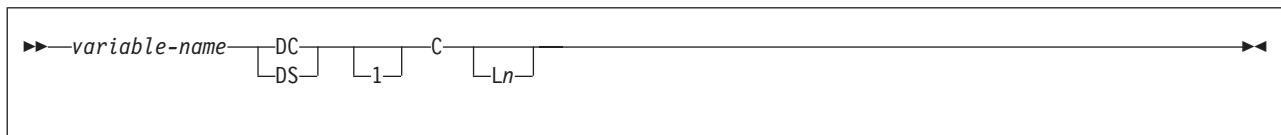


Figure 32. Fixed-length character strings

Figure 33 on page 150 shows the syntax for declarations of varying-length character strings.

Assembler

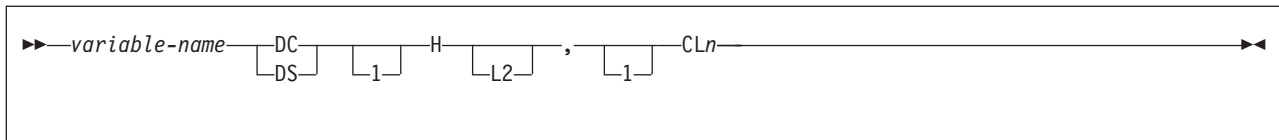


Figure 33. Varying-length character strings

Graphic host variables: The three valid forms for graphic host variables are:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following figures show the syntax for forms other than DBCLOBs. See Figure 38 on page 151 for the syntax of DBCLOBs. In the syntax diagrams, *value* denotes one or more DBCS characters, and the symbols < and > represent shift-out and shift-in characters.

Figure 34 shows the syntax for declarations of fixed-length graphic strings.

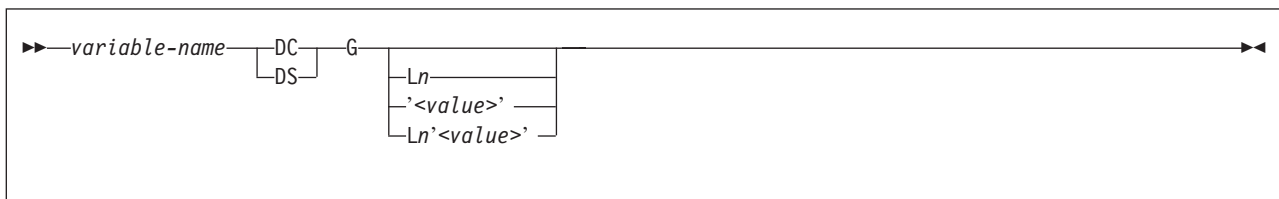


Figure 34. Fixed-length graphic strings

Figure 35 shows the syntax for declarations of varying-length graphic strings.

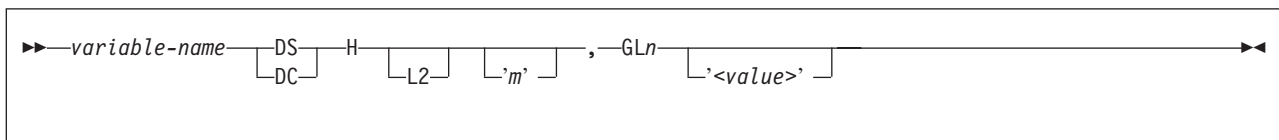


Figure 35. Varying-length graphic strings

Result set locators: Figure 36 shows the syntax for declarations of result set locators. See Chapter 25, “Using stored procedures for client/server processing,” on page 629 for a discussion of how to use these host variables.

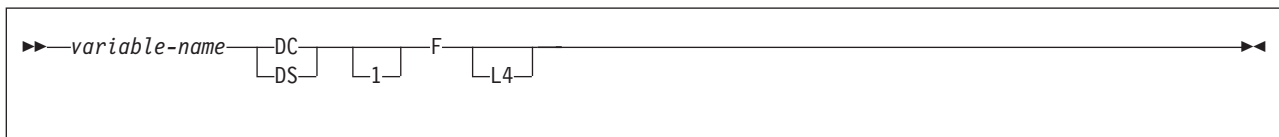


Figure 36. Result set locators

Table Locators: Figure 37 shows the syntax for declarations of table locators. See “Accessing transition tables in a user-defined function or stored procedure” on page 345 for a discussion of how to use these host variables.

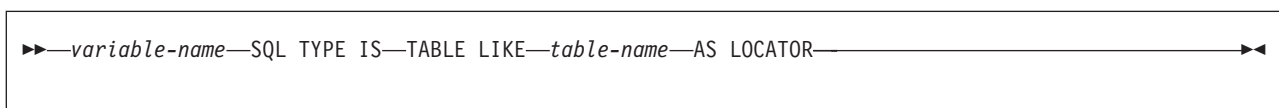


Figure 37. Table locators

LOB variables and locators: Figure 38 shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

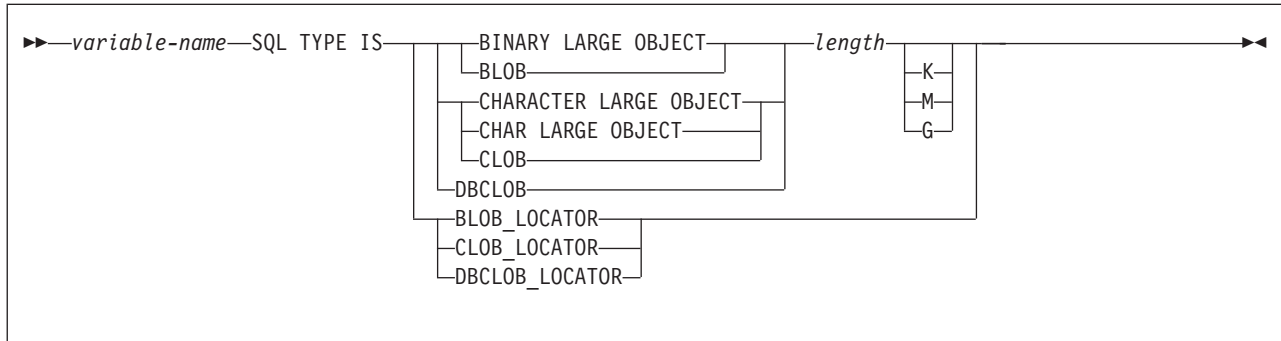


Figure 38. LOB variables and locators

If you specify the length of the LOB in terms of KB, MB, or GB, you must leave no spaces between the length and K, M, or G.

ROWIDs: Figure 39 shows the syntax for declarations of ROWID host variables. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

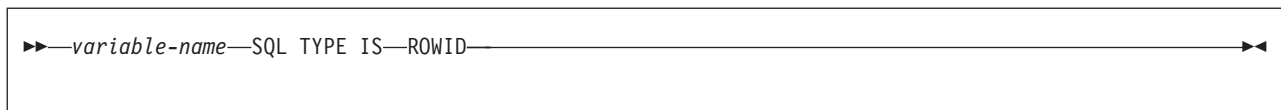


Figure 39. ROWID variables

Determining equivalent SQL and assembler data types

Table 11 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 11. SQL data types the precompiler uses for assembler declarations

Assembler data type	SQLTYPE of host variable	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) See the description for DECIMAL(p,s) in Table 12 on page 153.
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 CLn 1<=n<=255	452	n	CHAR(n)

Table 11. SQL data types the precompiler uses for assembler declarations (continued)

Assembler data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
DS HL2,CLn 1<=n<=255	448	n	VARCHAR(n)
DS HL2,CLn n>255	456	n	VARCHAR(n)
DS GLm 2<=m<=254 ¹	468	n	GRAPHIC(n) ²
DS HL2,GLm 2<=m<=254 ¹	464	n	VARGRAPHIC(n) ²
DS HL2,GLm m>254 ¹	472	n	VARGRAPHIC(n) ²
DS FL4	972	4	Result set locator ²
SQL TYPE IS TABLE LIKE table-name 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 ³
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB locator ³
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 ²	412	n	DBCLOB(n) ²
SQL TYPE IS ROWID	904	40	ROWID

Notes:

1. m is the number of bytes.
2. n is the number of double-byte characters.
3. This data type cannot be used as a column type.

Table 12 on page 153 helps you define host variables that receive output from the database. You can use Table 12 on page 153 to determine the assembler data type that is equivalent to a given SQL data type. For example, if you retrieve `TIMESTAMP` data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 12 on page 153 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information about compatible data types.

Table 12. SQL data types mapped to typical assembler declarations

SQL data type	Assembler equivalent	Notes
SMALLINT	DS HL2	
INTEGER	DS F	
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. $1 \leq p \leq 31$ and $0 \leq s \leq p$. $1 \leq n \leq 16$. <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 $2n-1$; 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 $2n-1=p$, 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	$1 \leq n \leq 21$
DOUBLE PRECISION, DOUBLE, or FLOAT(<i>n</i>)	DS DL8 DS DHL8 DS DBL8 ¹	$22 \leq n \leq 53$
CHAR(<i>n</i>)	DS CL <i>n</i>	$1 \leq n \leq 255$
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. $1 \leq n \leq 127$
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.
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.
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 12. SQL data types mapped to typical assembler declarations (continued)

SQL data type	Assembler equivalent	Notes
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
ROWID	SQL TYPE IS ROWID	

Notes:

1. IEEE floating-point host variables are not supported in user-defined functions and stored procedures.

Notes on assembler variable declaration and usage

You should be aware of the following considerations when you declare assembler variables.

Host graphic data type: You can use the assembler data type “host graphic” in SQL statements when the precompiler option GRAPHIC is in effect. However, you cannot use assembler DBCS literals in SQL statements, even when GRAPHIC is in effect.

Character host variables: If you declare a host variable as a character string without a length, for example DC C 'ABCD', DB2 interprets it as length 1. To get the correct length, give a length attribute (for example, DC CL4'ABCD').

Floating-point host variables: All floating-point data is stored in DB2 in System/390 hexadecimal floating-point format. However, your host variable data can be in System/390 hexadecimal floating-point format or IEEE binary floating-point format. DB2 uses the FLOAT precompiler option to determine whether your floating-point host variables are in IEEE binary floating-point format or System/390 hexadecimal floating-point format. DB2 does no checking to determine whether the host variable declarations or format of the host variable contents match the precompiler option. Therefore, you need to ensure that your floating-point host variable types and contents match the precompiler option.

Special purpose assembler data types: The locator data types are assembler language data types and SQL data types. You cannot use locators as column types. For information about how to use these data types, see the following sections:

Table locator “Accessing transition tables in a user-defined function or stored procedure” on page 345

LOB locators Chapter 14, “Programming for large objects,” on page 297

Overflow: Be careful of overflow. For example, suppose 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 provided an indicator variable.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a host variable 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, it removes any fractional part of the value.

Determining compatibility of SQL and assembler data types

Assembler host variables used in SQL statements must be type compatible with the columns with which you intend to use them.

- Numeric data types are compatible with each other: A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with a numeric assembler host variable.
- Character data types are compatible with each other: A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length assembler 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: A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length assembler 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.

- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length assembler character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB 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. For information about assigning and comparing distinct types, see Chapter 16, “Creating and using distinct types,” on page 367.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using indicator variables

An indicator variable is a 2-byte integer (*DS HL2*). 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 you find in X is irrelevant.

When your program uses 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.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see “Using indicator variables with host variables” on page 83 or Chapter 2 of *DB2 SQL Reference*.

Example: The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,      X
                                :DAY :DAYIND,  X
                                :BGN :BGNIND,  X
                                :END :ENDIND
```

You can declare 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

Figure 40 on page 157 shows the syntax for declarations of indicator host variables.

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 Appendix B, “Sample applications,” on page 1013 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*.

Macros for assembler applications

Data set DSN810.SDSNMACS contains all DB2 macros that are available for use.

Coding SQL statements in a C or C++ application

This section helps you with the programming techniques that are unique to coding SQL statements within a C or C++ program. Throughout this book, C is used to represent either C or C++, except where noted otherwise.

Defining the SQL communication area

A C program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable, declared as long integer. For example:
long SQLCODE;
- An SQLSTATE variable, declared as a character array of length 6. For example:


```
char SQLSTATE[6];
```

Alternatively, you can include an SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define the SQLCODE or SQLSTATE variable or an SQLCA in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)

When you use the precompiler 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 declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the BEGIN DECLARE SECTION and END DECLARE SECTION statements in your program declarations.

If you specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a C program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA;
```

A standard declaration includes both a structure definition and a static data area named 'sqlca'. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix D of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:

- 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 ... USING DESCRIPTOR *descriptor-name*
- OPEN ... USING DESCRIPTOR *descriptor-name*
- PREPARE ... INTO *descriptor-name*

Unlike the SQLCA, more than one SQLDA can exist in a program, and an SQLDA can have any valid name. You can code an SQLDA in a C program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA;
```


A standard declaration includes only a structure definition with the name 'sqllda'. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix E of *DB2 SQL Reference* for a complete description of SQLDA fields.

You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the precompiler option TWOPASS. You can place an SQLDA declaration wherever C allows a structure definition. Normal C scoping rules apply.

Embedding SQL statements

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 64 on page 484.

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 DSN8810.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 cannot nest comments.

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. You can include SQL comments in any embedded SQL statement.

Continuation for SQL statements: You can use a backslash to continue a character-string constant or delimited identifier on the following 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 Chapter 8, "Generating declarations for your tables using DCLGEN," on page 131.

Including code: 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.

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 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. In this chapter, NUL is the null character in C and NULL is the SQL null value.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

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 considerations:

- Using the C/370™ multi-tasking facility, in which multiple tasks execute SQL statements, causes unpredictable results.
- You must run the DB2 precompiler before running the C preprocessor.
- The 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.

Using host variables and host variable arrays

You must explicitly declare each host variable and each host variable array before using them in an SQL statement if you specify the ONEPASS precompiler option. If you use the precompiler option TWOPASS, you must declare each host variable before using it in the DECLARE CURSOR statement.

C

Precede C statements that define the host variables and host variable arrays with the BEGIN DECLARE SECTION statement, and follow the C statements with the END DECLARE SECTION statement. You can have more than one host variable declaration section in your program.

A colon (:) must precede all host variables and all host variable arrays in an SQL statement.

The names of host variables and host variable arrays must be unique within the program, even if the variables and variable arrays are in different blocks, classes, or procedures. You can qualify the names with a structure name to make them unique.

An SQL statement that uses a host variable or host variable array must be within the scope of the statement that declares that variable or array. You define host variable arrays for use with multiple-row FETCH and INSERT statements.

Declaring host variables

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.

Numeric host variables: Figure 41 shows the syntax for declarations of numeric host variables.

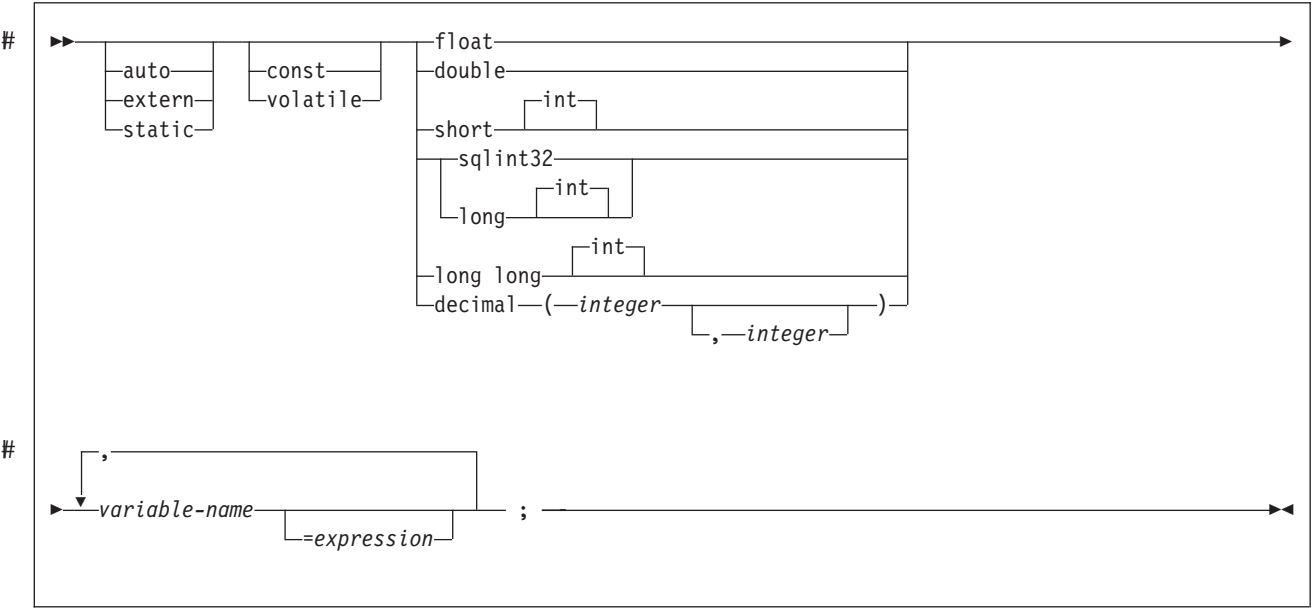


Figure 41. Numeric host variables

Character host variables: The four valid forms for character host variables are:

- Single-character form
- NUL-terminated character form
- VARCHAR structured form
- CLOBs

The following figures show the syntax for forms other than CLOBs. See Figure 50 on page 168 for the syntax of CLOBs.

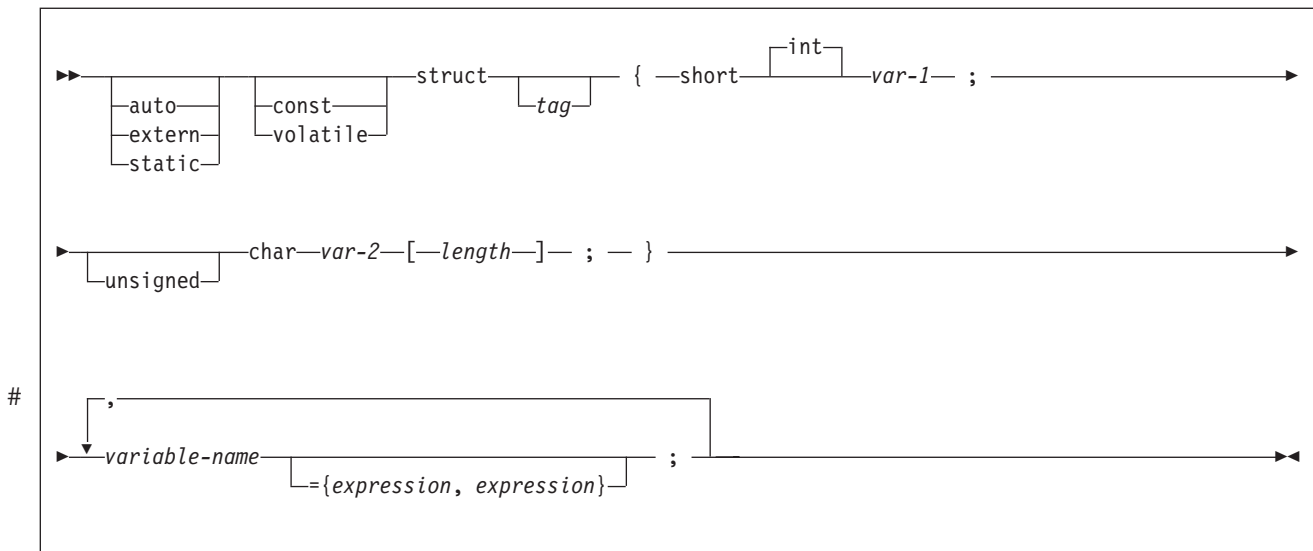


Figure 44. VARCHAR structured form

Notes:

1. *var-1* and *var-2* must be simple variable references. You cannot use them as host variables.
2. You can use the struct tag to define other data areas that you cannot use as host variables.

Example: The following examples show 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;
```

Graphic host variables: The four valid forms for graphic host variables are:

- Single-graphic form
- NUL-terminated graphic form
- VARGRAPHIC structured form.
- DBCLOBs

I You can use the C data type `sqldbcchar` to define a host variable that inserts, updates, deletes, and selects data from GRAPHIC or VARGRAPHIC columns.

The following figures show the syntax for forms other than DBCLOBs. See Figure 50 on page 168 for the syntax of DBCLOBs.

Figure 45 on page 165 shows the syntax for declarations of single-graphic host variables.

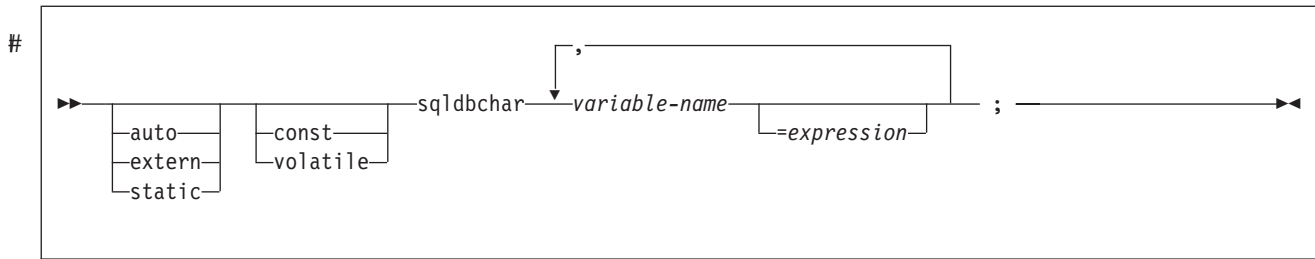


Figure 45. Single-graphic form

The single-graphic form declares a fixed-length graphic string of length 1. You cannot use array notation in *variable-name*.

Figure 46 shows the syntax for declarations of NUL-terminated graphic host variables.

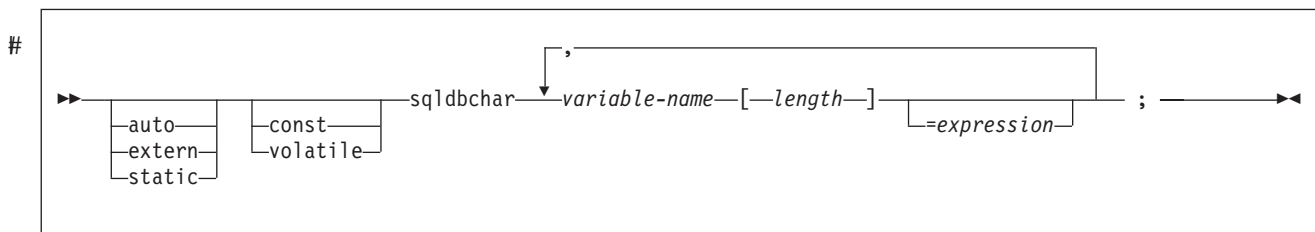


Figure 46. Nul-terminated graphic form

Notes:

1. *length* must be a decimal integer constant greater than 1 and not greater than 16352.
2. On input, the string in *variable-name* must be NUL-terminated.
3. On output, the string is NUL-terminated.
4. The NUL-terminated graphic form does not accept single-byte characters into *variable-name*.

Figure 47 on page 166 shows the syntax for declarations of graphic host variables that use the VARGRAPHIC structured form.

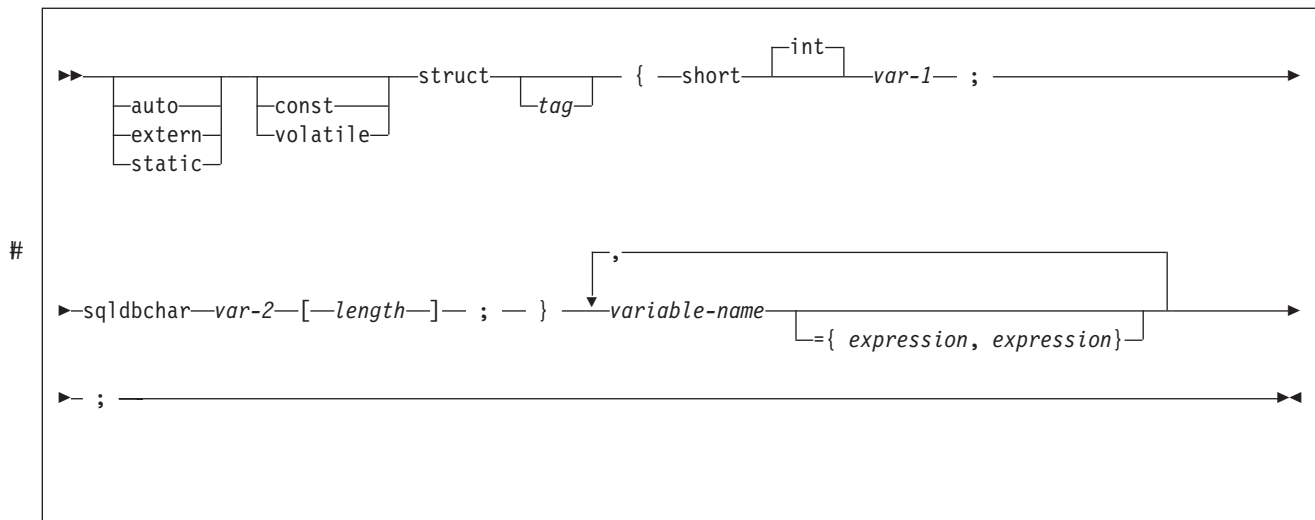


Figure 47. VARGRAPHIC structured form

Notes:

1. *length* must be a decimal integer constant greater than 1 and not greater than 16352.
2. *var-1* must be less than or equal to *length*.
3. *var-1* and *var-2* must be simple variable references. You cannot use them as host variables.
4. You can use the struct tag to define other data areas that you cannot use as host variables.

Example: The following examples show 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;
  sqlblobchar d[10];
} vgraph;

```

```

/* invalid declaration of host variable structured wgraph */
struct VARGRAPH wgraph;

```

Result set locators: Figure 48 on page 167 shows the syntax for declarations of result set locators. See Chapter 25, “Using stored procedures for client/server processing,” on page 629 for a discussion of how to use these host variables.

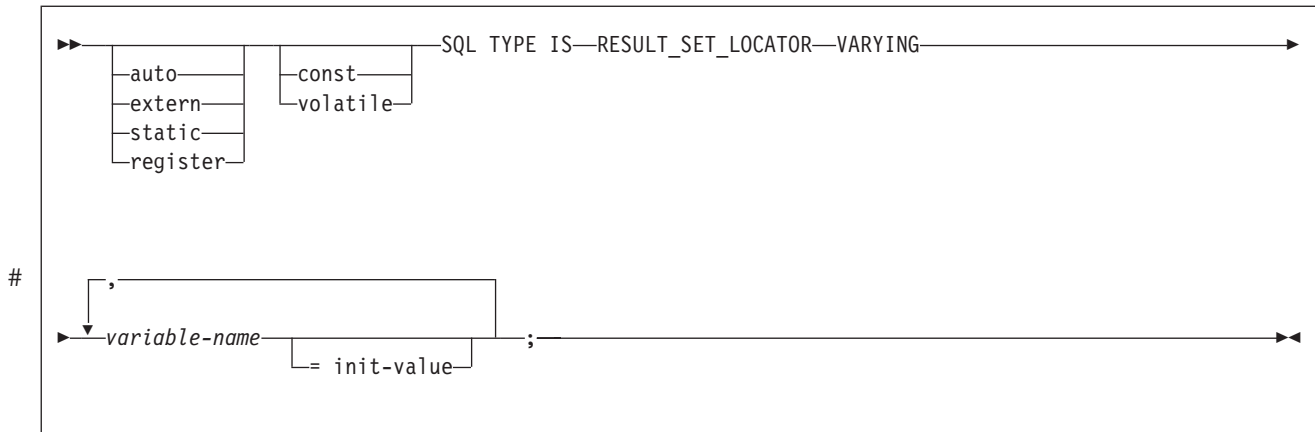


Figure 48. Result set locators

Table Locators: Figure 49 shows the syntax for declarations of table locators. See “Accessing transition tables in a user-defined function or stored procedure” on page 345 for a discussion of how to use these host variables.

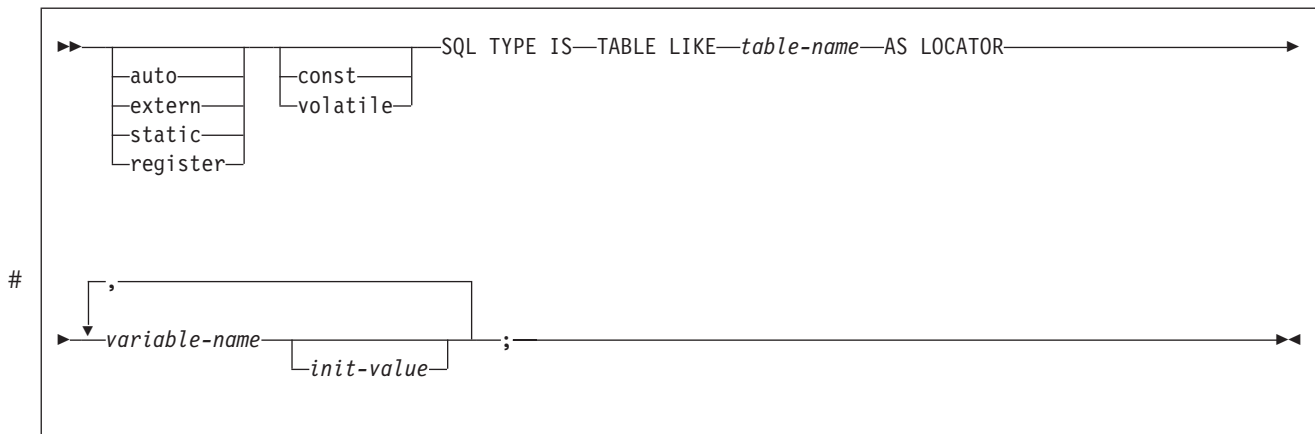


Figure 49. Table locators

LOB Variables and Locators: Figure 50 on page 168 shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

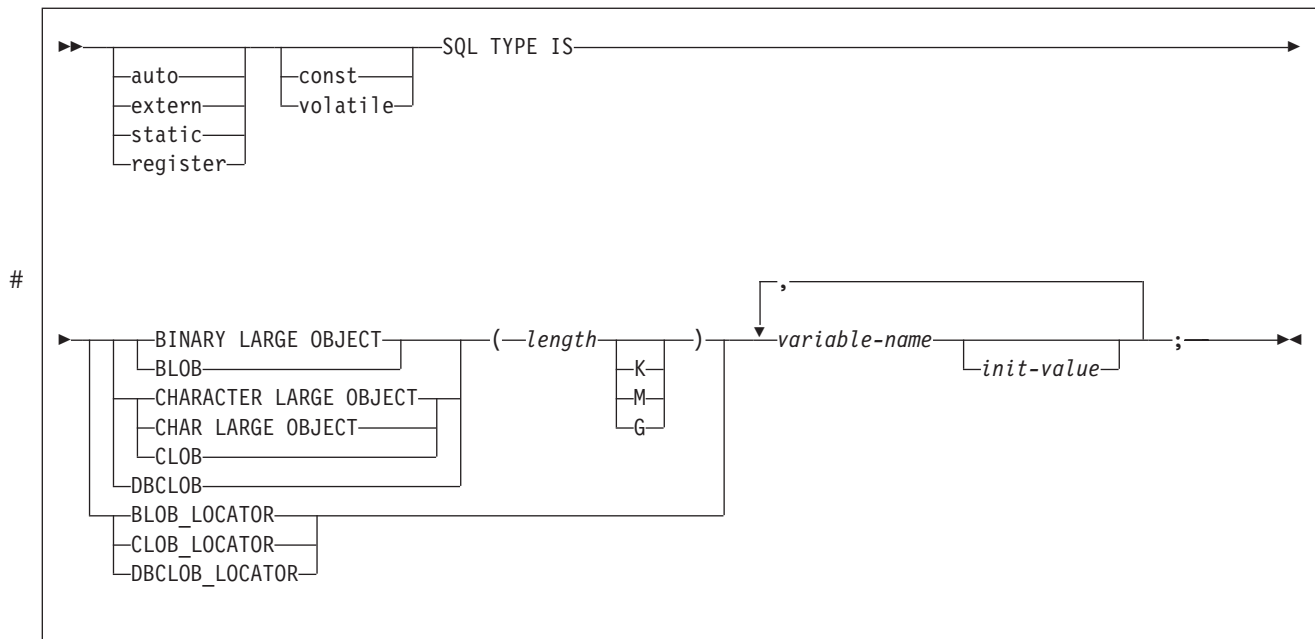


Figure 50. LOB variables and locators

ROWIDs: Figure 51 shows the syntax for declarations of ROWID host variables. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

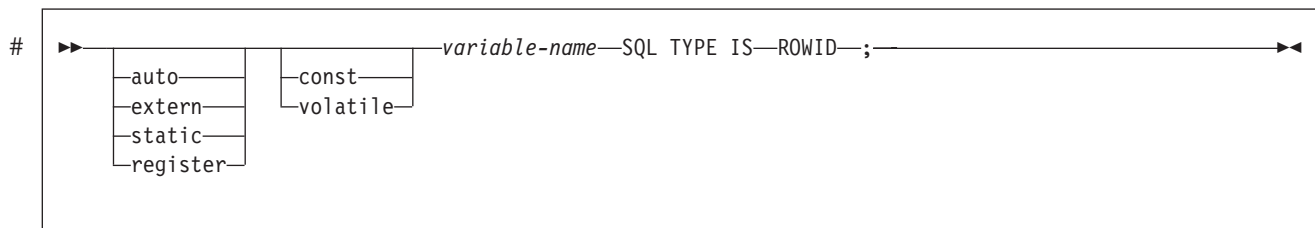


Figure 51. ROWID variables

Declaring host variable arrays

Only some of the valid C declarations are valid host variable array declarations. If the declaration for a variable array is not valid, then 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 varying-length character arrays, varying-length graphic arrays, or LOB arrays that are to be used in multiple-row INSERT and FETCH statements. 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: Figure 52 on page 169 shows the syntax for declarations of numeric host variable arrays.

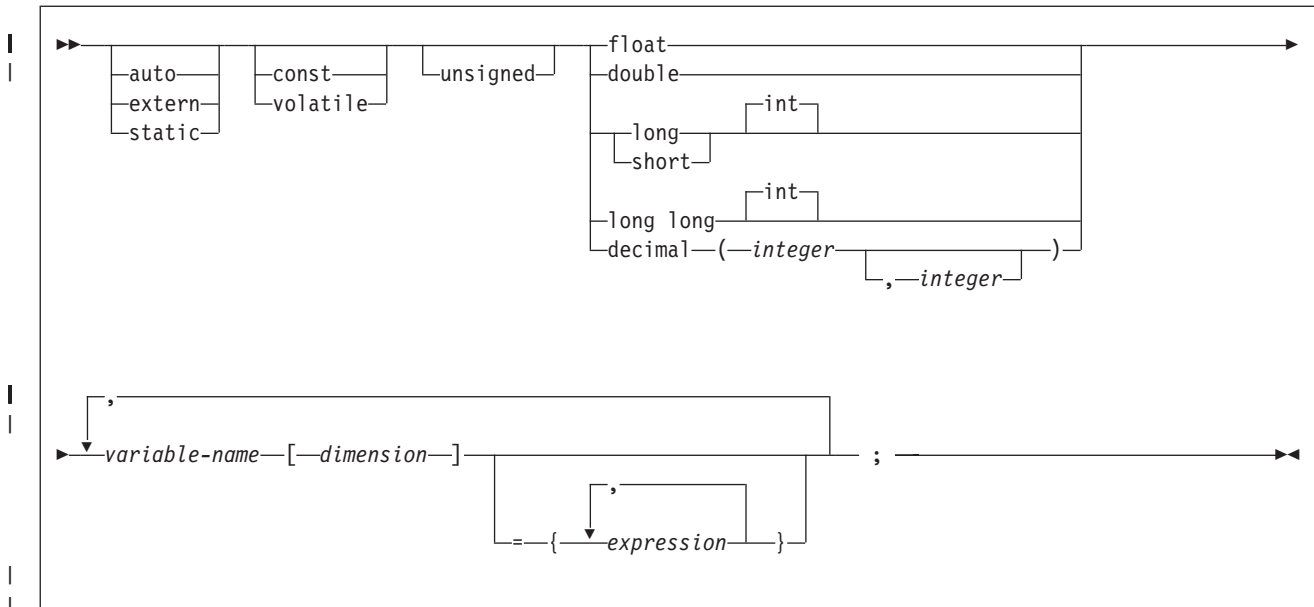


Figure 52. Numeric host variable arrays

Note:

1. *dimension* must be an integer constant between 1 and 32767.

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: The three valid forms for character host variable arrays are:

- NUL-terminated character form
- VARCHAR structured form
- CLOBs

The following figures show the syntax for forms other than CLOBs. See Figure 57 on page 173 for the syntax of CLOBs.

Figure 53 on page 170 shows the syntax for declarations of NUL-terminated character host variable arrays.

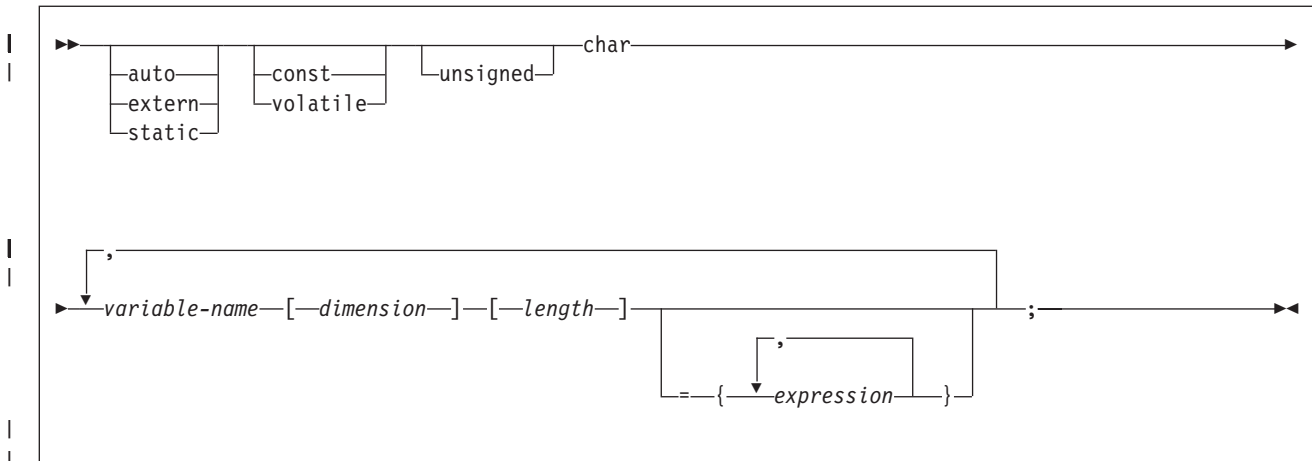


Figure 53. NUL-terminated character form

Notes:

1. On input, the strings contained in the variable arrays must be NUL-terminated.
2. On output, the strings are NUL-terminated.
3. The strings in a NUL-terminated character host variable array map to varying-length character strings (except for the NUL).
4. *dimension* must be an integer constant between 1 and 32767.

Figure 54 shows the syntax for declarations of varying-length character host variable arrays that use the VARCHAR structured form.

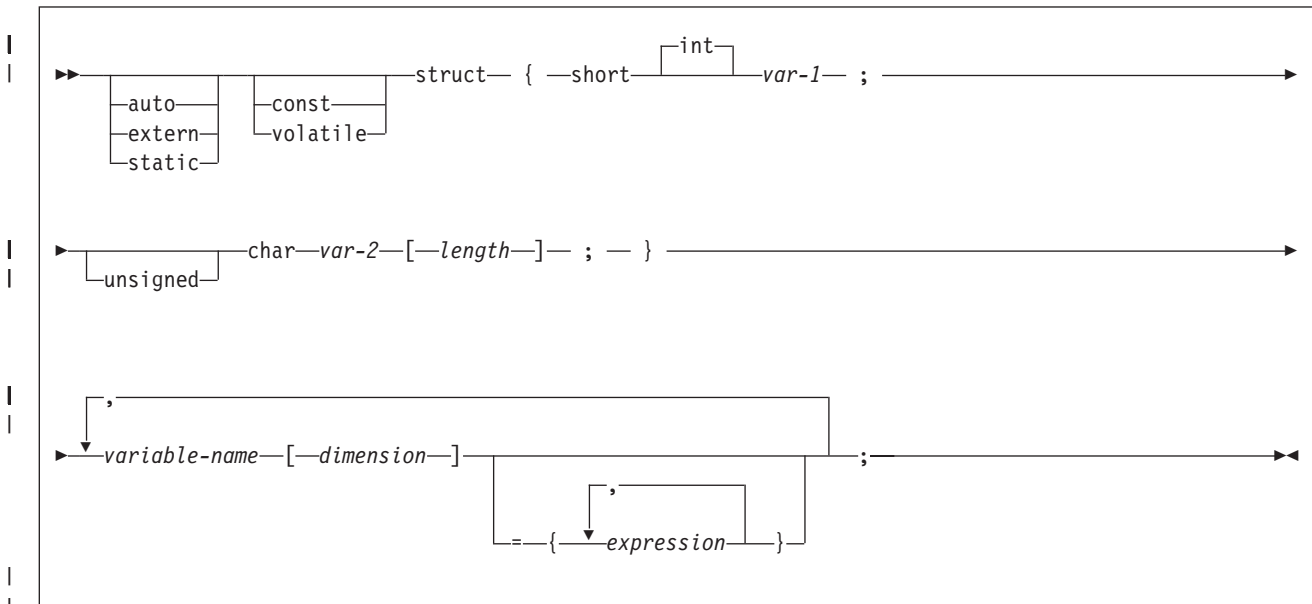


Figure 54. VARCHAR structured form

Notes:

1. *var-1* must be a simple variable reference, and *var-2* must be a variable array reference.
2. You can use the struct tag to define other data areas, which you cannot use as host variable arrays.
3. *dimension* must be an integer constant between 1 and 32767.

Example: The following examples show 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];
```

Graphic host variable arrays: The two valid forms for graphic host variable arrays are:

- NUL-terminated graphic form
- VARGRAPHIC structured form.

You can use the C data type `sqldbchar` to define a host variable array that inserts, updates, deletes, and selects data from GRAPHIC or VARGRAPHIC columns.

Figure 55 shows the syntax for declarations of NUL-terminated graphic host variable arrays.

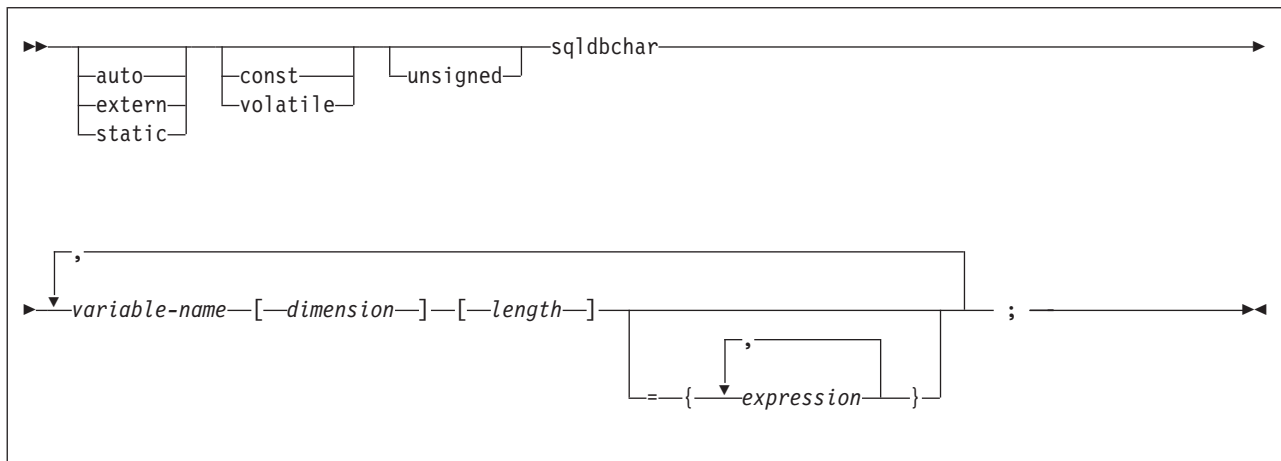


Figure 55. NUL-terminated graphic form

Notes:

1. *length* must be a decimal integer constant greater than 1 and not greater than 16352.
2. On input, the strings contained in the variable arrays must be NUL-terminated.
3. On output, the string is NUL-terminated.
4. The NUL-terminated graphic form does not accept single-byte characters into the variable array.
5. *dimension* must be an integer constant between 1 and 32767.

Figure 56 on page 172 shows the syntax for declarations of graphic host variable arrays that use the VARGRAPHIC structured form.

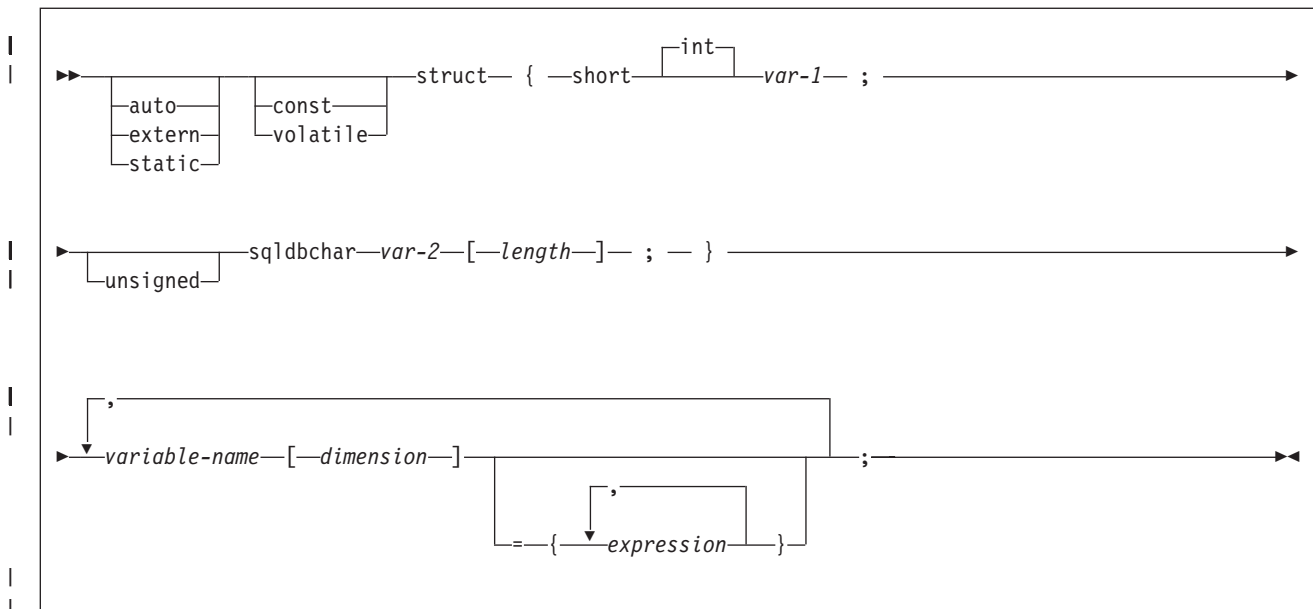


Figure 56. VARGRAPHIC structured form

Notes:

1. *length* must be a decimal integer constant greater than 1 and not greater than 16352.
2. *var-1* must be a simple variable reference, and *var-2* must be a variable array reference.
3. You can use the struct tag to define other data areas, which you cannot use as host variable arrays.
4. *dimension* must be an integer constant between 1 and 32767.

Example: The following examples show 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;
    sqldbchar d[10];
} vgraph[20];

/* invalid declaration of host variable array vgraph */
struct VARGRAPH vgraph[20];
  
```

LOB variable arrays and locators: Figure 57 on page 173 shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variable arrays and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use LOB variables.

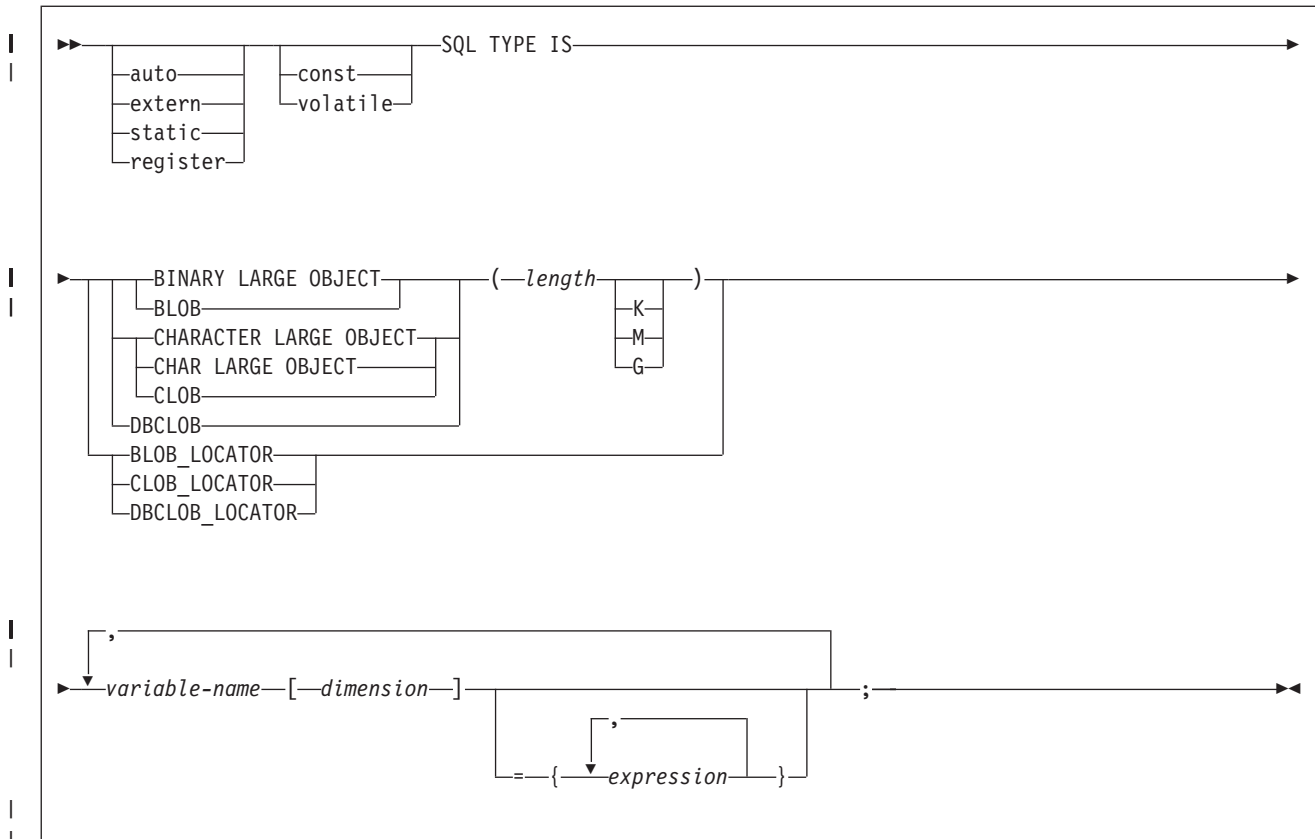


Figure 57. LOB variable arrays and locators

Note:

1. *dimension* must be an integer constant between 1 and 32767.

ROWIDs: Figure 58 shows the syntax for declarations of ROWID variable arrays. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variable arrays.

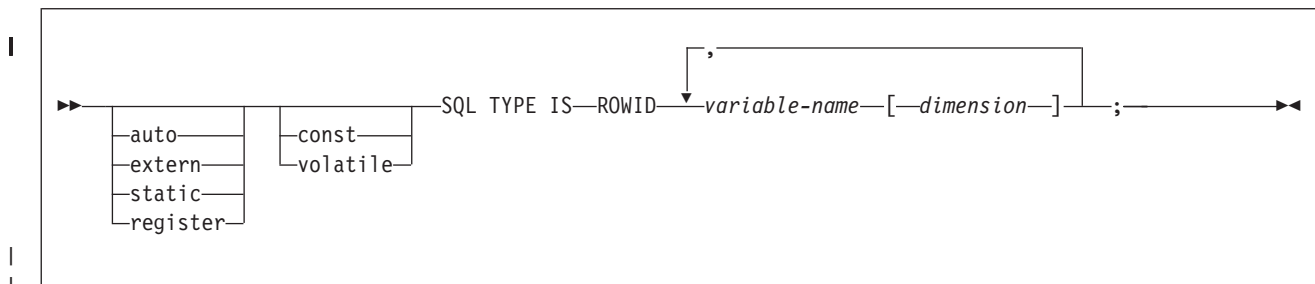


Figure 58. ROWID variable arrays

Note:

1. *dimension* must be an integer constant between 1 and 32767.

Using host structures

A C host structure contains an ordered group of data fields. For example:

C

```
struct {char c1[3];
       struct {short len;
               char data[5];
               }c2;
       char c3[2];
       }target;
```

In this example, *target* is the name of a host structure consisting of the *c1*, *c2*, and *c3* fields. *c1* and *c3* are character arrays, and *c2* is the host variable equivalent to the 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.

Figure 59 shows the syntax for declarations of host structures.

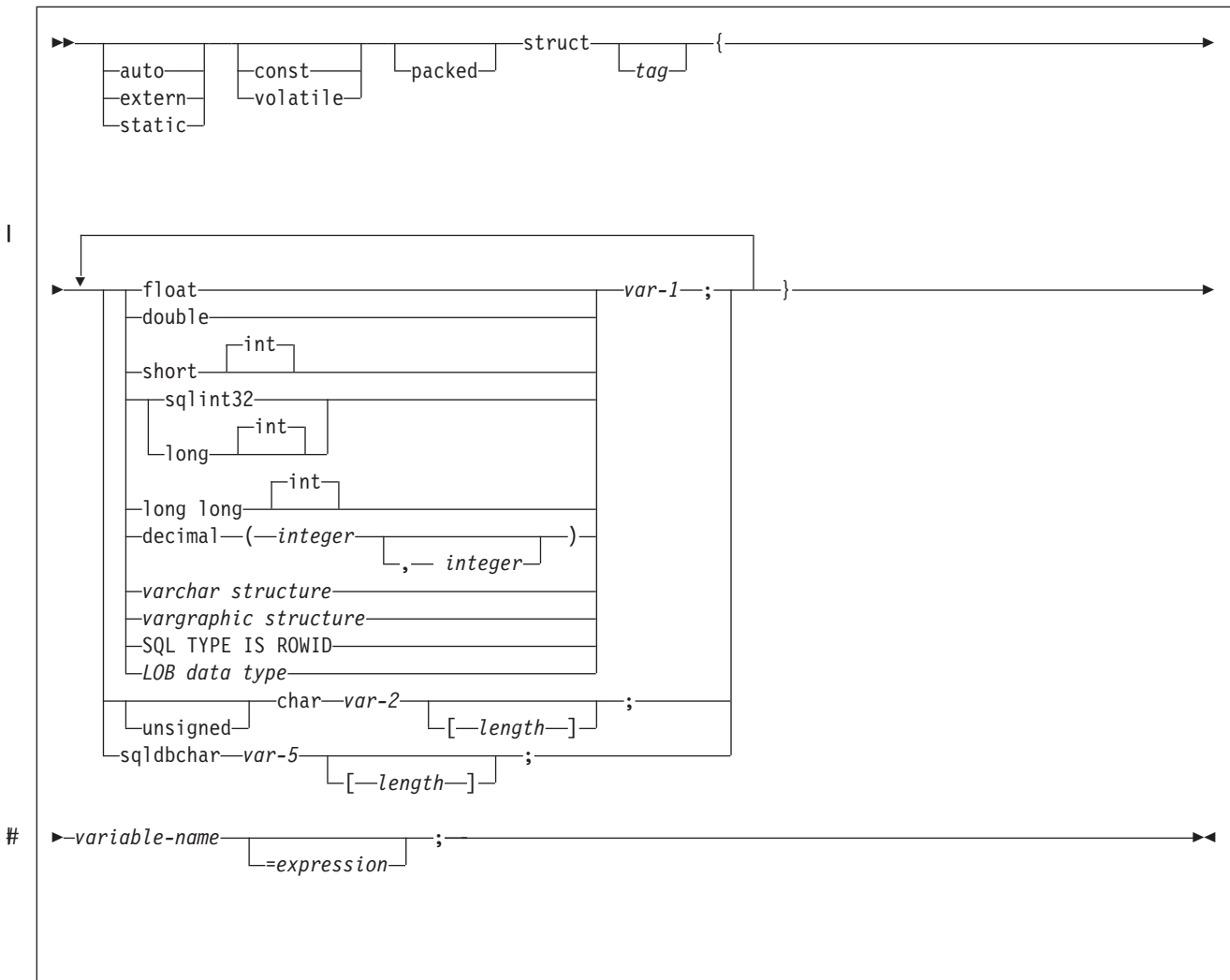


Figure 59. Host structures

Figure 60 on page 175 shows the syntax for VARCHAR structures that are used within declarations of host structures.

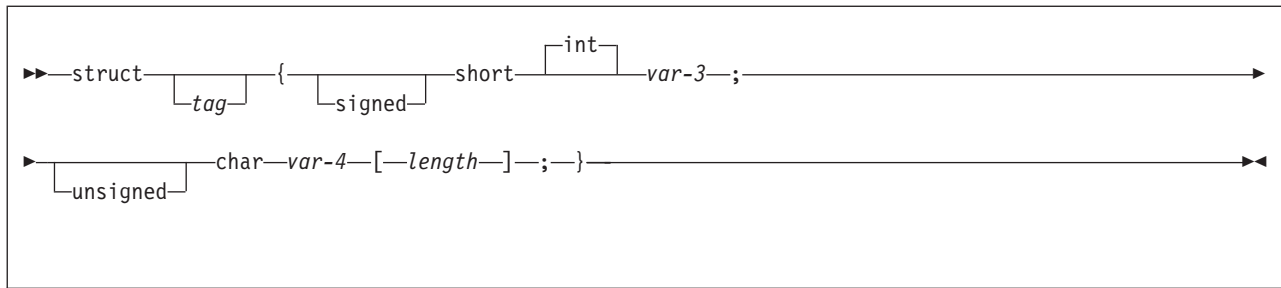


Figure 60. VARCHAR-structure

Figure 61 shows the syntax for VARGRAPHIC structures that are used within declarations of host structures.

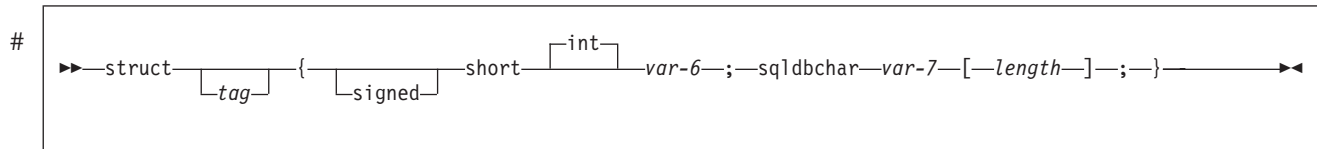


Figure 61. VARGRAPHIC-structure

Figure 62 shows the syntax for LOB data types that are used within declarations of host structures.

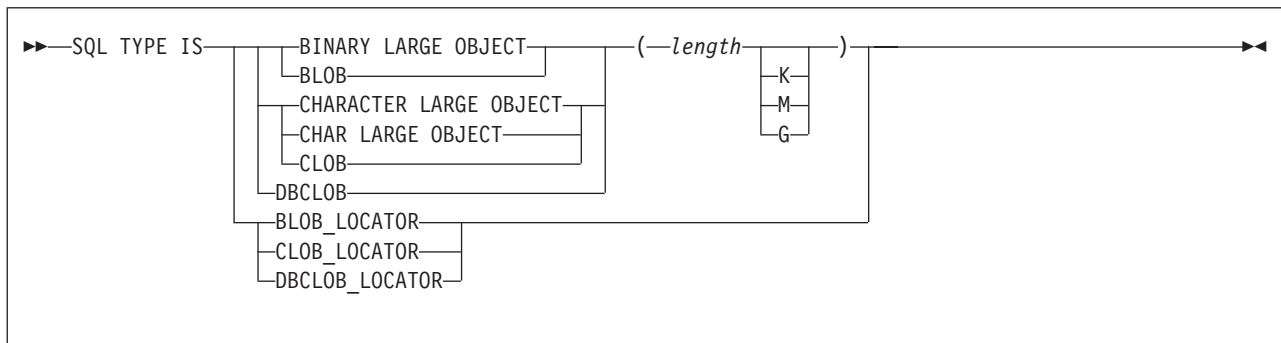


Figure 62. LOB data type

Determining equivalent SQL and C data types

Table 13 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 13. SQL data types the precompiler uses for C declarations

C data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
short int	500	2	SMALLINT
long int	496	4	INTEGER
long long	484	19 in byte 1, 0 in byte 2 ⁴	
decimal(p,s) ¹	484	p in byte 1, s in byte 2	DECIMAL(p,s) ¹

Table 13. SQL data types the precompiler uses for C declarations (continued)

C data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
float	480	4	FLOAT (single precision)
double	480	8	FLOAT (double precision)
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 (sqldbchar)	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 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
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>) ³
SQL TYPE IS ROWID	904	40	ROWID

Table 13. SQL data types the precompiler uses for C declarations (continued)

C data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type

Notes:

1. p is the *precision*; in SQL terminology, this is the total number of digits. In C, this is called the *size*.
 s is the *scale*; in SQL terminology, this is the number of digits to the right of the decimal point. In C, this is called the *precision*.
C++ does not support the decimal data type.
2. Do not use this data type as a column type.
3. n is the number of double-byte characters.
4. No exact equivalent. Use DECIMAL(19,0).

Table 14 helps you define host variables that receive output from the database. You can use the table to determine the C data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 14 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information about compatible data types.

Table 14. SQL data types mapped to typical C declarations

SQL data type	C data type	Notes
SMALLINT	short int	
INTEGER	long int	
DECIMAL(p,s) or NUMERIC(p,s)	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(n)	float	$1 \leq n \leq 21$
DOUBLE PRECISION or FLOAT(n)	double	$22 \leq n \leq 53$
CHAR(1)	single-character form	
CHAR(n)	no exact equivalent	If $n > 1$, use NUL-terminated character form
VARCHAR(n)	NUL-terminated character form	If data can contain character NULs (\0), use VARCHAR structured form. Allow at least $n+1$ to accommodate the NUL-terminator.
	VARCHAR structured form	
GRAPHIC(1)	single-graphic form	
GRAPHIC(n)	no exact equivalent	If $n > 1$, use NUL-terminated graphic form. n is the number of double-byte characters.

Table 14. SQL data types mapped to typical C declarations (continued)

SQL data type	C data type	Notes
VARGRAPHIC(<i>n</i>)	NUL-terminated graphic form	If data can contain graphic NUL values (<code>\0\0</code>), 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.
	VARGRAPHIC structured form	<i>n</i> is the number of double-byte characters.
DATE	NUL-terminated character 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.
	VARCHAR structured form	If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 10 characters.
TIME	NUL-terminated character 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.
	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 6; to include seconds, the length must be at least 8.
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.
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

Table 14. SQL data types mapped to typical C declarations (continued)

SQL data type	C data type	Notes
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
ROWID	SQL TYPE IS ROWID	

Notes on C variable declaration and usage

You should be aware of the following considerations when you declare C variables.

C data types with no SQL equivalent: C supports some data types and storage classes with no SQL equivalents, for example, register storage class, typedef, and long long, .

SQL data types with no C equivalent: If your C compiler does not have a decimal data type, no exact equivalent exists for the SQL DECIMAL data type. In this case, to hold the value of such a variable, you can use:

- An integer or floating-point variable, which converts the value. If you choose integer, you will 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, you can use floating-point numbers. 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 in this 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;
```

Floating-point host variables: All floating-point data is stored in DB2 in System/390 hexadecimal floating-point format. However, your host variable data can be in System/390 hexadecimal floating-point format or IEEE binary floating-point format. DB2 uses the FLOAT precompiler option to determine whether your floating-point host variables are in IEEE binary floating-point or System/390 hexadecimal floating-point format. DB2 does no checking to determine whether the contents of a host variable match the precompiler option. Therefore, you need to ensure that your floating-point data format matches the precompiler option.

Graphic host variables in user-defined function: The SQLUDF file, which is in data set DSN810.SDSNC.H, contains many data declarations for C language user-defined functions. SQLUDF contains the typedef sqldbchar, which you should use instead of wchar_t. Using sqldbchar lets you 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 DB2 platforms.

Special Purpose C Data Types: The locator data types are C data types and SQL data types. You cannot use locators as column types. For information about how to use these data types, see the following sections:

Result set locator

Chapter 25, "Using stored procedures for client/server processing," on page 629

Table locator "Accessing transition tables in a user-defined function or stored procedure" on page 345

LOB locators Chapter 14, "Programming for large objects," on page 297

String host variables: If you assign a string of length n to a NUL-terminated variable with a length that is:

- Less than or equal to n , DB2 inserts the characters into the host variable up to a length of $(n-1)$, and appends a NUL at the end of the string. DB2 sets SQLWARN[1] to W and any indicator variable you provide to the original length of the source string.
- Equal to $n+1$, DB2 inserts the characters into the host variable and appends a NUL at the end of the string.
- Greater than $n+1$, the rules depend on whether the source string is a value of a fixed-length string column or a varying-length string column. If the source is a fixed-length string, DB2 pads it with blanks on assignment to the NUL-terminated variable depending on whether the precompiler option PADNTSTR is specified. If the source is a varying-length string, DB2 assigns it to the first n bytes of the variable and appends a NUL at the end of the string. For information about host language precompiler options, see Table 64 on page 484.

PREPARE or DESCRIBE statements: You cannot use a host variable that is of the NUL-terminated form in either a PREPARE or DESCRIBE statement when you use the DB2 precompiler. However, if you use the DB2 coprocessor for either C or C++, you can use host variables of the NUL-terminated form in PREPARE, DESCRIBE, and EXECUTE IMMEDIATE statements.

L-literals: DB2 tolerates L-literals in C application programs. DB2 allows properly formed L-literals, although it does not check for all the restrictions that the C compiler imposes on the L-literal. You can use DB2 graphic string constants in SQL statements to work with the L-literal. Do not use L-literals in SQL statements.

Overflow: Be careful of overflow. For example, suppose 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.

Truncation: Be careful of truncation. Ensure that the host variable 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.

Notes on syntax differences for constants

You should be aware of the following syntax differences for constants.

Decimal constants versus real constants: In C, a string of digits with a decimal point is interpreted as a real constant. In an SQL statement, such a string is interpreted as a decimal constant. You must use exponential notation when specifying a real (that is, floating-point) constant in an SQL statement.

In C, a real (floating-point) constant can have a suffix of *f* or *F* to show a data type of *float* or a suffix of *l* or *L* to show a type of *long double*. A floating-point constant in an SQL statement must not use these suffixes.

Integer constants: In C, you can provide integer constants in hexadecimal form if the first two characters are *0x* or *0X*. You cannot use this form in an SQL statement.

In C, an integer constant can have a suffix of *u* or *U* to show that it is an unsigned integer. An integer constant can have a suffix of *l* or *L* to show a long integer. You cannot use these suffixes in SQL statements.

Character and string constants: In C, character constants and string constants can use escape sequences. You cannot use the escape sequences in SQL statements. Apostrophes and quotes have different meanings in C and SQL. In C, you can use double quotes to delimit string constants, and apostrophes to delimit character constants. The following examples illustrate the use of quotes and apostrophes in C.

Quotes

```
printf( "%d lines read. \n", num_lines);
```

Apostrophes

```
#define NUL '\0'
```

In SQL, you can use double quotes to delimit identifiers and apostrophes to delimit string constants. The following examples illustrate the use of apostrophes and quotes in SQL.

Quotes

```
SELECT "COL#1" FROM TBL1;
```

Apostrophes

```
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.

Determining compatibility of SQL and C data types

C host variables used in SQL statements must be type compatible with the columns with which you intend to use them:

- Numeric data types are compatible with each other. A *SMALLINT*, *INTEGER*, *DECIMAL*, or *FLOAT* column is compatible with any C host variable that is defined as type short int, long int, decimal, float, or double.
- Character data types are compatible with each other. A *CHAR*, *VARCHAR*, or *CLOB* column is compatible with a single-character, NUL-terminated, or *VARCHAR* structured form of a C 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. A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a single character, NUL-terminated, or VARGRAPHIC structured form of a C graphic 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.

- Datetime data types are compatible with character host variable. A DATE, TIME, or TIMESTAMP column is compatible with a single-character, NUL-terminated, or VARCHAR structured form of a C character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB 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. For information about assigning and comparing distinct types, see Chapter 16, “Creating and using distinct types,” on page 367.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Varying-length strings: For varying-length BIT data, use the VARCHAR structured form. 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.

Using indicator variables and indicator variable arrays

An indicator variable is a 2-byte integer (short int). An indicator variable array is an array of 2-byte integers (short int). You use indicator variables and indicator variable arrays in similar ways.

Using indicator variables: 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 you find in X is irrelevant.

When your program uses 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. For more information about indicator variables, see “Using indicator variables with host variables” on page 83.

Using indicator variable arrays: When you retrieve data into a host variable array, if a value in its indicator array is negative, you can disregard the contents of the corresponding element in the host variable array. For more information about indicator variable arrays, see “Using indicator variable arrays with host variable arrays” on page 87.

Declaring indicator variables: You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate.

Example: The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
                                :Day :DayInd,
                                :Bgn :BgnInd,
                                :End :EndInd;
```

You can declare 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;
```

Figure 63 shows the syntax for declarations of an indicator variable.

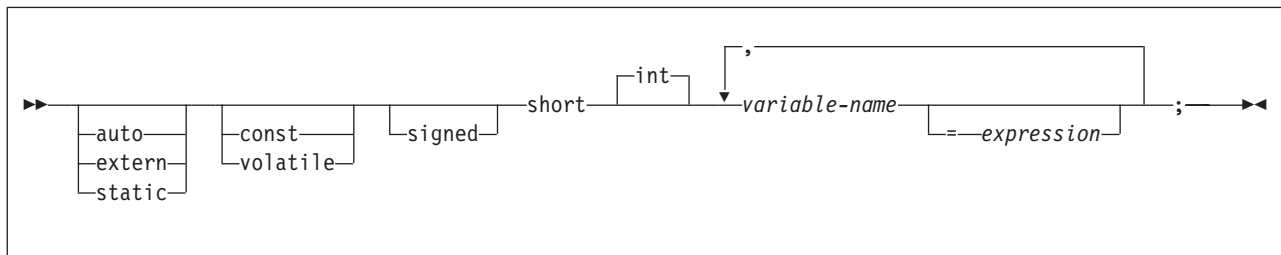


Figure 63. Indicator variable

Declaring indicator variable arrays: Figure 64 on page 184 shows the syntax for declarations of an indicator array or a host structure indicator array.

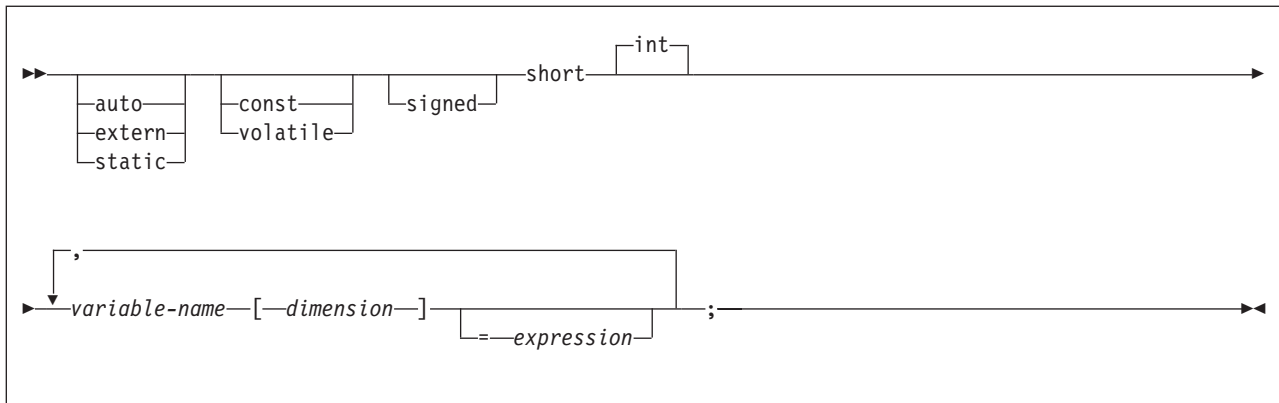


Figure 64. Host structure indicator array

Note: The *dimension* must be an integer constant between 1 and 32767.

Handling SQL error return codes

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 “Calling DSNTIAR to display SQLCA fields” on page 98.

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 “The GET DIAGNOSTICS statement” on page 94.

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
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, &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.

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,05)
```

For C++, include a statement similar to this:

```
extern "05" 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 DSN8810.SDSNSAMP. See Appendix B, "Sample applications," on page 1013 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 DSNTJ5A.

The assembler source code for DSNTIAC and job DSNTJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix*.SDSNSAMP.

Coding considerations for C and C++

Using C++ data types as host variables: When you code SQL statements in a C++ program, you can use class members as host variables. Class members used as host variables are accessible to any SQL statement within the class. However, you cannot use class objects as host variables.

Coding SQL statements in a COBOL application

This section helps you with the programming techniques that are unique to coding SQL statements within a COBOL program.

See *DB2 Program Directory* for a list of the supported versions of COBOL.

Defining the SQL communication area

A COBOL program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable declared as PIC S9(9) BINARY, PIC S9(9) COMP-4, PIC S9(9) COMP-5, or PICTURE S9(9) COMP
- An SQLSTATE variable declared as PICTURE X(5)

Alternatively, you can include an SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define the SQLCODE or SQLSTATE variable or an SQLCA in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)

When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

When you use the precompiler option 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 you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the SQLCODE and SQLSTATE variables within the BEGIN DECLARE SECTION and END DECLARE SECTION statements in your program declarations.

If you specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a COBOL program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

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. You can declare a stand-alone SQLCODE variable in either the WORKING-STORAGE SECTION or LINKAGE SECTION.

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix D of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:

- 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 ... USING 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. The SQL INCLUDE statement does not provide an SQLDA mapping for COBOL. You can define the SQLDA using one of the following two methods:

- For COBOL programs compiled with any compiler, you can code the SQLDA declarations in your program. For more information, see “Using dynamic SQL in COBOL” on page 627. 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.
- For COBOL programs compiled with any compiler, you can call a subroutine (written in C, PL/I, or assembler language) that uses the INCLUDE SQLDA statement to define the SQLDA. The subroutine can also include SQL statements for any dynamic SQL functions you need. For more information on using dynamic SQL, see Chapter 24, “Coding dynamic SQL in application programs,” on page 593.

You must place SQLDA declarations before the first SQL statement that references the data descriptor. An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Embedding SQL statements

You can code SQL statements in the COBOL program sections shown in Table 15.

Table 15. Allowable SQL statements for COBOL program sections

SQL statement	Program section
BEGIN DECLARE SECTION END DECLARE SECTION	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE SQLCA	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE text-file-name	PROCEDURE DIVISION or DATA DIVISION ¹
DECLARE TABLE DECLARE CURSOR	DATA DIVISION or PROCEDURE DIVISION
Other	PROCEDURE DIVISION

Table 15. Allowable SQL statements for COBOL program sections (continued)

SQL statement	Program section
Notes: 1. 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 the SQL statement appears between two COBOL statements, the period 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. 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 a COBOL program as follows:

```
EXEC SQL
    UPDATE DSN8810.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, except between the keywords EXEC and SQL. The precompiler also treats COBOL debugging and page-eject lines (/ in column 7) as comment lines. For an SQL INCLUDE statement, DB2 treats any text that follows the period after END-EXEC, and on the same line as END-EXEC, as a comment.

In addition, you can include SQL comments in any embedded SQL statement.

Debugging lines: The precompiler ignores the 'D' in column 7 on debugging lines and treats it as a blank.

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.

COPY: Do not use a COBOL COPY statement within host variable declarations because the DB2 precompiler will not evaluate the statement.

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. For more information, see Chapter 8, “Generating declarations for your tables using DCLGEN,” on page 131.

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. See “Defining SQL descriptor areas” on page 187 for more information on 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 SQL coprocessor, none of these restrictions apply.

#

Margins: You must code EXEC SQL in columns 12 through 72. Otherwise the DB2 precompiler does not recognize the SQL statement. 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.

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, if you wish.

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 and CAF

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(RRSF) 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 a 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, 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.
- 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 in Chapter 2 of *DB2 SQL Reference* when you name 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 1ST_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 UDB for OS/390 Version 6 or later.
 - The application program accesses remote sites, and the local site and remote sites are DB2 UDB for OS/390 Version 6 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 and COBOL, the following additional restrictions apply:
 - All SQL statements and any host variables they reference must be within the first program when using nested programs or batch compilation.
 - DB2 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.

If your program 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.

Using host variables and host variable arrays

| You must explicitly declare all host variables and host variable arrays 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 precede COBOL statements that define the host variables and host
 | variable arrays with the statement BEGIN DECLARE SECTION, and follow the
 | statements with the statement END DECLARE SECTION. You must use the
 | statements BEGIN DECLARE SECTION and END DECLARE SECTION when you
 | use the precompiler option STDSQL(YES).

COBOL

| A colon (:) must precede all host variables and all host variable arrays in an SQL
| statement.

| The names of host variables and host variable arrays should be unique within the
| source data set or member, even if the variables and variable arrays are in different
| blocks, classes, or procedures. You can qualify the names with a structure name to
| make them unique.

| An SQL statement that uses a host variable or host variable array must be within
| the scope of the statement that declares that variable or array. You define host
| variable arrays for use with multiple-row FETCH and INSERT statements.

| 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.

Declaring host variables

Only some of the valid COBOL declarations are valid host variable declarations. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.

Numeric host variables: The three valid forms of numeric host variables are:

- Floating-point numbers
- Integers and small integers
- Decimal numbers

Figure 65 shows the syntax for declarations of floating-point or real host variables.

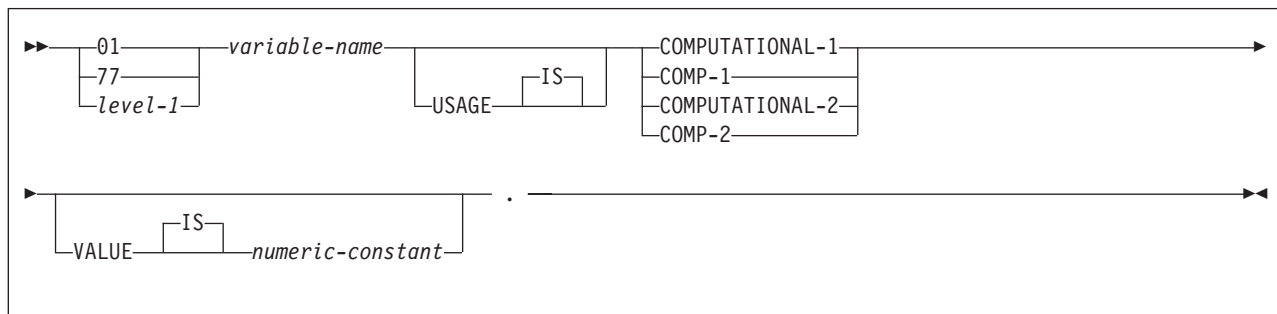


Figure 65. Floating-point host variables

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. COMPUTATIONAL-1 and COMP-1 are equivalent.
3. COMPUTATIONAL-2 and COMP-2 are equivalent.

Figure 66 on page 193 shows the syntax for declarations of integer and small integer host variables.

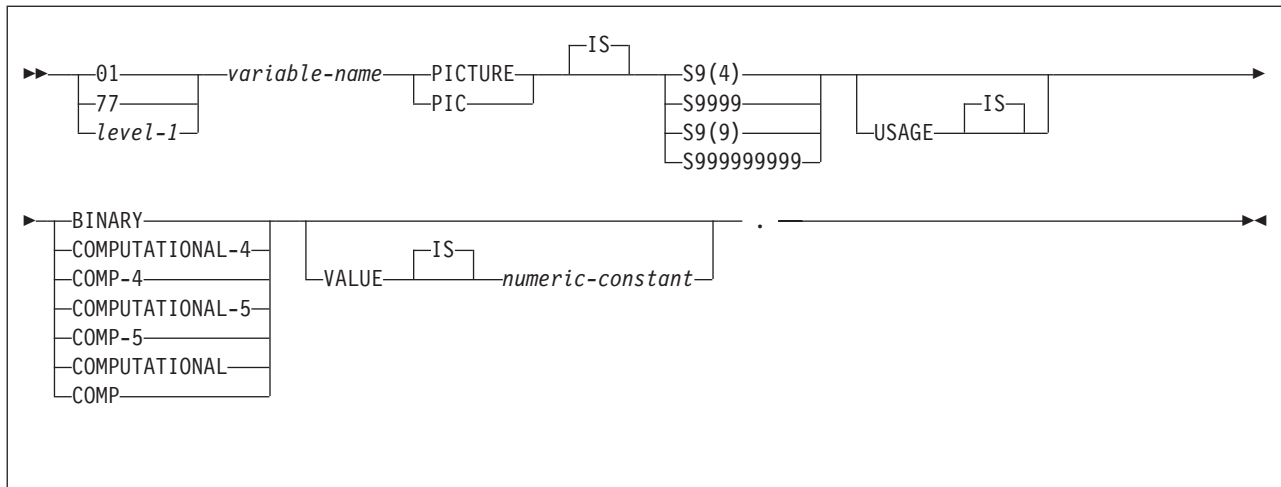


Figure 66. Integer and small integer host variables

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. The COBOL binary integer data types BINARY, COMPUTATIONAL, COMP, COMPUTATIONAL-4, and COMP-4 are equivalent.
3. COMPUTATIONAL-5 (and COMP-5) are equivalent to the other COBOL binary integer data types if you compile the other data types with TRUNC(BIN).
4. Any specification for scale is ignored.

Figure 67 shows the syntax for declarations of decimal host variables.

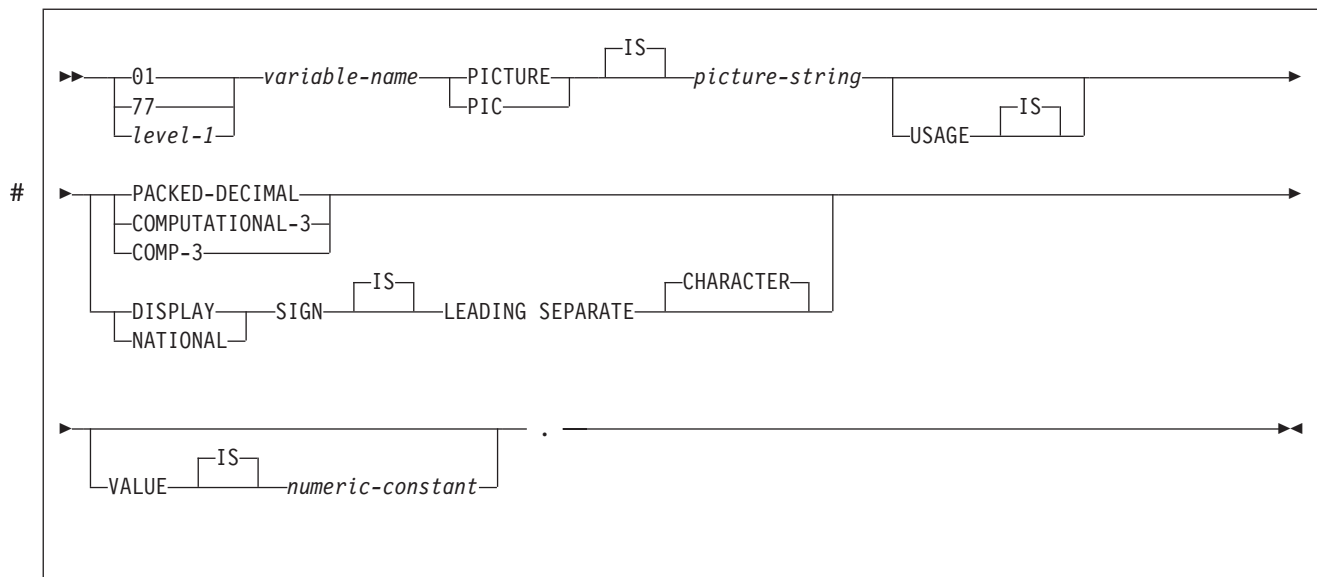


Figure 67. Decimal host variables

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 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).

Character host variables: The three valid forms of character host variables are:

- Fixed-length strings
- Varying-length strings
- CLOBs

The following figures show the syntax for forms other than CLOBs. See Figure 74 on page 198 for the syntax of CLOBs.

Figure 68 shows the syntax for declarations of fixed-length character host variables.

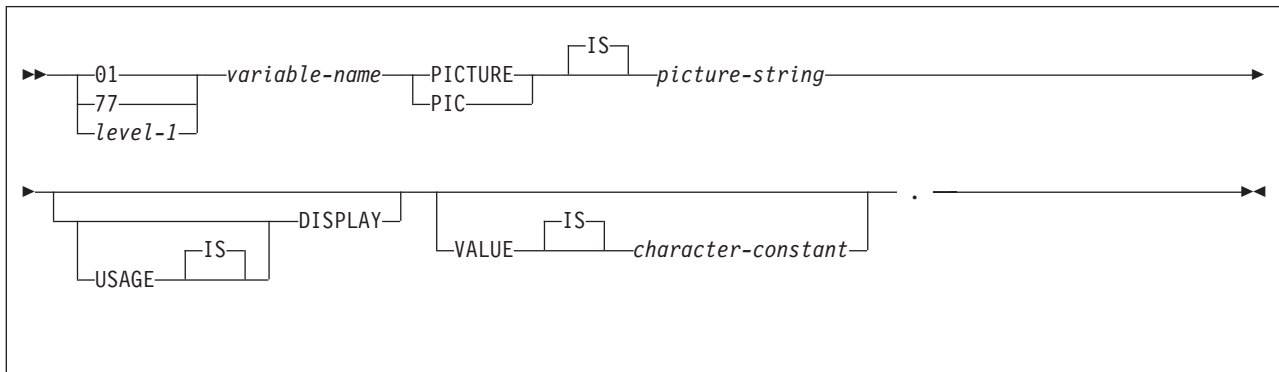


Figure 68. Fixed-length character strings

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. The *picture-string* that is associated with these forms must be X(*m*) (or XX...X, with *m* instances of X), with $1 \leq m \leq 32767$ for fixed-length strings. However, the maximum length of the CHAR data type (fixed-length character string) in DB2 is 255 bytes.

Figure 69 on page 195 shows the syntax for declarations of varying-length character host variables.

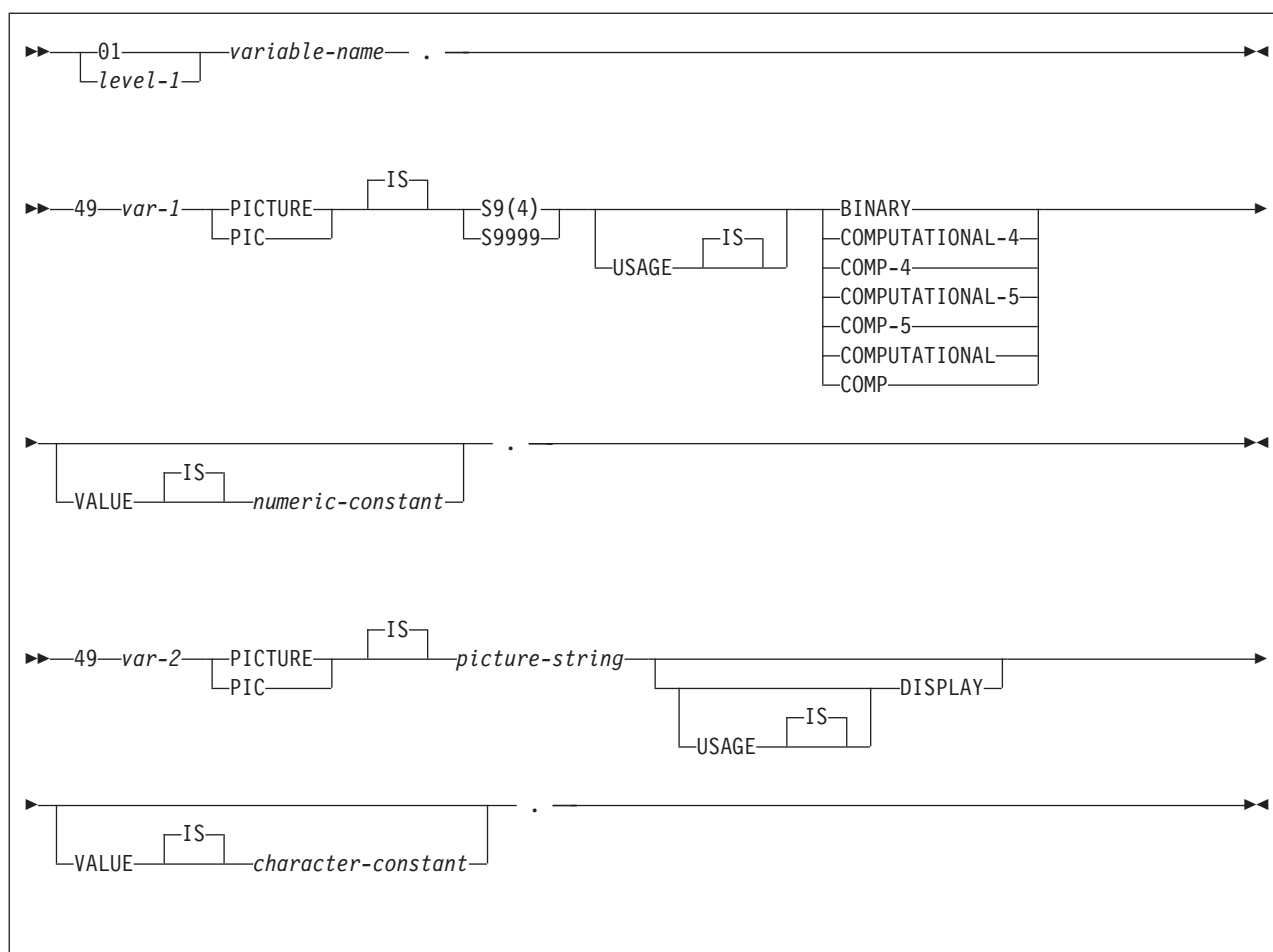


Figure 69. Varying-length character strings

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) only recognizes values up to 9999. This 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.
3. For fixed-length strings, the *picture-string* must be X(*m*) (or XX...X, with *m* instances of X), with $1 \leq m \leq 32767$; for other strings, *m* cannot be greater than the maximum size of a varying-length character string.
4. You cannot directly reference *var-1* and *var-2* as host variables.
5. You cannot use an intervening REDEFINE at level 49.

Graphic character host variables: The three valid forms for graphic character host variables are:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following figures show the syntax for forms other than DBCLOBs. See Figure 74 on page 198 for the syntax of DBCLOBs.

Figure 70 shows the syntax for declarations of fixed-length graphic host variables.

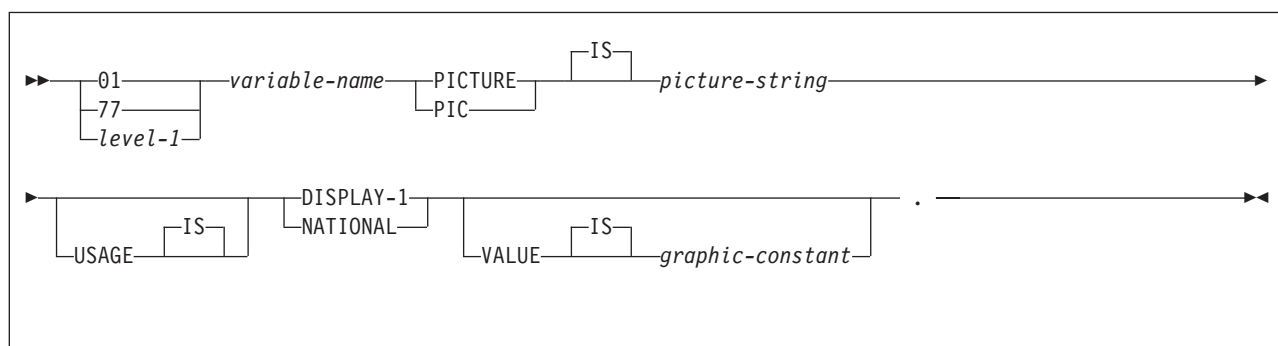


Figure 70. Fixed-length graphic strings

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. For fixed-length strings, the *picture-string* is *G(m)* or *N(m)* (or, *m* instances of *GG...G* or *NN...N*), with $1 \leq m \leq 127$; 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**. `USAGE NATIONAL` is supported only through the DB2 coprocessor.

Figure 71 on page 197 shows the syntax for declarations of varying-length graphic host variables.

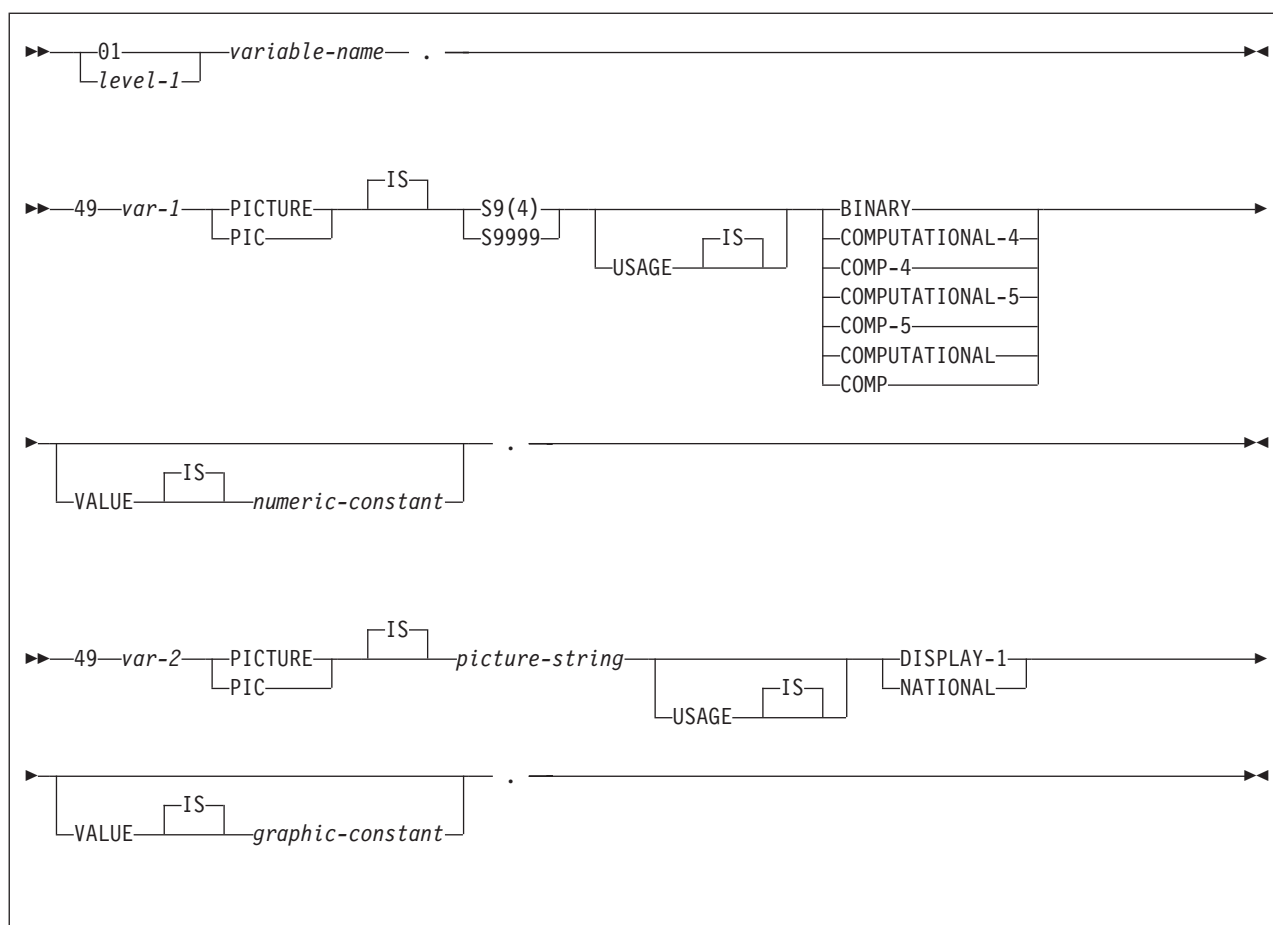


Figure 71. Varying-length graphic strings

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) only recognizes values up to 9999. This 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.
3. For fixed-length strings, the *picture-string* is G(*m*) or N(*m*) (or, *m* instances of GG...G or NN...N), with $1 \leq m \leq 127$; for other strings, *m* cannot be greater than the maximum size of a varying-length graphic string.
4. 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 through the DB2 coprocessor.
5. You cannot directly reference *var-1* and *var-2* as host variables.

Result set locators: Figure 72 on page 198 shows the syntax for declarations of result set locators. See Chapter 25, "Using stored procedures for client/server processing," on page 629 for a discussion of how to use these host variables.

#

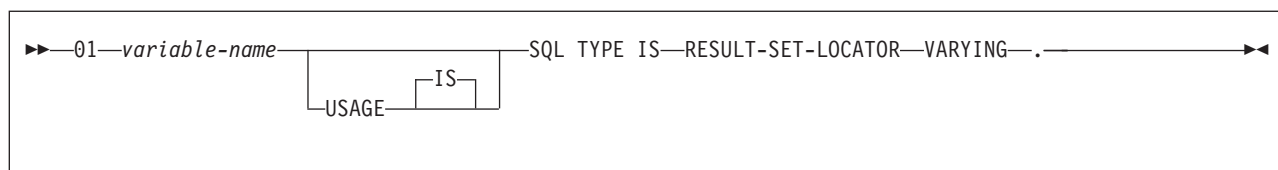


Figure 72. Result set locators

Table Locators: Figure 73 shows the syntax for declarations of table locators. See “Accessing transition tables in a user-defined function or stored procedure” on page 345 for a discussion of how to use these host variables.

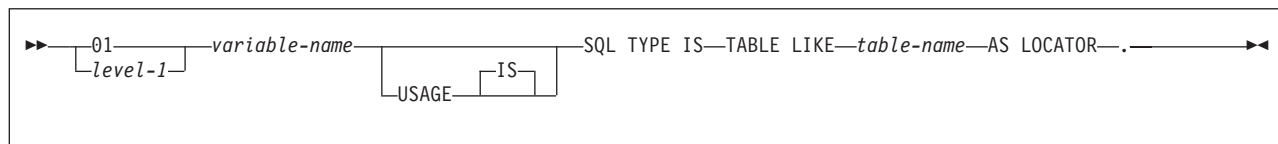


Figure 73. Table locators

Note: *level-1* indicates a COBOL level between 2 and 48.

LOB Variables and Locators: Figure 74 shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

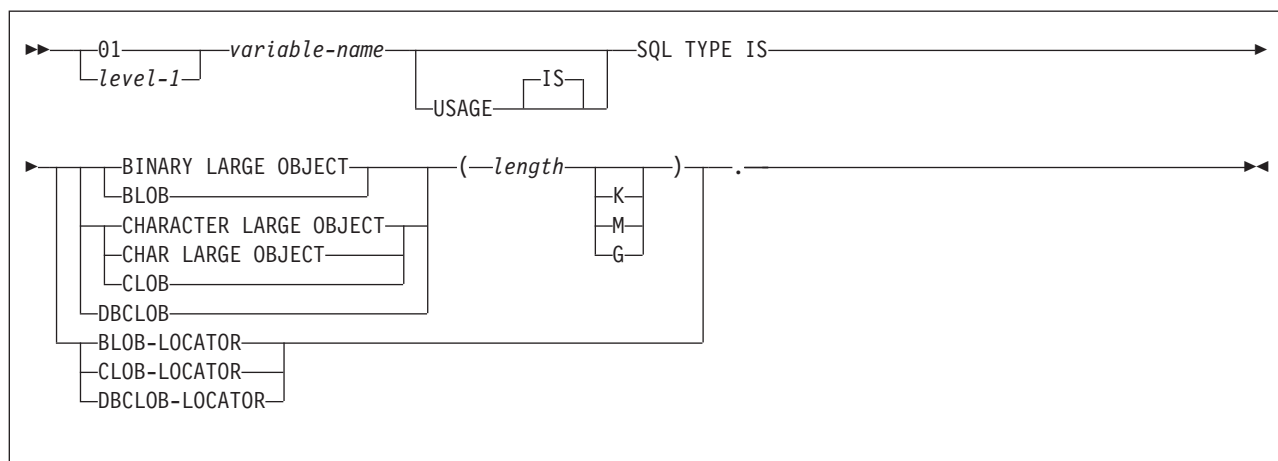


Figure 74. LOB variables and locators

Note: *level-1* indicates a COBOL level between 2 and 48.

ROWIDs: Figure 75 shows the syntax for declarations of ROWID host variables. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

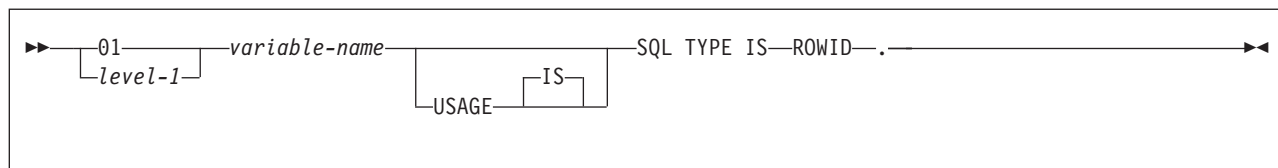


Figure 75. ROWID variables

Note: *level-1* indicates a COBOL level between 2 and 48.

Declaring host variable arrays

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: The three valid forms of numeric host variable arrays are:

- Floating-point numbers
- Integers and small integers
- Decimal numbers

Figure 76 shows the syntax for declarations of floating-point host variable arrays.

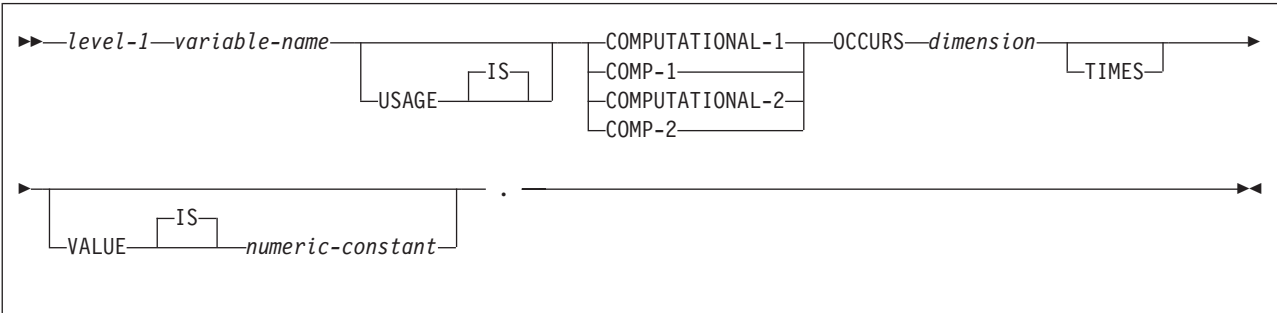


Figure 76. Floating-point host variable arrays

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. COMPUTATIONAL-1 and COMP-1 are equivalent.
3. COMPUTATIONAL-2 and COMP-2 are equivalent.
4. *dimension* must be an integer constant between 1 and 32767.

Figure 77 shows the syntax for declarations of integer and small integer host variable arrays.

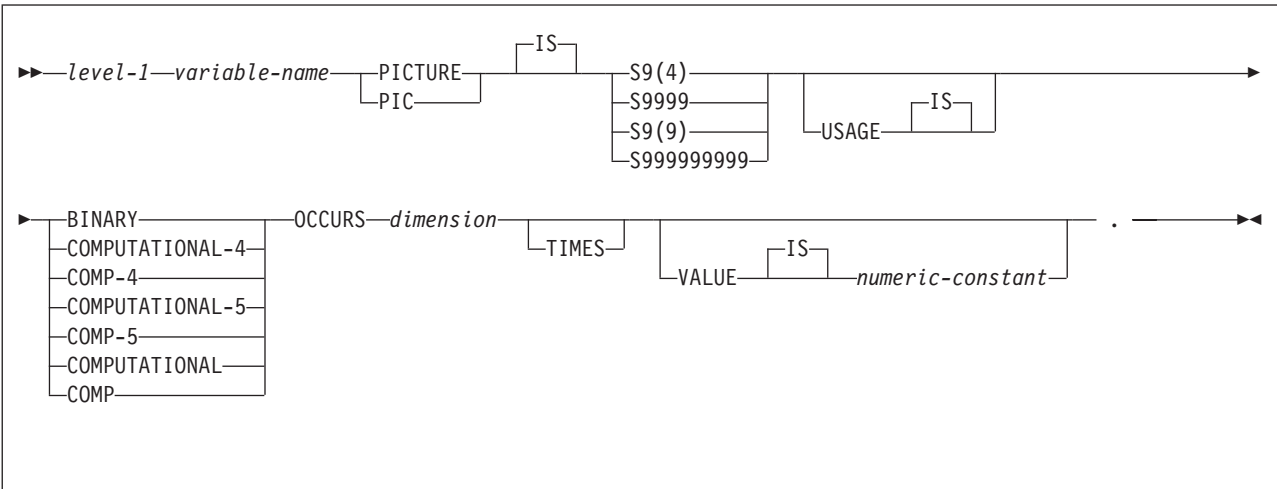


Figure 77. Integer and small integer host variable arrays

Notes:

COBOL

1. *level-1* indicates a COBOL level between 2 and 48.
2. The COBOL binary integer data types BINARY, COMPUTATIONAL, COMP, COMPUTATIONAL-4, and COMP-4 are equivalent.
3. COMPUTATIONAL-5 (and COMP-5) are equivalent to the other COBOL binary integer data types if you compile the other data types with TRUNC(BIN).
4. Any specification for scale is ignored.
5. *dimension* must be an integer constant between 1 and 32767.

Figure 78 shows the syntax for declarations of decimal host variable arrays.

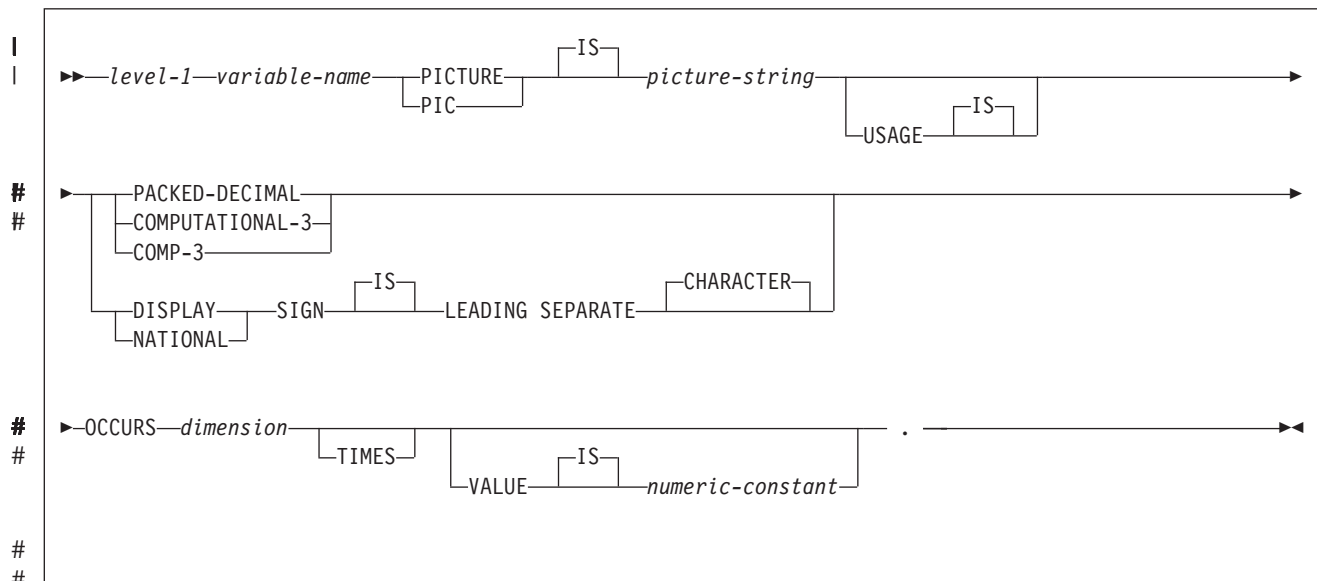


Figure 78. Decimal host variable arrays

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: The three valid forms of character host variable arrays are:

- Fixed-length character strings
- Varying-length character strings
- CLOBs

The following figures show the syntax for forms other than CLOBs. See Figure 83 on page 205 for the syntax of CLOBs.

Figure 79 on page 201 shows the syntax for declarations of fixed-length character string arrays.

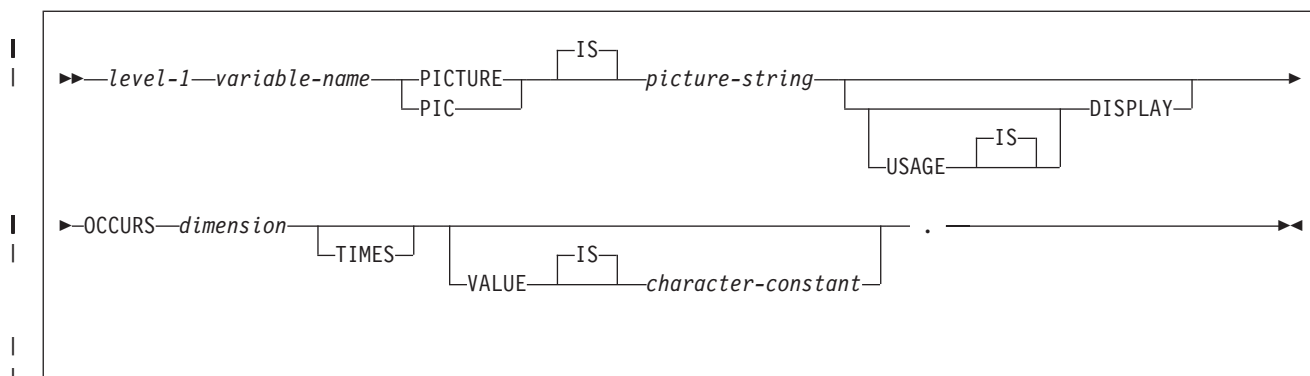


Figure 79. Fixed-length character string arrays

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. The *picture-string* that is associated with these forms must be $X(m)$ (or $XX...X$, with m instances of X), with $1 \leq m \leq 32767$ 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.

Figure 80 on page 202 shows the syntax for declarations of varying-length character string arrays.

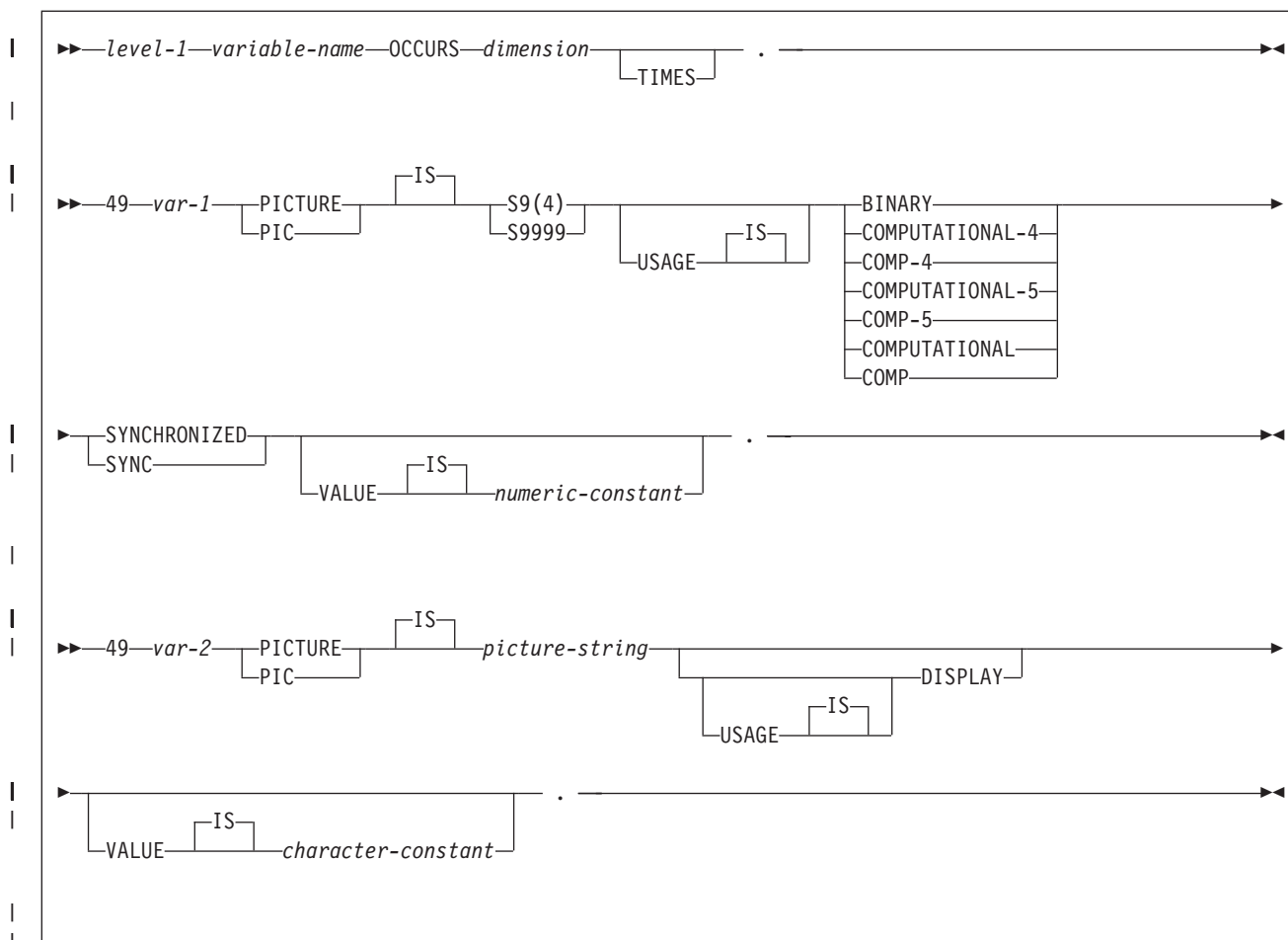


Figure 80. Varying-length character string arrays

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes only values up to 9999. This 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.
3. The *picture-string* that is associated with these forms must be X(*m*) (or XX...X, with *m* instances of X), with $1 \leq m \leq 32767$ for fixed-length strings; for other strings, *m* cannot be greater than the maximum size of a varying-length character string.
4. You cannot directly reference *var-1* and *var-2* as host variable arrays.
5. You cannot use an intervening REDEFINE at level 49.
6. *dimension* must be an integer constant between 1 and 32767.

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: The three valid forms for graphic character host variable arrays are:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following figures show the syntax for forms other than DBCLOBs. See Figure 83 on page 205 for the syntax of DBCLOBs.

Figure 81 shows the syntax for declarations of fixed-length graphic string arrays.

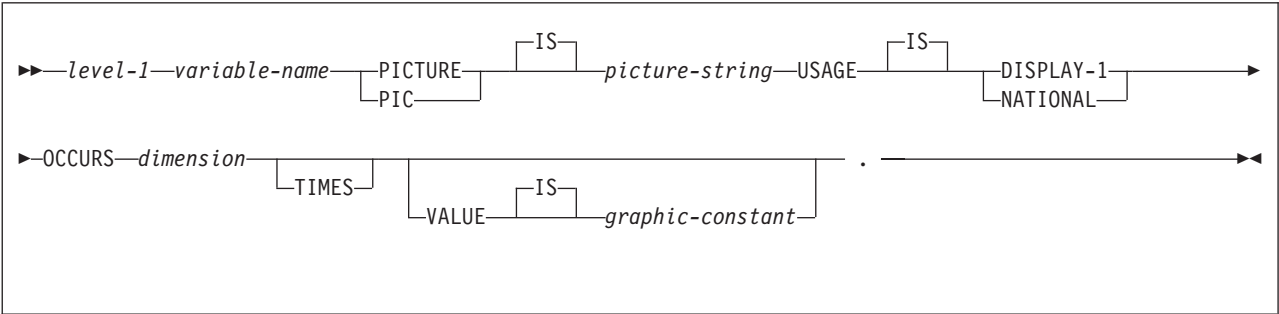


Figure 81. Fixed-length graphic string arrays

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. For fixed-length strings, the *picture-string* is *G(m)* or *N(m)* (or, *m* instances of *GG...G* or *NN...N*), with $1 \leq m \leq 127$; 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*. *USAGE NATIONAL* is supported only through the DB2 coprocessor.
4. *dimension* must be an integer constant between 1 and 32767.

Figure 82 on page 204 shows the syntax for declarations of varying-length graphic string arrays.

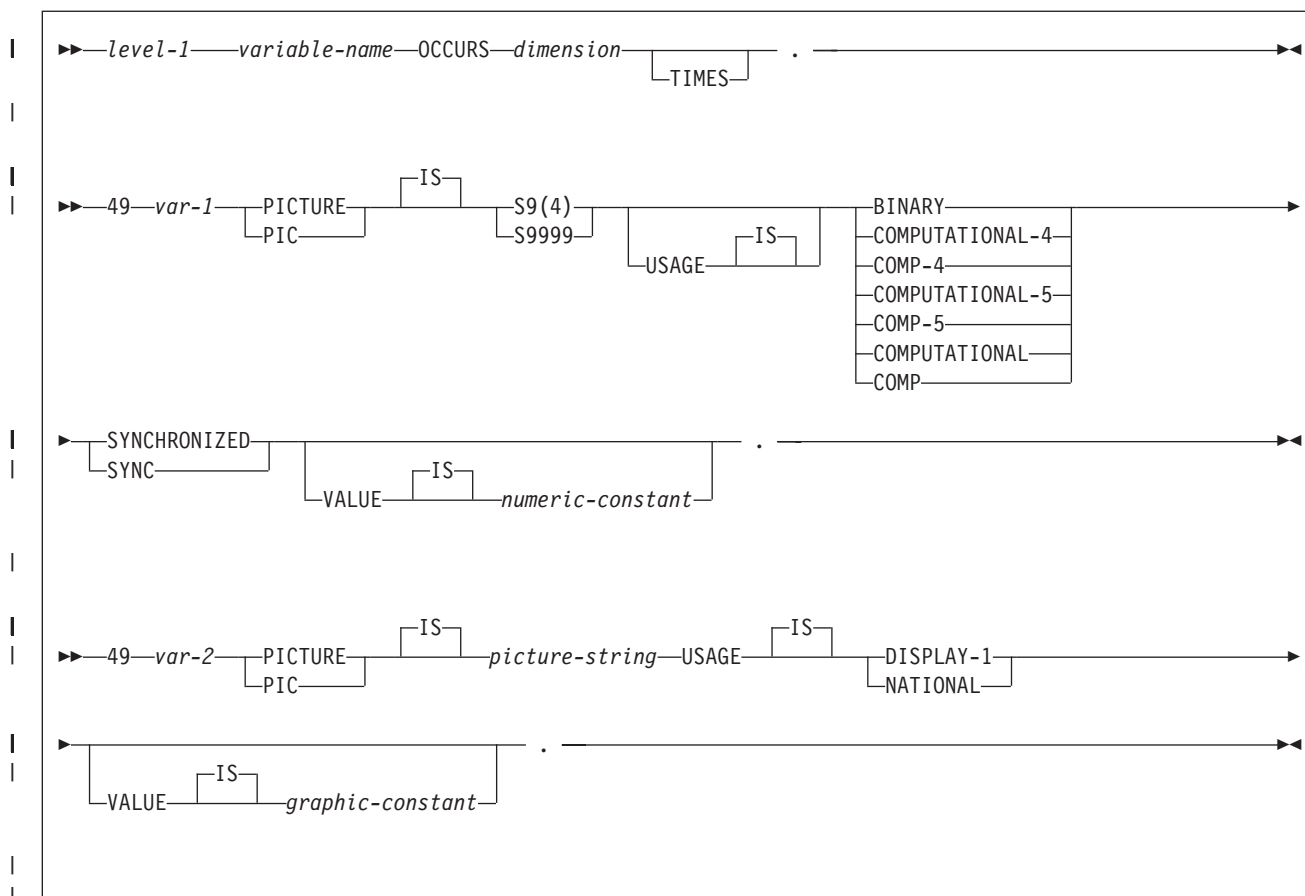


Figure 82. Varying-length graphic string arrays

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes only values up to 9999. This 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.
3. For fixed-length strings, the *picture-string* is G(*m*) or N(*m*) (or, *m* instances of GG...G or NN...N), with $1 \leq m \leq 127$; for other strings, *m* cannot be greater than the maximum size of a varying-length graphic string.
4. 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 through the DB2 coprocessor.
5. You cannot directly reference *var-1* and *var-2* as host variable arrays.
6. *dimension* must be an integer constant between 1 and 32767.

LOB variable arrays and locators: Figure 83 on page 205 shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variable arrays and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use LOB variables.

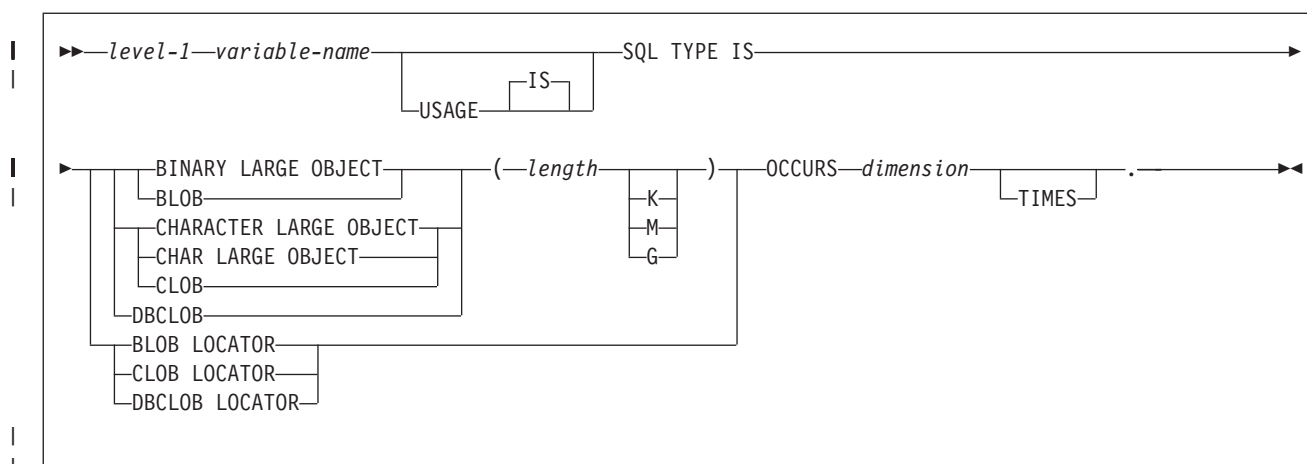


Figure 83. LOB variable arrays and locators

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. *dimension* must be an integer constant between 1 and 32767.

ROWIDs: Figure 84 shows the syntax for declarations of ROWID variable arrays. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

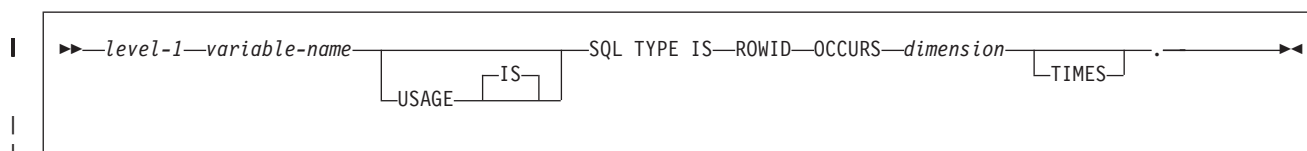


Figure 84. ROWID variable arrays

Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. *dimension* must be an integer constant between 1 and 32767.

Using host structures

A COBOL host structure is a named set of host variables defined in your program’s WORKING-STORAGE SECTION or LINKAGE SECTION. COBOL host structures have a maximum of two levels, even though the host structure might occur within a structure with multiple levels. However, you can declare a varying-length character string, which must be level 49.

A host structure name can be a group name whose subordinate levels name elementary data items. In the following example, B is the name of a host structure consisting of the elementary items C1 and C2.

```

01 A
  02 B
    03 C1 PICTURE ...
    03 C2 PICTURE ...
  
```

When you write an SQL statement using a qualified host variable name (perhaps to identify a field within a structure), use the name of the structure followed by a period and the name of the field. For example, specify B.C1 rather than C1 OF B or C1 IN B.

The precompiler does not recognize host variables or host structures on any subordinate levels after one of these items:

- A COBOL item that must begin in area A
- Any SQL statement (except SQL INCLUDE)
- Any SQL statement within an included member

When the precompiler encounters one of the preceding items in a host structure, it considers the structure to be complete.

Figure 85 shows the syntax for declarations of host structures.

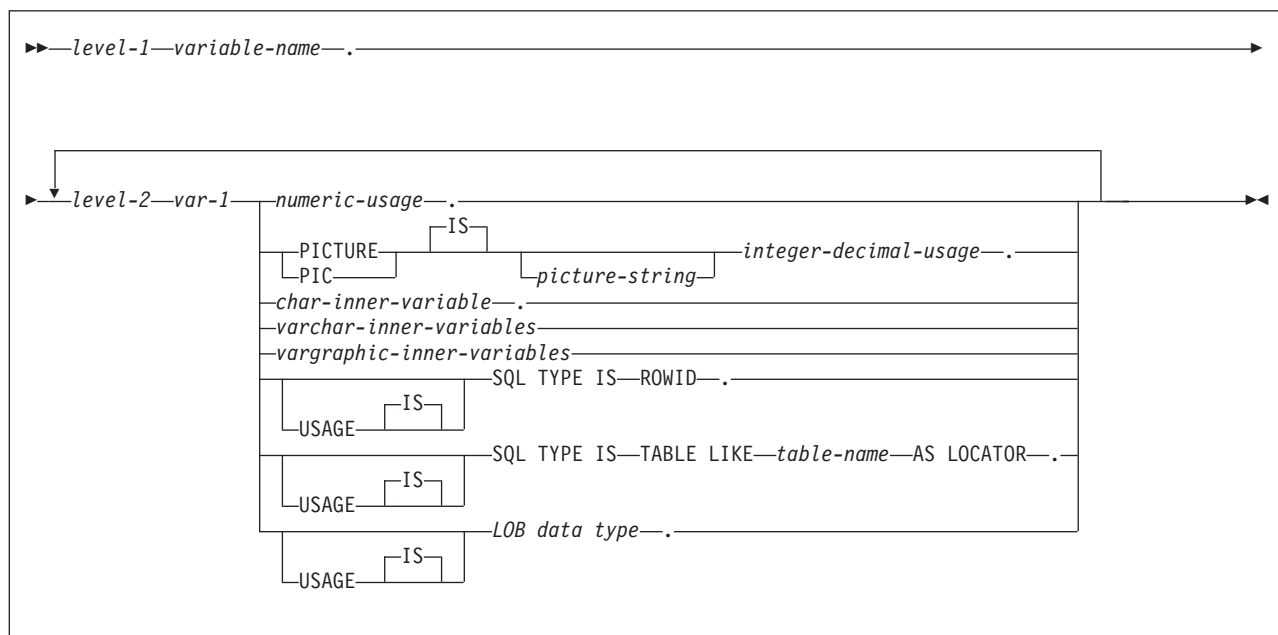


Figure 85. Host structures in COBOL

Figure 86 shows the syntax for numeric-usage items that are used within declarations of host structures.

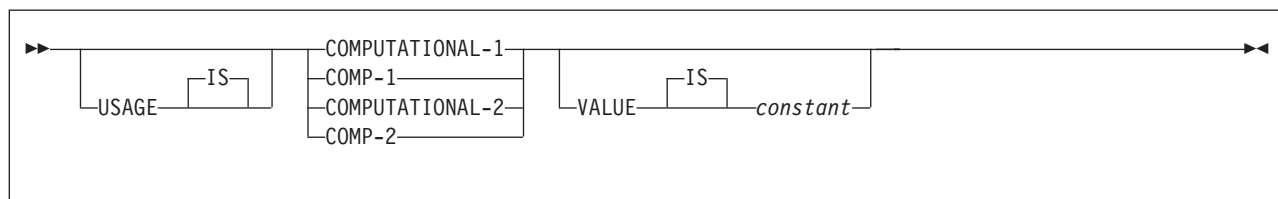


Figure 86. Numeric-usage

Figure 87 on page 207 shows the syntax for integer and decimal usage items that are used within declarations of host structures.

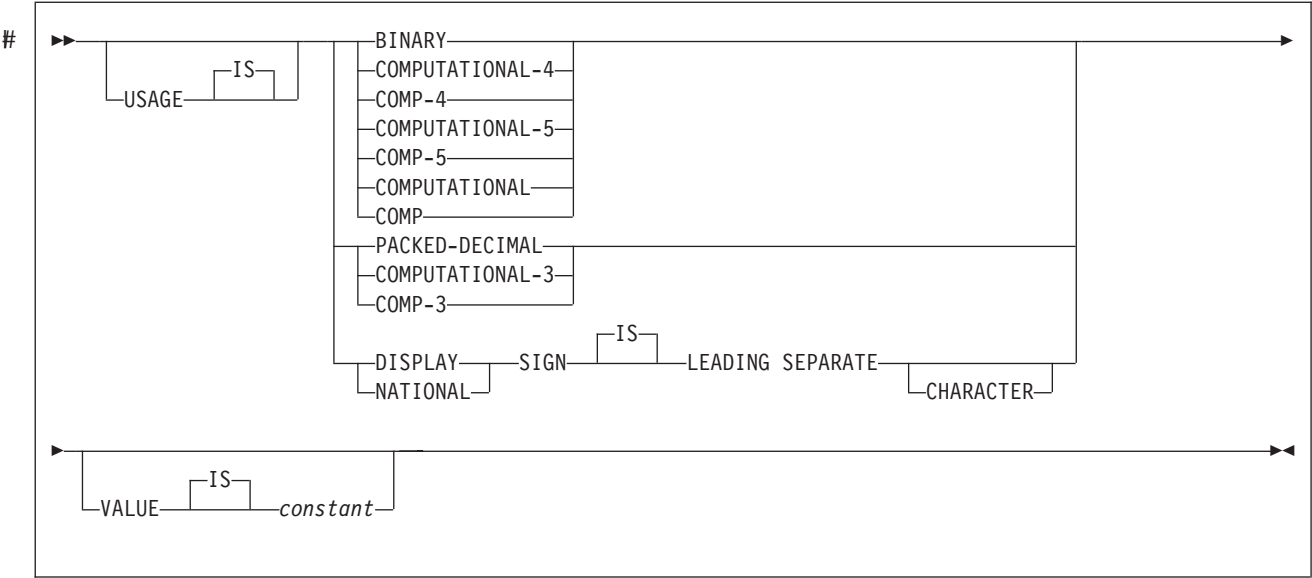


Figure 87. Integer-decimal-usage

Figure 88 shows the syntax for CHAR inner variables that are used within declarations of host structures.

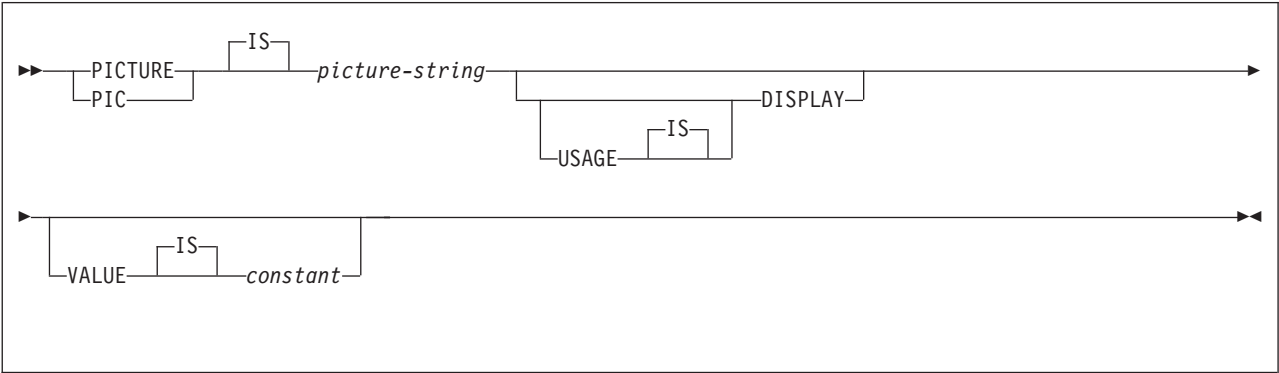


Figure 88. CHAR-inner-variable

Figure 89 on page 208 shows the syntax for VARCHAR inner variables that are used within declarations of host structures.

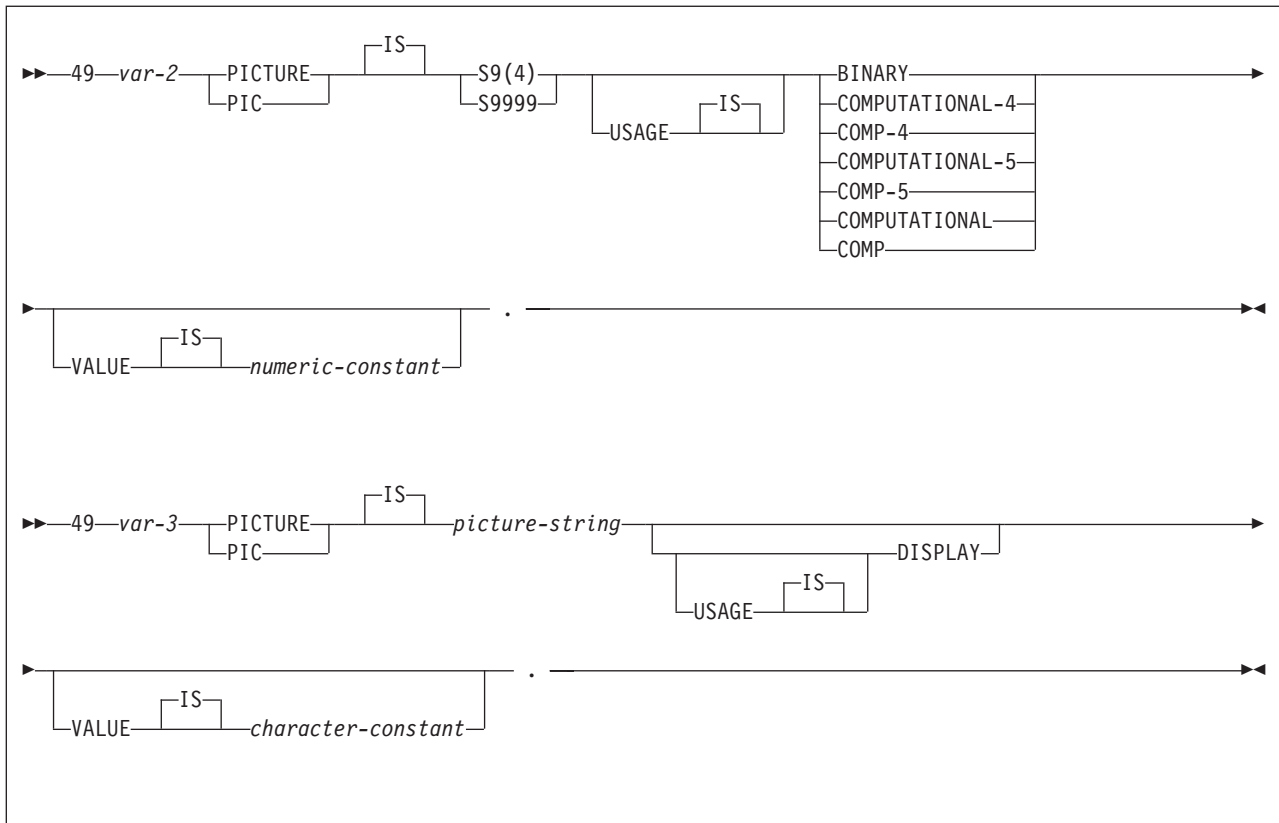


Figure 89. VARCHAR-inner-variables

Figure 90 on page 209 shows the syntax for VARGRAPHIC inner variables that are used within declarations of host structures.

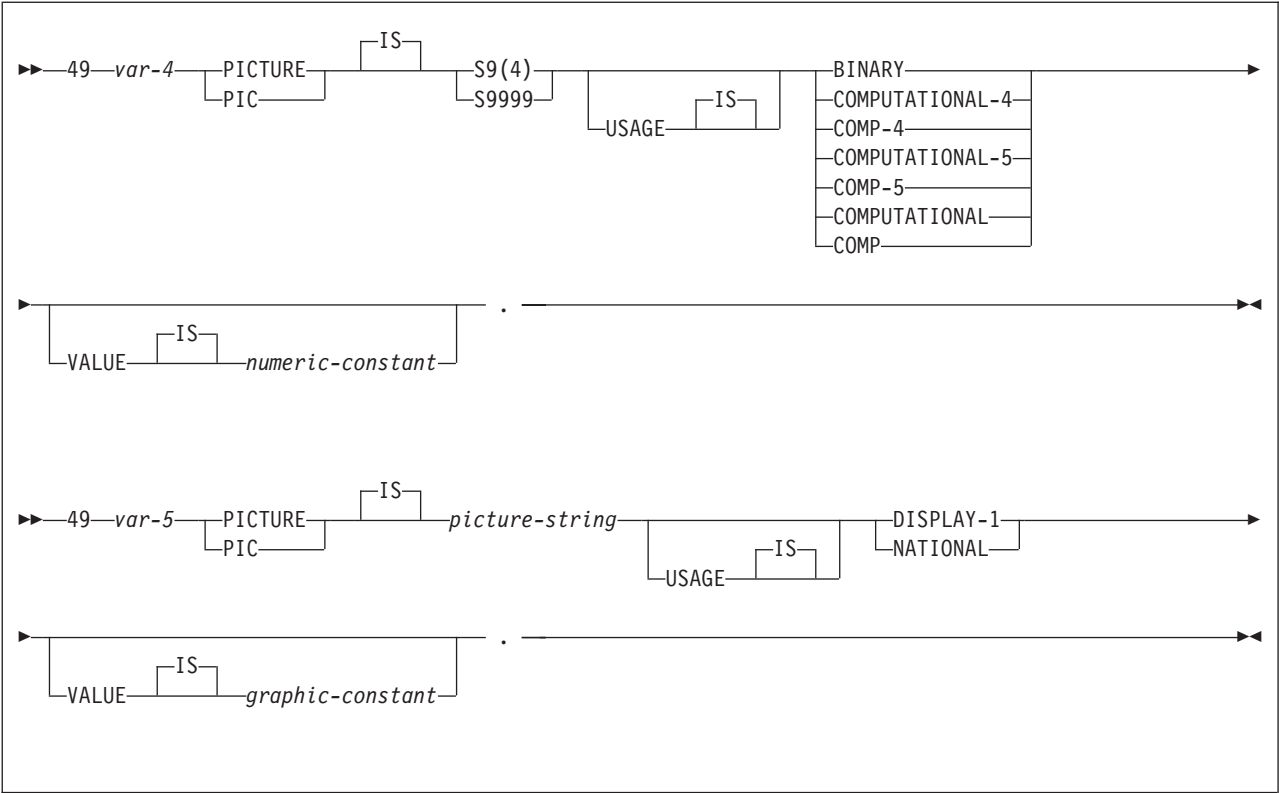


Figure 90. VARGRAPHIC-inner-variables

Notes:

1. For fixed-length strings, the *picture-string* is G(*m*) or N(*m*) (or, *m* instances of GG...G or NN...N), with 1 <= *m* <= 127; for other strings, *m* cannot be greater than the maximum size of a varying-length graphic string.
2. 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 through the DB2 coprocessor.

#

Figure 91 shows the syntax for LOB variables and locators that are used within declarations of host structures.

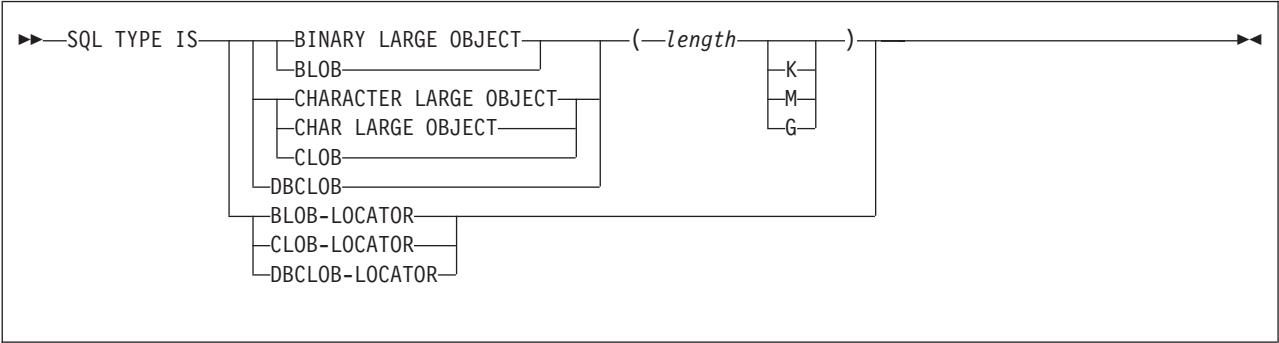


Figure 91. LOB variables and locators

Notes:

1. *level-1* indicates a COBOL level between 1 and 47.
2. *level-2* indicates a COBOL level between 2 and 48.

3. For elements within a structure, use any level 02 through 48 (rather than 01 or 77), up to a maximum of two levels.
4. Using a FILLER or optional FILLER item within a host structure declaration can invalidate the whole structure.

Determining equivalent SQL and COBOL data types

Table 16 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 16. SQL data types the precompiler uses for COBOL declarations

COBOL data type	SQLTYPE of host variable	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
S9(<i>i</i>)V9(<i>d</i>) COMP-3 or S9(<i>i</i>)V9(<i>d</i>) PACKED-DECIMAL	484	<i>i</i> + <i>d</i> in byte 1, <i>d</i> in byte 2	DECIMAL(<i>i</i> + <i>d</i> , <i>d</i>) or NUMERIC(<i>i</i> + <i>d</i> , <i>d</i>)
S9(<i>i</i>)V9(<i>d</i>) DISPLAY SIGN LEADING SEPARATE	504	<i>i</i> + <i>d</i> in byte 1, <i>d</i> in byte 2	No exact equivalent. Use DECIMAL(<i>i</i> + <i>d</i> , <i>d</i>) or NUMERIC(<i>i</i> + <i>d</i> , <i>d</i>)
# S9(<i>i</i>)V9(<i>d</i>) NATIONAL SIGN # LEADING SEPARATE #	504	<i>i</i> + <i>d</i> in byte 1, <i>d</i> in byte 2	No exact equivalent. Use DECIMAL(<i>i</i> + <i>d</i> , <i>d</i>) or NUMERIC(<i>i</i> + <i>d</i> , <i>d</i>)
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
Fixed-length character data	452	<i>n</i>	CHAR(<i>n</i>)
Varying-length character data 1<= <i>n</i> <=255	448	<i>n</i>	VARCHAR(<i>n</i>)
Varying-length character data <i>m</i> >255	456	<i>m</i>	VARCHAR(<i>m</i>)
Fixed-length graphic data	468	<i>m</i>	GRAPHIC(<i>m</i>)
Varying-length graphic data 1<= <i>m</i> <=127	464	<i>m</i>	VARGRAPHIC(<i>m</i>)
Varying-length graphic data <i>m</i> >127	472	<i>m</i>	VARGRAPHIC(<i>m</i>)
SQL TYPE IS RESULT-SET-LOCATOR	972	4	Result set locator ¹
SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	976	4	Table locator ¹
SQL TYPE IS BLOB-LOCATOR	960	4	BLOB locator ¹
SQL TYPE IS CLOB-LOCATOR	964	4	CLOB locator ¹
USAGE IS SQL TYPE IS DBCLOB-LOCATOR	968	4	DBCLOB locator ¹
USAGE IS SQL TYPE IS BLOB(<i>n</i>) 1≤ <i>n</i> ≤2147483647	404	<i>n</i>	BLOB(<i>n</i>)

Table 16. SQL data types the precompiler uses for COBOL declarations (continued)

COBOL data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
USAGE IS SQL TYPE IS CLOB(<i>n</i>) 1≤ <i>n</i> ≤2147483647	408	<i>n</i>	CLOB(<i>n</i>)
USAGE IS SQL TYPE IS DBCLOB(<i>m</i>) 1≤ <i>m</i> ≤1073741823 ²	412	<i>n</i>	DBCLOB(<i>m</i>) ²
SQL TYPE IS ROWID	904	40	ROWID

Notes:

1. Do not use this data type as a column type.
2. *m* is the number of double-byte characters.

Table 17 helps you define host variables that receive output from the database. You can use the table to determine the COBOL data type that is equivalent to a given SQL data type. For example, if you retrieve `TIMESTAMP` data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 17 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

Table 17. SQL data types mapped to typical COBOL declarations

SQL data type	COBOL data type	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
DOUBLE PRECISION, DOUBLE or FLOAT (<i>n</i>)	COMP-2	22≤ <i>n</i> ≤53
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.

COBOL

Table 17. SQL data types mapped to typical COBOL declarations (continued)

SQL data type	COBOL data type	Notes
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.
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 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.
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
ROWID	SQL TYPE IS ROWID	

Notes on COBOL variable declaration and usage

You should be aware of the following considerations when you declare COBOL host variables.

Controlling the CCSID: IBM Enterprise COBOL for z/OS Version 3 Release 2 or later, and the DB2 coprocessor for the COBOL compiler, support:

- The NATIONAL data type that is used for declaring Unicode values in the UTF-16 format (that is, CCSID 1200)
- The COBOL CODEPAGE compiler option that is used to specify the default EBCDIC CCSID of character data items

You can use the NATIONAL data type and the CODEPAGE compiler option to control the CCSID of the character host variables in your application.

For example, if you declare the host variable HV1 as USAGE NATIONAL, then DB2 handles HV1 as if you had used this DECLARE VARIABLE statement:

```
DECLARE :HV1 VARIABLE CCSID 1200
```

In addition, the COBOL DB2 coprocessor uses the CCSID that is specified in the CODEPAGE compiler option to indicate that 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.

Example: Assume that the COBOL CODEPAGE compiler option is specified as CODEPAGE(1234). 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.
```

The CCSID for each of these host variables is:

HV1 1200

HV2 1234

HV3 1047

SQL data types with no COBOL equivalent: 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 choose integer, 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, you can use floating-point numbers. 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.

Special purpose COBOL data types: The locator data types are COBOL data types and SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Result set locator

Chapter 25, “Using stored procedures for client/server processing,” on page 629

Table locator “Accessing transition tables in a user-defined function or stored procedure” on page 345

LOB locators Chapter 14, “Programming for large objects,” on page 297

Level 77 data description entries: 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.

SMALLINT and INTEGER data types: 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.

Overflow: Be careful of overflow. For example, suppose 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.

VARCHAR and VARGRAPHIC data types: If your varying-length character host variables receive values whose length is greater than 9999 characters, compile the applications in which you use those host variables with the option TRUNC(BIN). TRUNC(BIN) lets the length field for the character string receive a value of up to 32767.

Truncation: 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.

Determining compatibility of SQL and COBOL data types

COBOL host variables that are used in SQL statements must be type compatible with the columns with which you intend to use them:

- Numeric data types are compatible with each other. See Table 17 on page 211 for the COBOL data types that are compatible with the SQL data types SMALLINT, INTEGER, DECIMAL, REAL, and DOUBLE PRECISION.
- Character data types are compatible with each other. A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length COBOL 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. A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length COBOL graphic string 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.

- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying length COBOL character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB 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. For information on assigning and comparing distinct types, see Chapter 16, "Creating and using distinct types," on page 367.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using indicator variables and indicator variable arrays

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 use indicator variables and indicator variable arrays in similar ways.

Using indicator variables: 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 you find in X is irrelevant.

When your program uses 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. For more information about indicator variables, see “Using indicator variables with host variables” on page 83.

Using indicator variable arrays: When you retrieve data into a host variable array, if a value in its indicator array is negative, you can disregard the contents of the corresponding element in the host variable array. For more information about indicator variable arrays, see “Using indicator variable arrays with host variable arrays” on page 87.

Declaring indicator variables: You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. You can define indicator variables as scalar variables or as array elements in a structure form or as an array variable using a single level OCCURS clause.

Example: The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :CLS-CD,
                                :DAY :DAY-IND,
                                :BGN :BGN-IND,
                                :END :END-IND
END-EXEC.
```

You can declare the 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.
```

Figure 92 on page 217 shows the syntax for declarations of indicator variables.

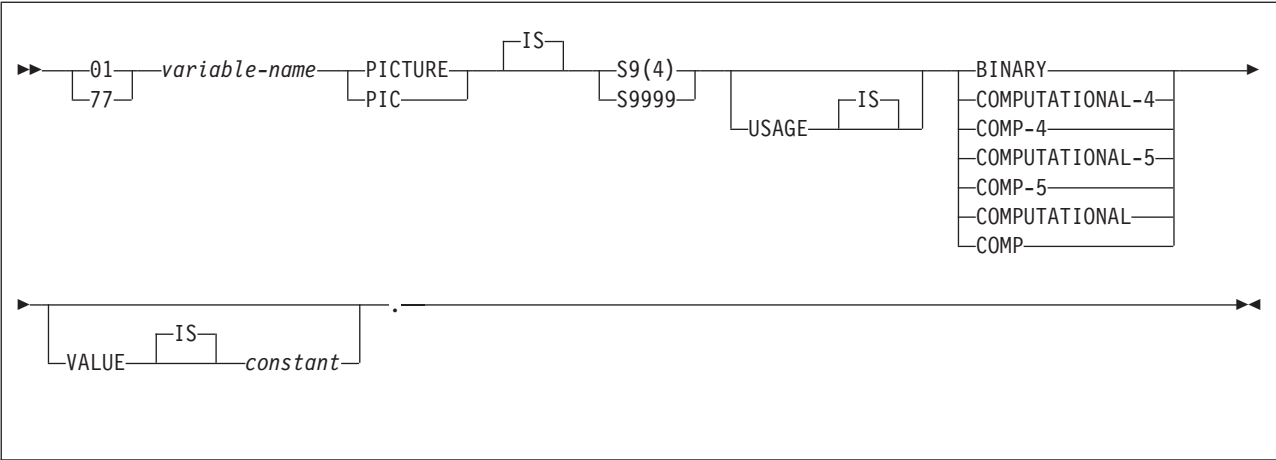


Figure 92. Indicator variable

Declaring indicator variable arrays: Figure 93 shows the syntax for valid indicator array declarations.

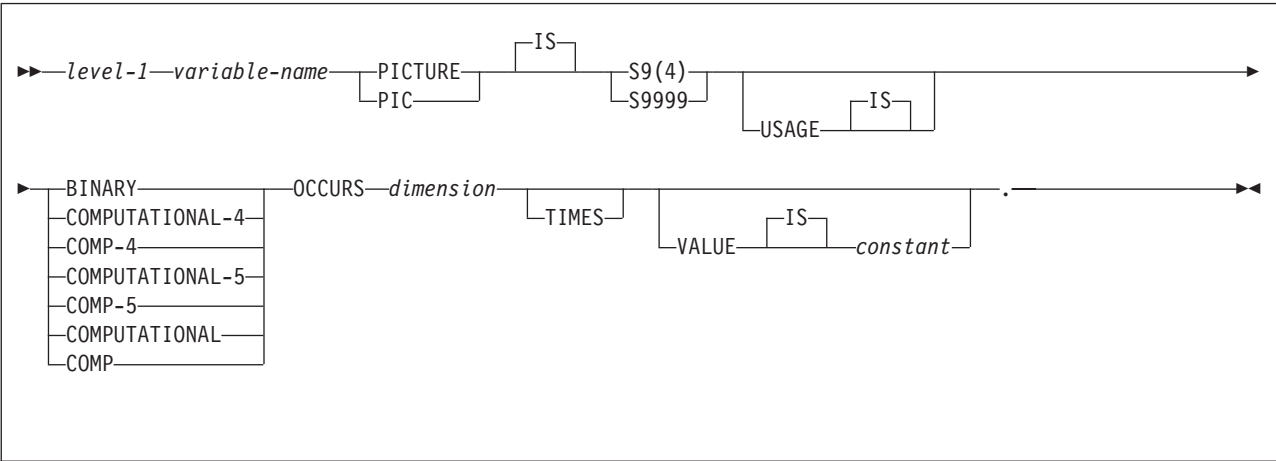


Figure 93. Host structure indicator array

- Notes:**
- 1. *level-1* must be an integer between 2 and 48.
 - 2. *dimension* must be an integer constant between 1 and 32767.

Handling SQL error return codes

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 “Calling DSNTIAR to display SQLCA fields” on page 98.

| 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 “The GET DIAGNOSTICS statement” on page 94.

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 DSN8810.SDSNSAMP. See Appendix B, “Sample applications,” on page 1013 for instructions on how to access and print the source code for the sample program.

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.

<i>eib</i>	EXEC interface block
<i>commarea</i>	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 DSNTJ5A.

The assembler source code for DSNTIAC and job DSNTJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix.SDSNSAMP*.

Coding considerations for object-oriented extensions in COBOL

When you use object-oriented extensions in a COBOL application, be aware of the following considerations:

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.

Coding SQL statements in a Fortran application

This section helps you with the programming techniques that are unique to coding SQL statements within a Fortran program.

Defining the SQL communication area

A Fortran program that contains SQL statements must include one or both of the following host variables:

- An SQLCOD variable declared as INTEGER*4
- An SQLSTA (or SQLSTATE) variable declared as CHARACTER*5

Alternatively, you can include an SQLCA, which contains the SQLCOD and SQLSTA variables.

DB2 sets the SQLCOD and SQLSTA (or SQLSTATE) values after each SQL statement executes. An application can check these values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCOD and SQLSTA (or SQLSTATE) variables.

Whether you define the SQLCOD or SQLSTA variable or an SQLCA in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)

When you use the precompiler 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 declare an SQLSTA (or SQLSTATE) variable, it must not be an element of a structure. You must declare the host variables SQLCOD and SQLSTA within the BEGIN DECLARE SECTION and END DECLARE SECTION statements in your program declarations.

If you specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a Fortran program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA
```

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix D of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:

- 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...USING 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. DB2 does not support the INCLUDE SQLDA statement for Fortran programs. If present, an error message results.

A Fortran program can call a subroutine (written in C, PL/I or assembler language) that uses the INCLUDE SQLDA statement to define the SQLDA and that also includes the necessary SQL statements for the dynamic SQL functions you want to perform. See Chapter 24, “Coding dynamic SQL in application programs,” on page 593 for more information about dynamic SQL.

You must place SQLDA declarations before the first SQL statement that references the data descriptor.

Embedding SQL statements

Fortran source statements must be fixed-length 80-byte records. The DB2 precompiler does not support free-form source input.

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.

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 DSN8810.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 section 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 220.

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.

Using host variables

You must explicitly declare each host variable that is used in SQL statements before its first use. You cannot implicitly declare any host variables through default typing or by using the IMPLICIT statement.

You can precede Fortran statements that define the host variables with a BEGIN DECLARE SECTION statement and follow the statements with an END DECLARE SECTION statement. You must use the BEGIN DECLARE SECTION and END DECLARE SECTION statements when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables in an SQL statement.

The names of host variables should be unique within the program, even if the host variables are in different blocks, functions, or subroutines.

When you declare a character host variable, you must 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 its associated SQL statement executes.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

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. See *DB2 SQL Reference* for more information.

Declaring host variables

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.

Numeric host variables: Figure 94 on page 224 shows the syntax for declarations of numeric host variables.

FORTRAN

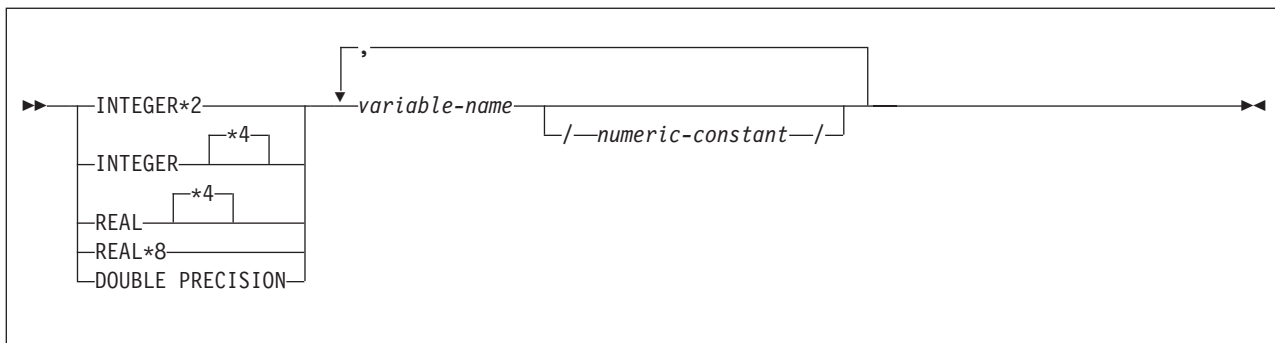


Figure 94. Numeric host variables

Character host variables: Figure 95 shows the syntax for declarations of character host variables other than CLOBs. See Figure 97 for the syntax of CLOBs.

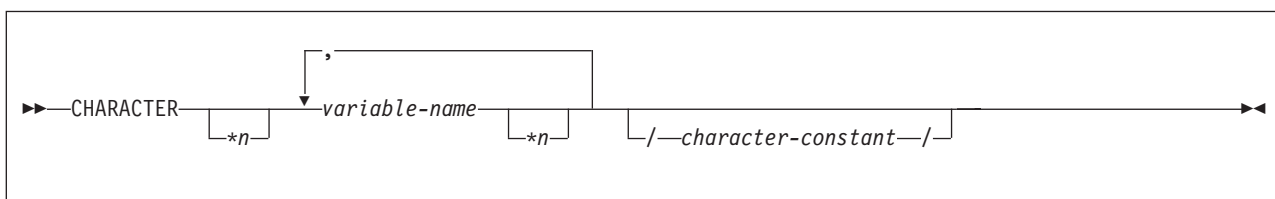


Figure 95. Character host variables

Result set locators: Figure 96 shows the syntax for declarations of result set locators. See Chapter 25, “Using stored procedures for client/server processing,” on page 629 for a discussion of how to use these host variables.

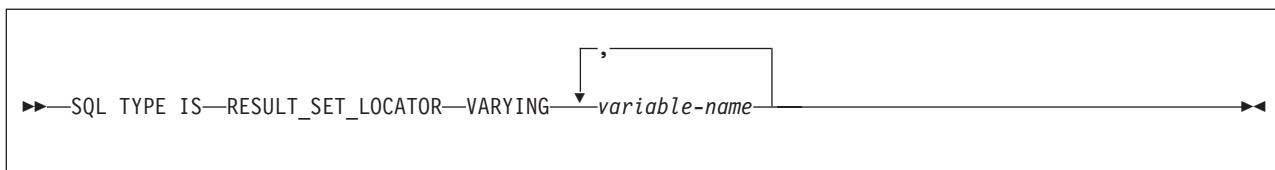


Figure 96. Result set locators

LOB Variables and Locators: Figure 97 shows the syntax for declarations of BLOB and CLOB host variables and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

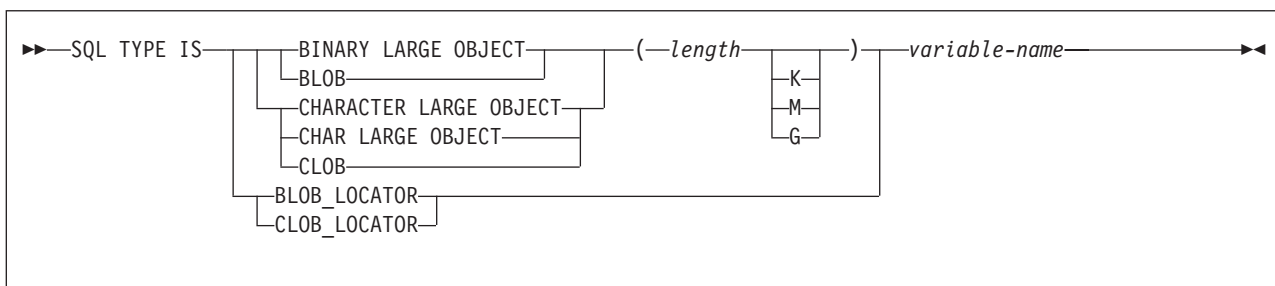


Figure 97. LOB variables and locators

ROWIDs: Figure 98 on page 225 shows the syntax for declarations of ROWID variables. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

►►—SQL TYPE IS—ROWID—*variable-name*—►►

Figure 98. ROWID variables

Determining equivalent SQL and Fortran data types

Table 18 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 18. SQL data types the precompiler uses for Fortran declarations

Fortran data type	SQLTYPE of host variable	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

Table 19 helps you define host variables that receive output from the database. You can use the table to determine the Fortran data type that is equivalent to a given SQL data type. For example, if you retrieve `TIMESTAMP` data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 19 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

Table 19. SQL data types mapped to typical Fortran declarations

SQL data type	Fortran equivalent	Notes
SMALLINT	INTEGER*2	
INTEGER	INTEGER*4	

FORTRAN

Table 19. SQL data types mapped to typical Fortran declarations (continued)

SQL data type	Fortran 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.
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.
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.
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	not supported	
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>)	not supported	
ROWID	SQL TYPE IS ROWID	

Notes on Fortran variable declaration and usage

You should be aware of the following when you declare Fortran variables.

Fortran data types with no SQL equivalent: 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.

SQL data types with no Fortran equivalent: Fortran does not provide an equivalent for the decimal data type. To hold the value of such a variable, you can use:

- An integer or floating-point variable, which converts the value. If you choose integer, however, 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, you can use floating-point numbers. Floating-point numbers are approximations of real numbers. When you assign a decimal number to a floating-point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

Special-purpose Fortran data types: The locator data types are Fortran data types and SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Result set locator

Chapter 25, “Using stored procedures for client/server processing,” on page 629

LOB locators Chapter 14, “Programming for large objects,” on page 297

Overflow: 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.

Truncation: 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.

Processing Unicode data: Because Fortran does not support graphic data types, Fortran applications can process only Unicode tables that use UTF-8 encoding.

Notes on syntax differences for constants

You should be aware of the following syntax differences for constants.

Real constants: 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.

Exponent indicators: In Fortran, a real (floating-point) constant having 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).

Determining compatibility of SQL and Fortran data types

Host variables must be type compatible with the column values with which you intend to use them.

- Numeric data types are compatible with each other. For example, if a column value is INTEGER, you must declare the host variable as INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, or DOUBLE PRECISION.
- Character data types are compatible with each other. A CHAR, VARCHAR, or CLOB column is compatible with Fortran 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.

- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a Fortran character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB 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. For information on assigning and comparing distinct types, see Chapter 16, “Creating and using distinct types,” on page 367.

Using indicator variables

An indicator variable is a 2-byte integer (INTEGER*2). 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 you find in X is irrelevant.

When your program uses 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.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see “Using indicator variables with host variables” on page 83.

Example: The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,
C                                :DAY :DAYIND,
C                                :BGN :BGNIND,
C                                :END :ENDIND
```

You can declare variables as follows:

```
CHARACTER*7 CLSCD
INTEGER*2   DAY
CHARACTER*8 BGN, END
INTEGER*2   DAYIND, BGNIND, ENDIND
```

Figure 99 shows the syntax for declarations of indicator variables.

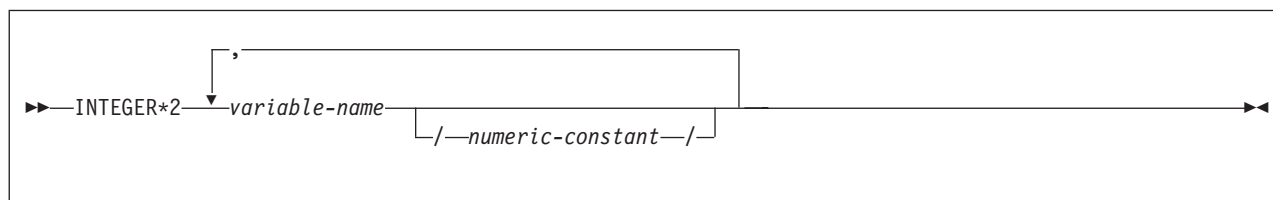


Figure 99. Indicator variable

Handling SQL error return codes

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 “Calling DSNTIAR to display SQLCA fields” on page 98.

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 “The GET DIAGNOSTICS statement” on page 94.

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 DSN8810.SDSNSAMP. See Appendix B, “Sample applications,” on page 1013 for instructions on how to access and print the source code for the sample program.

Coding SQL statements in a PL/I application

This section helps you with the programming techniques that are unique to coding SQL statements within a PL/I program.

Defining the SQL communication area

A PL/I program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable, declared as BIN FIXED (31)
- An SQLSTATE variable, declared as CHARACTER(5)

Alternatively, you can include an SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define the SQLCODE or SQLSTATE variable or an SQLCA in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If you specify STDSQL(YES)

When you use the precompiler 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 declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the BEGIN DECLARE SECTION and END DECLARE SECTION statements in your program declarations.

If you specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a PL/I program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA;
```

See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix D of *DB2 SQL Reference* for a complete description of SQLCA fields.

Defining SQL descriptor areas

The following statements require an SQLDA:

- 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 ... USING 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. You can code an SQLDA in a PL/I program, either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA;
```

You must declare an SQLDA before the first SQL statement that references that data descriptor, unless you use the precompiler option TWOPASS. See Chapter 5 of *DB2 SQL Reference* for more information about the INCLUDE statement and Appendix E of *DB2 SQL Reference* for a complete description of SQLDA fields.

Embedding SQL 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. See Chapter 25, “Using stored procedures for client/server processing,” on page 629 for more information.

You can code SQL statements in a PL/I program wherever you can use executable statements.

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 DSN8810.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. For more information, see Chapter 8, “Generating declarations for your tables using DCLGEN,” on page 131.

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

See Chapter 2 of *DB2 SQL Reference* for detailed information about these language elements.

- 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;
```

For instructions on preparing SQL statements dynamically, see Chapter 24, “Coding dynamic SQL in application programs,” on page 593.

- 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. See Chapter 2 of *DB2 SQL Reference* for the rules for DBCS SQL ordinary 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. See the RUN(DSN) command in Part 3 of *DB2 Command Reference*.

Using host variables and host variable arrays

You must explicitly declare all host variables and all host variable arrays before their first use in SQL statements, unless you specify the precompiler option TWOPASS. If you specify the precompiler option TWOPASS, you must declare a host variable before its use in the statement DECLARE CURSOR.

You can precede PL/I statements that define the host variables and host variable arrays with the BEGIN DECLARE SECTION statement, and follow the statements with the END DECLARE SECTION statement. You must use the BEGIN DECLARE SECTION and END DECLARE SECTION statements when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables and host variable arrays in an SQL statement, with the following exception. If the SQL statement meets the following conditions, a host variable or host variable array in the SQL statement **cannot** be preceded by a colon:

- The SQL statement is an EXECUTE IMMEDIATE or PREPARE statement.
- 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.

The names of host variables and host variable arrays should be unique within the program, even if the variables and variable arrays are in different blocks or procedures. You can qualify the names with a structure name to make them unique.

An SQL statement that uses a host variable or host variable array must be within the scope of the statement that declares that variable or array. You define host variable arrays for use with multiple-row FETCH and multiple-row INSERT statements.

Declaring host variables

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 precompiler uses only the names and data attributes of the variables; it ignores the alignment, scope, and storage attributes. Even though the precompiler ignores alignment, scope, and storage, if you ignore the restrictions on their use, you might have problems compiling the PL/I source code that the precompiler generates. These restrictions are as follows:

- 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.

Numeric host variables: Figure 100 on page 235 shows the syntax for declarations of numeric host variables.

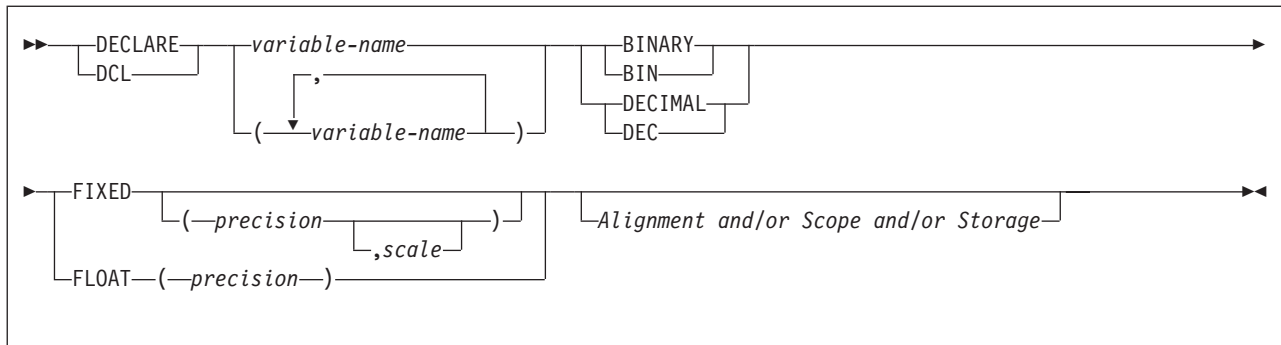


Figure 100. Numeric host variables

Notes:

1. You can specify host variable attributes in any order that is acceptable to PL/I. For example, BIN FIXED(31), BINARY FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.
2. You can specify a *scale* only for DECIMAL FIXED.

Character host variables: Figure 101 shows the syntax for declarations of character host variables, other than CLOBs. See Figure 105 on page 236 for the syntax of CLOBs.

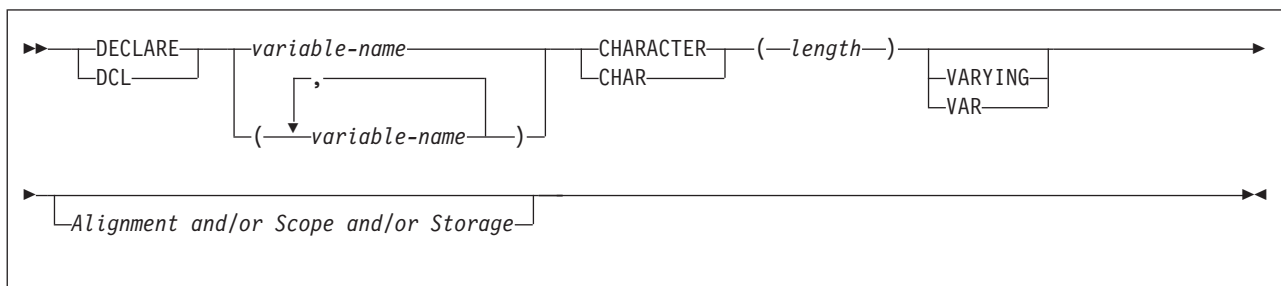


Figure 101. Character host variables

Graphic host variables: Figure 102 shows the syntax for declarations of graphic host variables, other than DBCLOBs. See Figure 105 on page 236 for the syntax of DBCLOBs.

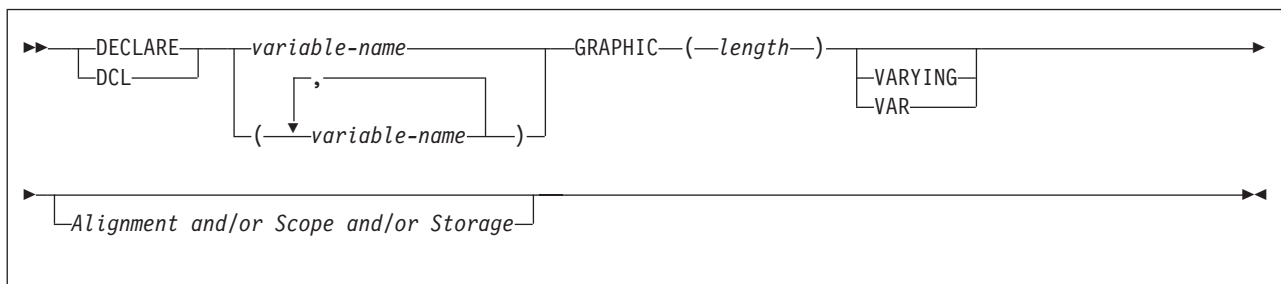


Figure 102. Graphic host variables

Result set locators: Figure 103 on page 236 shows the syntax for declarations of result set locators. See Chapter 25, "Using stored procedures for client/server processing," on page 629 for a discussion of how to use these host variables.

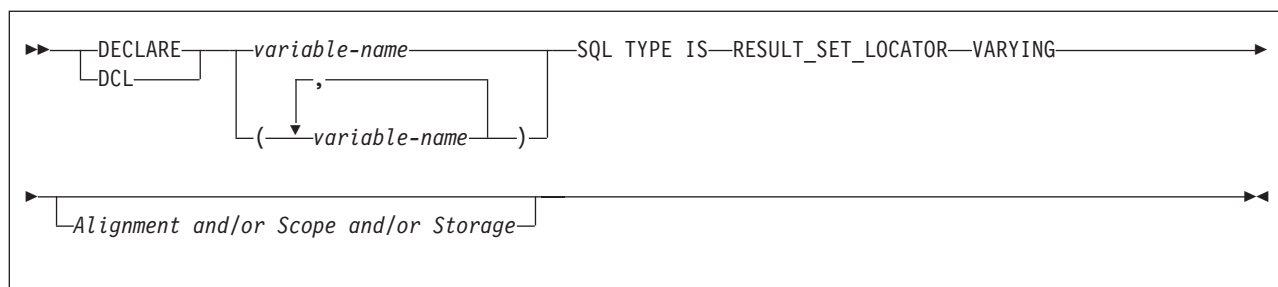


Figure 103. Result set locators

Table locators: Figure 104 shows the syntax for declarations of table locators. See “Accessing transition tables in a user-defined function or stored procedure” on page 345 for a discussion of how to use these host variables.

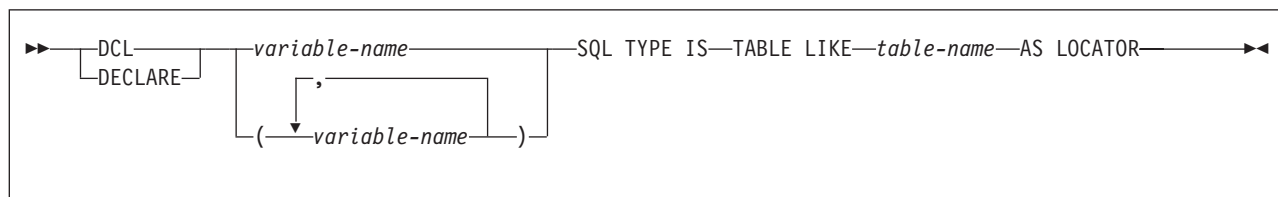


Figure 104. Table locators

LOB variables and locators: Figure 105 shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

A single PL/I declaration that contains a LOB variable declaration is limited to no more than 1000 lines of source code.

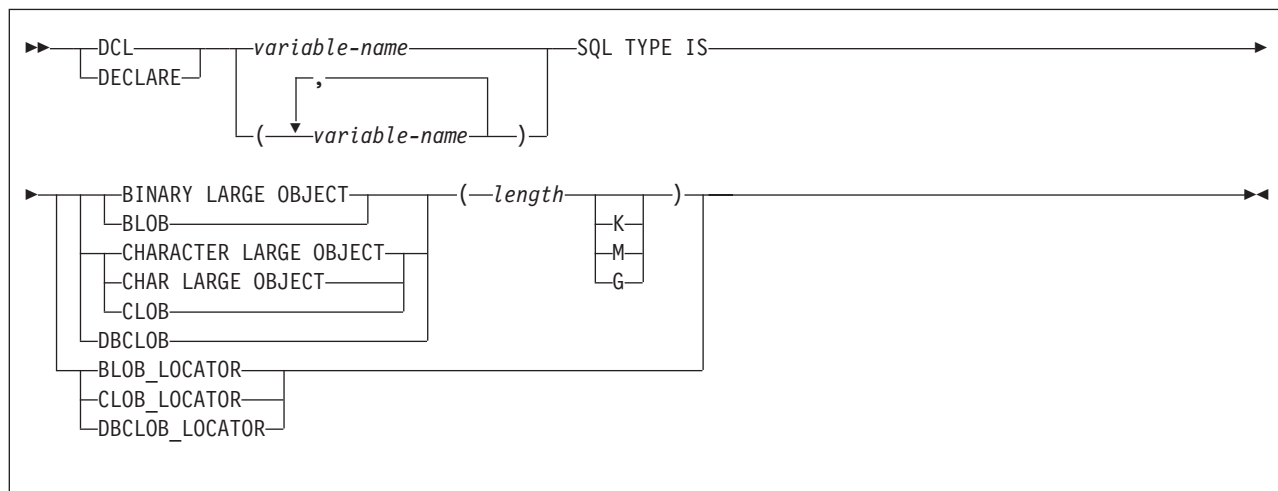


Figure 105. LOB variables and locators

Note: Variable attributes such as `STATIC` and `AUTOMATIC` are ignored if specified on a LOB variable declaration.

ROWIDs: Figure 106 on page 237 shows the syntax for declarations of ROWID host variables. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

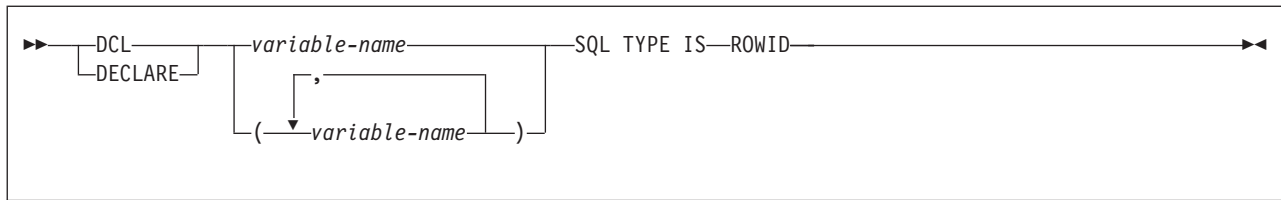


Figure 106. ROWID variables

Declaring host variable arrays

Only some of the valid PL/I declarations are valid host variable array declarations. The precompiler uses the data attribute defaults that are specified in the PL/I DEFAULT statement. If the declaration for a variable array is not valid, then any SQL statement that references the host variable array might result in the message UNDECLARED HOST VARIABLE ARRAY.

The precompiler uses only the names and data attributes of the variable arrays; it ignores the alignment, scope, and storage attributes. Even though the precompiler ignores alignment, scope, and storage, if you ignore the restrictions on their use, you might have problems compiling the PL/I source code that the precompiler generates. These restrictions are as follows:

- 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.

Declaring host variable arrays: 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: Figure 107 shows the syntax for declarations of numeric host variable arrays.

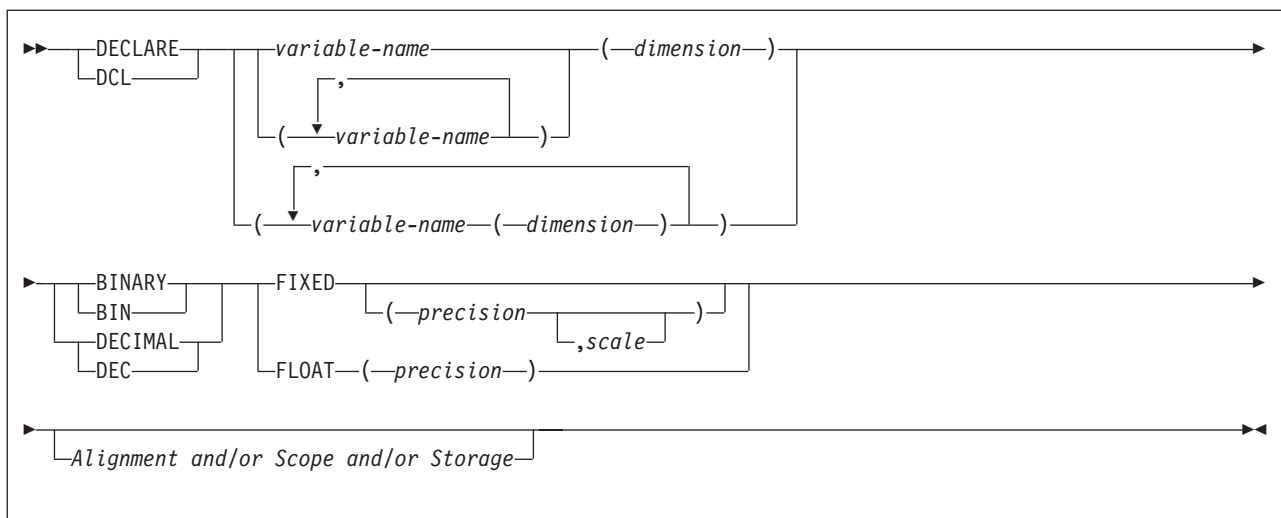


Figure 107. Numeric host variable arrays

Notes:

1. You can specify host variable array attributes in any order that is acceptable to PL/I. For example, BIN FIXED(31), BINARY FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.
2. You can specify the scale for only DECIMAL FIXED.
3. *dimension* must be an integer constant between 1 and 32767.

Example: The following example shows a declaration of an indicator array:

```
DCL IND_ARRAY(100) BIN FIXED(15);  /* DCL ARRAY of 100 indicator variables */
```

Character host variable arrays: Figure 108 shows the syntax for declarations of character host variable arrays, other than CLOBs. See Figure 110 on page 239 for the syntax of CLOBs.

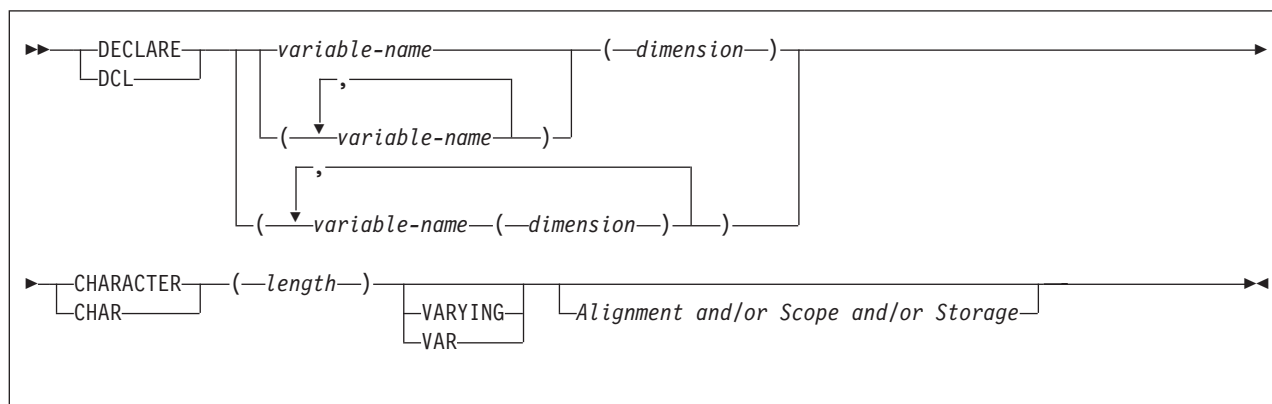


Figure 108. Character host variable arrays

Notes:

1. *dimension* must be an integer constant between 1 and 32767.

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: Figure 109 on page 239 shows the syntax for declarations of graphic host variable arrays, other than DBCLOBs. See Figure 110 on page 239 for the syntax of DBCLOBs.

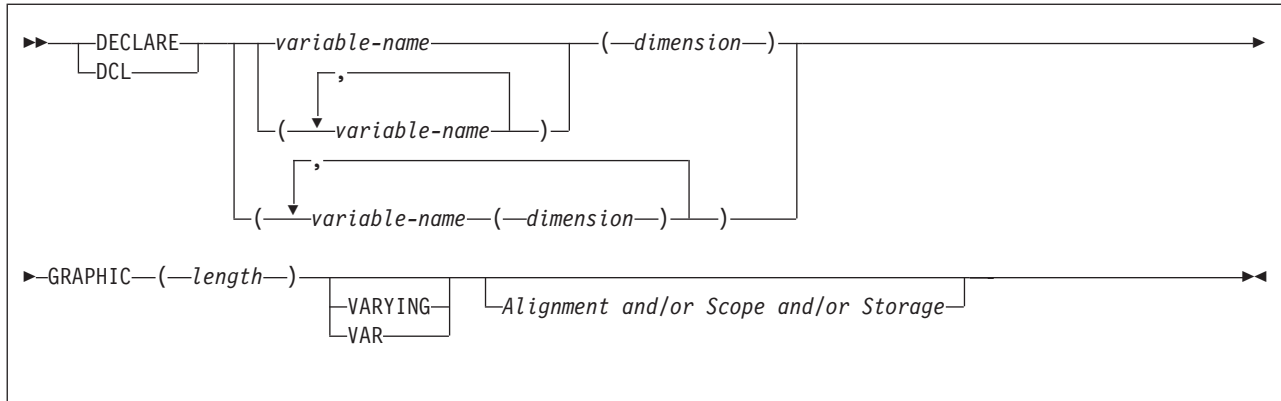


Figure 109. Graphic host variable arrays

Notes:

1. *dimension* must be an integer constant between 1 and 32767.

LOB variable arrays and locators: Figure 110 shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variable arrays and locators. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

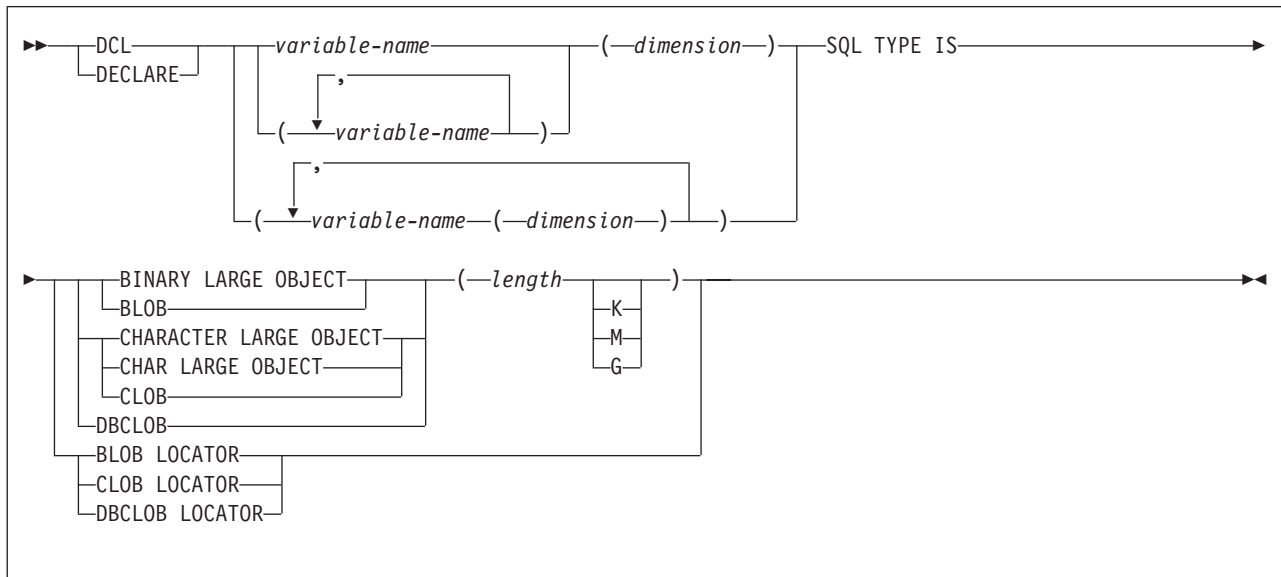


Figure 110. LOB variable arrays and locators

Notes:

1. *dimension* must be an integer constant between 1 and 32767.

ROWIDs: Figure 111 on page 240 shows the syntax for declarations of ROWID variable arrays. See Chapter 14, “Programming for large objects,” on page 297 for a discussion of how to use these host variables.

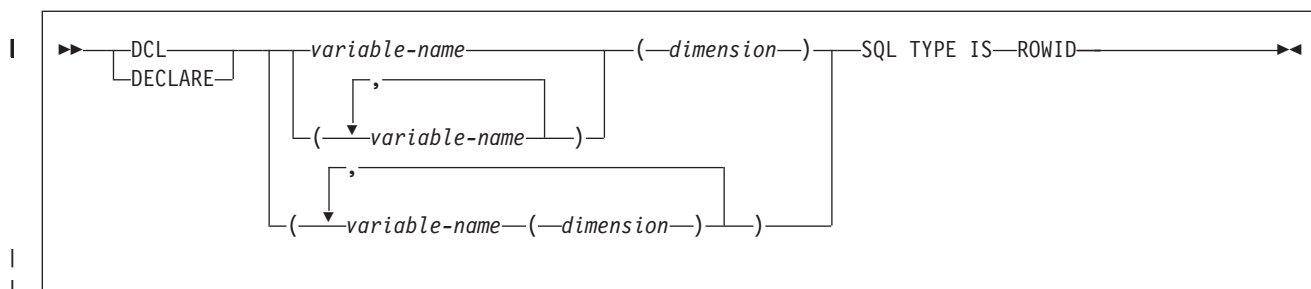


Figure 111. ROWID variable arrays

Notes:

1. *dimension* must be an integer constant between 1 and 32767.

Using host structures

A PL/I host structure name can be a structure name whose subordinate levels name scalars. For example:

```

DCL 1 A,
    2 B,
    3 C1 CHAR(...),
    3 C2 CHAR(...);
  
```

In this example, B is the name of a host structure consisting of the scalars C1 and C2.

You can use the structure name as shorthand notation for a list of scalars. You can qualify a host variable with a structure name (for example, STRUCTURE.FIELD). Host structures are limited to two levels. You can think of a host structure for DB2 data as a named group of host variables.

You must terminate the host structure variable by ending the declaration with a semicolon. For 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.

Figure 112 on page 241 shows the syntax for declarations of host structures.

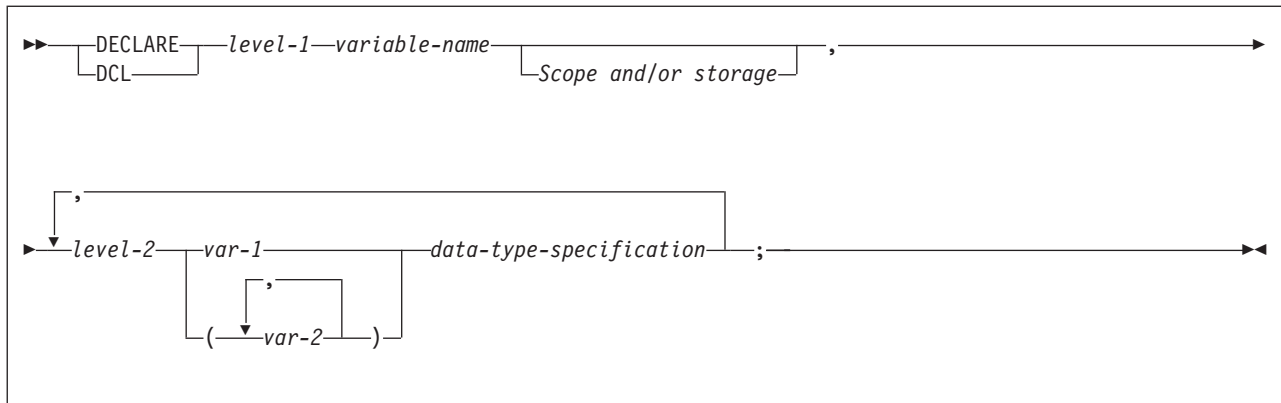


Figure 112. Host structures

Figure 113 shows the syntax for data types that are used within declarations of host structures.

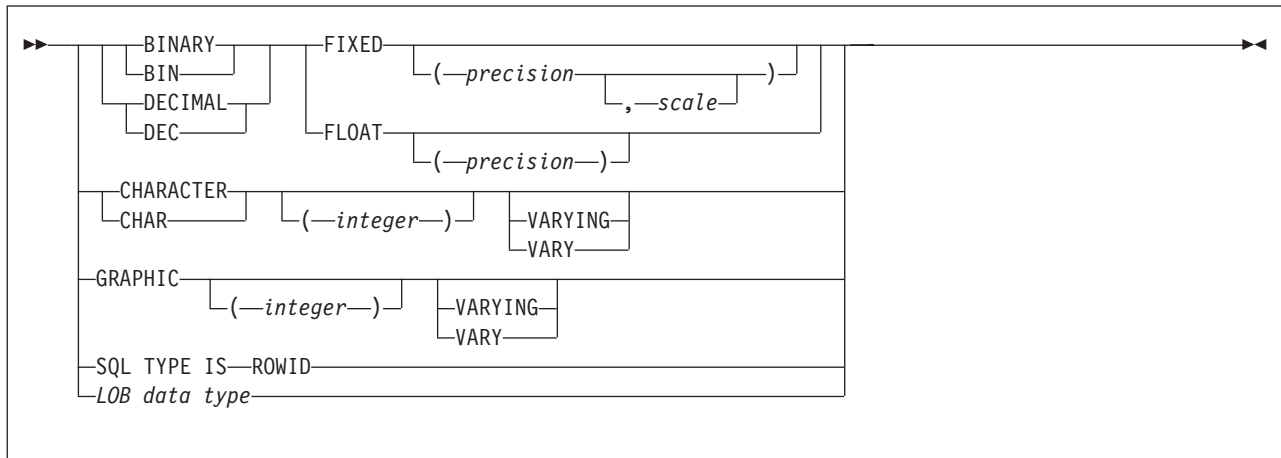


Figure 113. Data type specification

Figure 114 shows the syntax for LOB data types that are used within declarations of host structures.

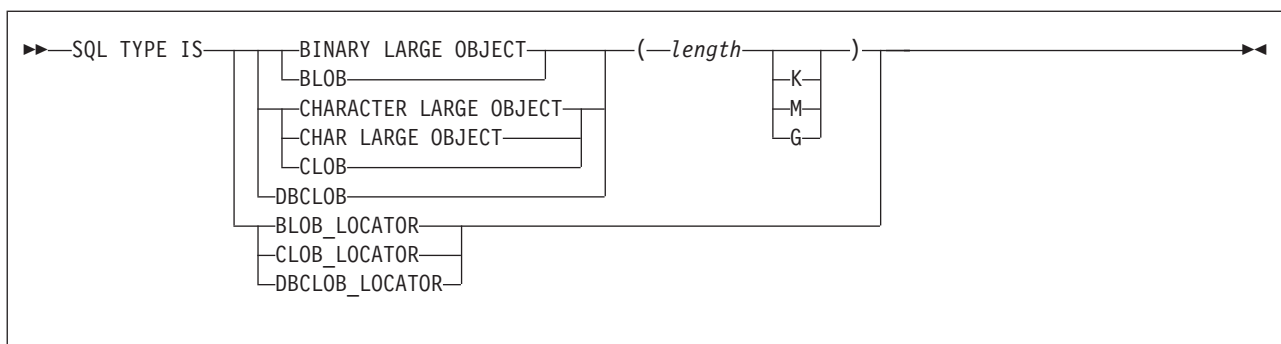


Figure 114. LOB data type

Determining equivalent SQL and PL/I data types

Table 20 on page 242 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 20. SQL data types the precompiler uses for PL/I declarations

PL/I data type	SQLTYPE of host variable	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
DEC FIXED(<i>p,s</i>) 0<= <i>p</i> <=31 and 0<= <i>s</i> <= <i>p</i> ¹	484	<i>p</i> in byte 1, <i>s</i> in byte 2	DECIMAL(<i>p,s</i>)
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(<i>n</i>) VARYING 1<= <i>n</i> <=127	464	<i>n</i>	VARGRAPHIC(<i>n</i>)
GRAPHIC(<i>n</i>) VARYING <i>n</i> >127	472	<i>n</i>	VARGRAPHIC(<i>n</i>)
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
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 ³	412	<i>n</i>	DBCLOB(<i>n</i>) ³
SQL TYPE IS ROWID	904	40	ROWID

Notes:

1. 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).
2. Do not use this data type as a column type.
3. *n* is the number of double-byte characters.

Table 21 on page 243 helps you define host variables that receive output from the database. You can use the table to determine the PL/I data type that is equivalent to a given SQL data type. For example, if you retrieve **TIMESTAMP** data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 21 shows direct conversions between DB2 data types and host data types. However, a number of DB2 data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 does conversions between those compatible data types. See Table 1 on page 5 for information on compatible data types.

Table 21. SQL data types mapped to typical PL/I declarations

SQL data type	PL/I equivalent	Notes
SMALLINT	BIN FIXED(<i>n</i>)	$1 \leq n \leq 15$
INTEGER	BIN FIXED(<i>n</i>)	$16 \leq n \leq 31$
DECIMAL(<i>p,s</i>) or NUMERIC(<i>p,s</i>)	If $p < 16$: DEC FIXED(<i>p</i>) or DEC FIXED(<i>p,s</i>)	<i>p</i> is precision; <i>s</i> is scale. $1 \leq p \leq 31$ and $0 \leq s \leq p$ If $p > 15$, the PL/I compiler must support 31-digit decimal variables.
REAL or FLOAT(<i>n</i>)	BIN FLOAT(<i>p</i>) or DEC FLOAT(<i>m</i>)	$1 \leq n \leq 21$, $1 \leq p \leq 21$, and $1 \leq m \leq 6$
DOUBLE PRECISION, DOUBLE, or FLOAT(<i>n</i>)	BIN FLOAT(<i>p</i>) or DEC FLOAT(<i>m</i>)	$22 \leq n \leq 53$, $22 \leq p \leq 53$, and $7 \leq m \leq 16$
CHAR(<i>n</i>)	CHAR(<i>n</i>)	$1 \leq n \leq 255$
VARCHAR(<i>n</i>)	CHAR(<i>n</i>) VAR	
GRAPHIC(<i>n</i>)	GRAPHIC(<i>n</i>)	<i>n</i> refers to the number of double-byte characters, not to the number of bytes. $1 \leq n \leq 127$
VARGRAPHIC(<i>n</i>)	GRAPHIC(<i>n</i>) VAR	<i>n</i> refers to the number of double-byte characters, not to the number of bytes.
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.
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.

Table 21. SQL data types mapped to typical PL/I declarations (continued)

SQL data type	PL/I 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
ROWID	SQL TYPE IS ROWID	

Notes on PL/I variable declaration and usage

You should be aware of the following when you declare PL/I variables.

PL/I data types with no SQL equivalent: 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.

SQL data types with no PL/I equivalent: If the PL/I compiler you are using does not support a decimal data type with a precision greater than 15, use the following types of variables 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 choose integer, 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, you can use floating-point numbers. Floating-point numbers are approximations of real numbers. When you assign a decimal number to a floating-point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

Floating-point host variables: All floating-point data is stored in DB2 in System/390 hexadecimal floating-point format. However, your host variable data can be in System/390 hexadecimal floating-point format or IEEE binary floating-point format. DB2 uses the FLOAT precompiler option to determine whether your floating-point host variables are in IEEE binary floating-point format or System/390 hexadecimal floating-point format. DB2 does no checking to determine whether the host variable declarations or format of the host variable contents match the precompiler option. Therefore, you need to ensure that your floating-point host variable types and contents match the precompiler option.

Special purpose PL/I data types: The locator data types are PL/I data types as well as SQL data types. You cannot use locators as column types. For information on how to use these data types, see the following sections:

Result set locator

Chapter 25, "Using stored procedures for client/server processing," on page 629

Table locator "Accessing transition tables in a user-defined function or stored procedure" on page 345

LOB locators Chapter 14, "Programming for large objects," on page 297

PL/I scoping rules: The precompiler does not support PL/I scoping rules.

Overflow: 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.

Truncation: 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.

Determining compatibility of SQL and PL/I data types

When you use PL/I host variables in SQL statements, the variables must be type compatible with the columns with which you use them.

- Numeric data types are compatible with each other. A SMALLINT, INTEGER, 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. 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. 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.

- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length PL/I character host variable.
- A BLOB column or a BLOB locator is compatible only with a BLOB 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. For information on assigning and comparing distinct types, see Chapter 16, “Creating and using distinct types,” on page 367.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using indicator variables and indicator variable arrays

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 use indicator variables and indicator variable arrays in similar ways.

Using indicator variables: 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 you find in X is irrelevant.

When your program uses 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.

Using indicator variable arrays: When you retrieve data into a host variable array, if a value in its indicator array is negative, you can disregard the contents of the corresponding element in the host variable array. For more information about indicator variable arrays, see “Using indicator variable arrays with host variable arrays” on page 87.

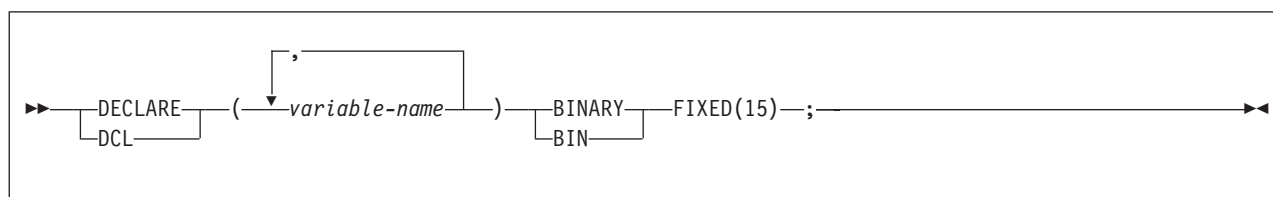
Declaring indicator variables: You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see “Using indicator variables with host variables” on page 83.

Example: The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement:

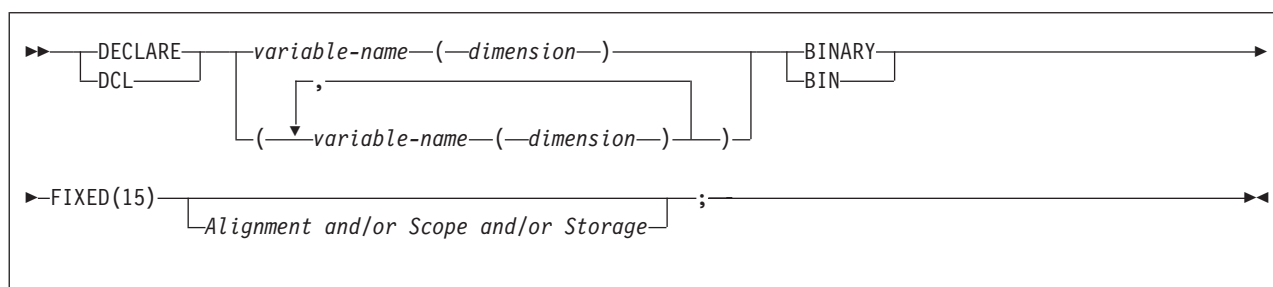
```
EXEC SQL FETCH CLS_CURSOR INTO :CLS_CD,
                                :DAY :DAY_IND,
                                :BGN :BGN_IND,
                                :END :END_IND;
```

You can declare the variables as follows:

Figure 115 shows the syntax for declarations of indicator variables.



Declaring indicator arrays: Figure 116 shows the syntax for declarations of indicator arrays.



Notes:

1. *dimension* must be an integer constant between 1 and 32767.

Handling SQL error return codes

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 “Calling DSNTIAR to display SQLCA fields” on page 98.

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 “The GET DIAGNOSTICS statement” on page 94.

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 DSN8810.SDSNSAMP. See Appendix B, "Sample applications," on page 1013 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.

Coding SQL statements in a REXX application

This section helps you with the programming techniques that are unique to coding SQL statements in a REXX procedure. For an example of a complete DB2 REXX procedure, see “Sample DB2 REXX application” on page 1047.

Defining the SQL communication area

When DB2 prepares a REXX procedure that contains SQL statements, DB2 automatically includes an SQL communication area (SQLCA) in the procedure. The REXX SQLCA differs from the SQLCA for other languages in the following ways:

- The REXX SQLCA consists of a set of separate variables, rather than a structure.
If you use the ADDRESS DSNREXX 'CONNECT' *ssid* syntax to connect to DB2, the SQLCA variables are a set of simple variables.
If you connect to DB2 using the CALL SQLDBS 'ATTACH T0' syntax, the SQLCA variables are compound variables that begin with the stem SQLCA.
See “Accessing the DB2 REXX Language Support application programming interfaces” on page 250 for a discussion of the methods for connecting a REXX application to DB2.
- You cannot use the INCLUDE SQLCA statement to include an SQLCA in a REXX program.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variable values to determine whether the last SQL statement was successful.

See Appendix D of *DB2 SQL Reference* for information on the fields in the REXX SQLCA.

Defining SQL descriptor areas

The following statements require an SQL descriptor area (SQLDA):

- 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...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- PREPARE...INTO *descriptor-name*

A REXX procedure can contain more than one SQLDA. 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. DB2 does not support the INCLUDE SQLDA statement in REXX.

See Appendix E of *DB2 SQL Reference* for information on the fields in a REXX SQLDA.

Accessing the DB2 REXX Language Support application programming interfaces

DB2 REXX Language Support includes the following application programming interfaces:

CONNECT

Connects the REXX procedure to a DB2 subsystem. You must execute CONNECT before you can execute SQL statements. The syntax of CONNECT is:

```

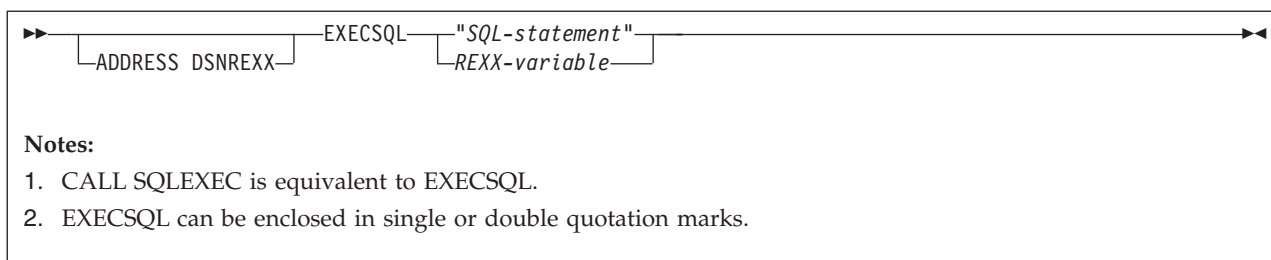
  >> _____ 'CONNECT' _____ <<
      | ADDRESS DSNREXX |             | 'subsystem-ID' |
      |                 |             | REXX-variable |

```

Note: CALL SQLDBS 'ATTACH TO' *ssid* is equivalent to ADDRESS DSNREXX 'CONNECT' *ssid*.

EXECSQL

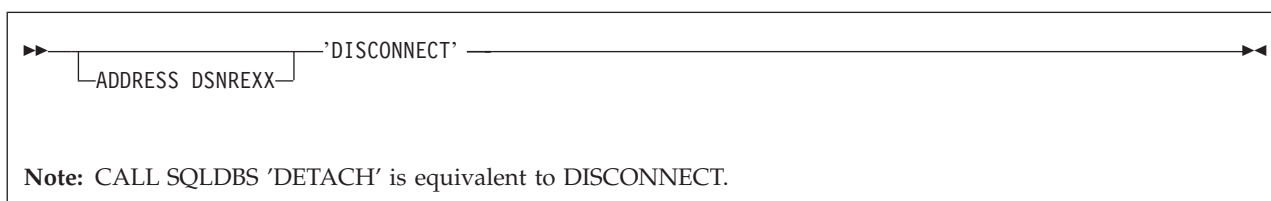
Executes SQL statements in REXX procedures. The syntax of EXECSQL is:



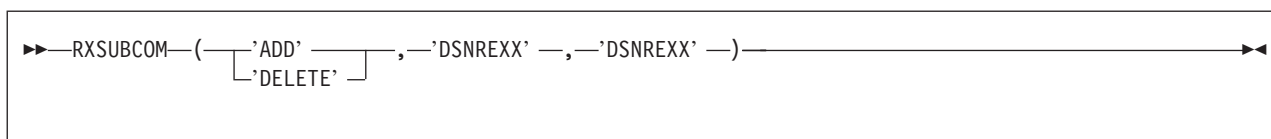
See “Embedding SQL statements in a REXX procedure” on page 252 for more information.

DISCONNECT

Disconnects the REXX procedure from a DB2 subsystem. You should execute DISCONNECT to release resources that are held by DB2. The syntax of DISCONNECT is:



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:



The ADD function adds DSNREXX to the REXX host command environment table. The DELETE function deletes DSNREXX from the REXX host command environment table.

Figure 117 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')
ADDRESS DSNREXX           /* ADD HOST CMD ENVIRONMENT */
                          /* SEND ALL COMMANDS OTHER */
                          /* THAN REXX INSTRUCTIONS TO */
                          /* DSNREXX */
                          /* CALL CONNECT, EXEC SQL, AND */
                          /* DISCONNECT INTERFACES */
:
S_RC = RXSUBCOM('DELETE', 'DSNREXX', 'DSNREXX')
                          /* WHEN DONE WITH */
                          /* DSNREXX, REMOVE IT. */

```

Figure 117. Making DSNREXX available to an application

Embedding SQL statements in a REXX procedure

You can code SQL statements in a REXX procedure wherever you can use REXX commands. DB2 REXX Language Support allows all SQL statements that DB2 UDB for z/OS supports, **except** the following statements:

- BEGIN DECLARE SECTION
- DECLARE STATEMENT
- END DECLARE SECTION
- INCLUDE
- SELECT INTO
- WHENEVER

Each SQL statement in a REXX procedure 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 procedure:

```
EXECSQL "COMMIT"
rexxvar="COMMIT"
EXECSQL rexxvar
```

You cannot execute a SELECT, INSERT, UPDATE, 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 “Using REXX host variables and data types” on page 254 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 procedure.

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 DSN8810.DEPT" ,
    "SET MGRNO = '000010'" ,
    "WHERE DEPTNO = 'D11'"
"EXECSQL " || ,
"  UPDATE DSN8810.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 procedure.

Margins: Like REXX commands, SQL statements can begin and end anywhere on a line.

Names: 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 procedure. 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. Table 22 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 22. 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" on page 250.

Using cursors and statement names

In REXX SQL applications, 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 SQL 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.

Using REXX host variables and data types

You do not declare host variables in REXX. When you need a new variable, you use it in a REXX command. When you use a REXX variable as a host variable in an SQL statement, you must precede the variable with a colon.

A REXX host variable can be a simple or compound variable. DB2 REXX Language Support evaluates compound variables before DB2 processes SQL statements that contain the variables. In the following example, the host variable that is passed to DB2 is :x.1.2:

```
a=1
b=2
EXECSQL 'OPEN C1 USING :x.a.b'
```

Determining equivalent SQL and REXX data types

All REXX data is string data. Therefore, when a REXX procedure assigns input data to a table column, DB2 converts the data from a string type to the table column type. When a REXX procedure 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.

Letting DB2 determine the input data type

You can let DB2 assign a data type to input data based on the format of the input string. Table 23 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.

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.

Table 23. 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 -2147483647 and 2147483647, inclusive.

Table 23. SQL input data types and REXX data formats (continued)

SQL data type assigned by DB2	SQLTYPE for data type	REXX input data format
DECIMAL(<i>p,s</i>)	484/485	One of the following formats: - 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 -2147483647 or greater than 2147483647.
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.
VARCHAR(<i>n</i>)	448/449	One of the following formats: - 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	One of the following formats: - 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.

Ensuring that DB2 correctly interprets character input data

To ensure that DB2 REXX Language Support does not interpret character literals as graphic or numeric literals, precede and follow character literals with a double quotation mark, followed by a single quotation mark, followed by another double quotation mark ("'").

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 host variable stringvar to DB2. The value that you want to pass is the string '100'. The first thing that you need to do is to assign the string to the host variable. You might write a REXX command like this:

```
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 command like this:

```
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 desired result.

Passing the data type of an input variable to DB2

In some cases, you might want to determine the data type of input data for DB2. For example, 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 indicate the data type of input data to DB2, use an SQLDA.

Example: Specifying CHAR: Suppose 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:

```

INSQLDA.SQDL = 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: Specifying DECIMAL with precision and scale: Suppose 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:

```

INSQLDA.SQDL = 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.SQSCALE = 8     /* Scale of variable is 8 */
INSQLDA.1.SQDATA = 9876543210.87654321 /* Value in variable */
```

Retrieving data from DB2 tables

Although all output data is string data, 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. Table 24 on page 257 gives the format for each type of output data.

Table 24. SQL output data types and REXX data formats

SQL data type	REXX output data format
SMALLINT INTEGER	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 -2147483647 and 2147483647, inclusive.
DECIMAL(<i>p,s</i>)	A string of numerics with one of the following formats: - 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 -2147483647 or greater than 2147483647. 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.
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
```

Using indicator variables

When you retrieve a null value from a column, DB2 puts a negative value in an indicator variable to indicate that the data in the corresponding host variable is null. When you pass a null value to DB2, you assign a negative value to an indicator variable to indicate that the corresponding host variable has a null value.

The way that you use indicator variables for input host variables in REXX procedures is slightly different from 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, to set a value of WORKDEPT in table EMP to null, use statements like these:

```
SQLSTMT="UPDATE EMP" ,
      "SET WORKDEPT = ?"
HVWORKDEPT='000'
INDWORKDEPT=-1
"EXECSQL PREPARE S100 FROM :SQLSTMT"
"EXECSQL EXECUTE S100 USING :HVWORKDEPT :INDWORKDEPT"
```

After you retrieve data from a column that can contain null values, you should always check the indicator variable that corresponds to the output host variable for that column. If the indicator variable value is negative, the retrieved value is null, so you can disregard the value in the host variable.

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 DSN8810.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')
```

Setting the isolation level of SQL statements in a REXX procedure

When you install DB2 REXX Language Support, you bind four packages for accessing DB2, each with a different isolation level:

Package name	Isolation level
DSNREXRR	Repeatable read (RR)
DSNREXRS	Read stability (RS)
DSNREXCS	Cursor stability (CS)
DSNREXUR	Uncommitted read (UR)

To change the isolation level for SQL statements in a REXX procedure, execute the SET CURRENT PACKAGESET statement to select the package with the isolation level you need. For example, to change the isolation level to cursor stability, execute this SQL statement:

```
"EXECSQL SET CURRENT PACKAGESET='DSNREXCS' "
```

Chapter 10. Using constraints to maintain data integrity

When you modify DB2 tables, you need to ensure that the data is valid. DB2 provides two ways to help you maintain valid data: *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 “Using check constraints.”
- Referential constraints preserve relationships between tables. Referential constraints are discussed in “Using referential constraints” on page 261. A specific type of referential constraints, the informational referential constraint, is discussed in “Using informational referential constraints” on page 267.

Triggers are a series of actions that are invoked when a table is updated. Triggers are discussed in Chapter 12, “Using triggers for active data,” on page 277.

Using check constraints

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 statement
- An SQL UPDATE statement

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.

How check constraints set CHECK-pending status

Maintaining check integrity requires enforcing check constraints on data in a table. When check integrity is compromised or cannot be guaranteed, the table space or partition that contains the table is placed in CHECK-pending status. The definition of that status includes violations of check constraints as well as referential constraints.

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.

Using referential constraints

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.

Figure 118 shows the relationships that exist among the tables in the sample application. Arrows point from parent tables to dependent tables.

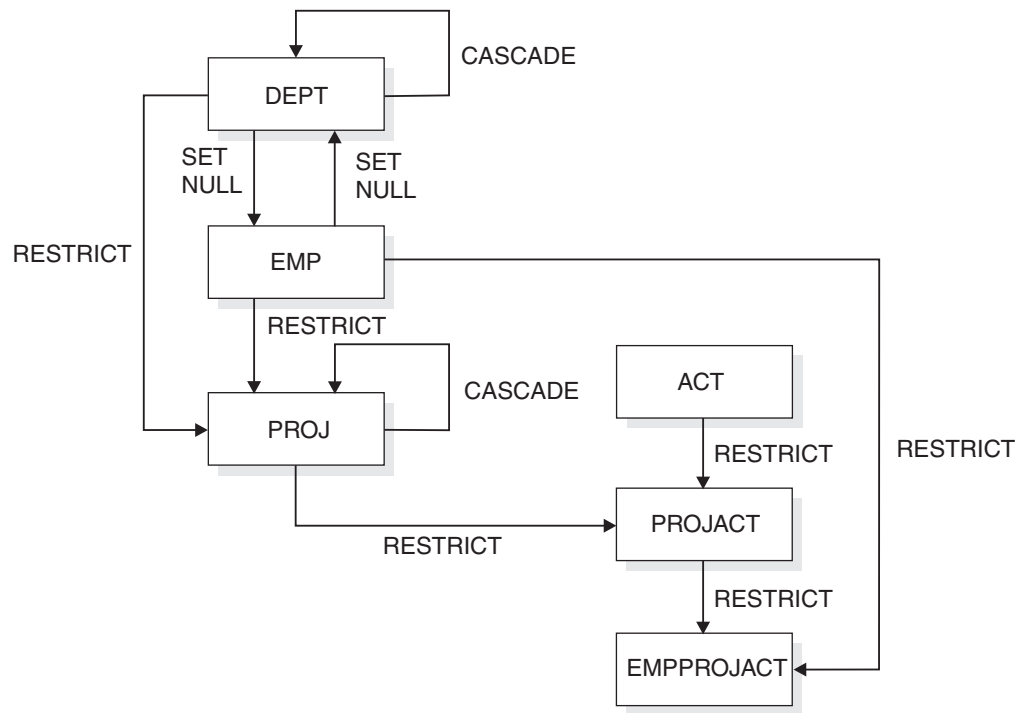


Figure 118. Relationships among tables in the sample application

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.

Parent key columns

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 25 shows part of the project table which has the primary key column, PROJNO.

Table 25. 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 26 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 26. 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 parent key and a unique index

The primary key of a table, if one exists, uniquely identifies each occurrence of an entity about which the table contains information. 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 create a table using the UNIQUE clause of the CREATE TABLE or ALTER TABLE statement. Like the PRIMARY KEY clause, specifying a UNIQUE clause prevents use of the table until you create an index to enforce the uniqueness of the key. If you use the UNIQUE clause in an ALTER TABLE statement, a unique index must already exist. For more information about the UNIQUE clause, see Chapter 5 of *DB2 SQL Reference*.

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, read Chapter 10, “Using constraints to maintain data integrity,” on page 259 before creating or altering any of the tables involved.

A table can have no more than one primary key. A primary key obeys the same restrictions as do index keys:

- The key can include no more than 64 columns.
- No column can be named 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.

Incomplete definition

If a table is created with a primary key, its *primary index* is the first unique index created on its primary key columns, with the same order of columns as the primary key columns. The columns of the primary index can be in either ascending or descending order. The table has an incomplete definition until you create an index on the parent key. This incomplete definition status is recorded as a P in the TABLESTATUS column of SYSIBM.SYSTABLES. Use of a table with an incomplete definition is severely restricted: you can drop the table, create the primary index, and drop or create other indexes; you cannot load the table, insert data, retrieve data, update data, delete data, or create foreign keys that reference the primary key.

Because of these restrictions, plan to create the primary index soon after creating the table. For example, to create the primary index for the project activity table, issue:

```
CREATE UNIQUE INDEX XPROJAC1  
  ON DSN8810.PROJACT (PROJNO, ACTNO, ACSTDATE);
```

Creating the primary index resets the incomplete definition status and its associated restrictions. But if you drop the primary index, it reverts to incomplete definition status; to reset the status, you must create the primary index or alter the table to drop the primary key.

If the primary key is added later with ALTER TABLE, a unique index on the key columns must already exist. If more than one unique index is on those columns, DB2 chooses one arbitrarily to be the primary index.

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.

Defining a foreign key

You define a list of columns as a foreign key of a table with the FOREIGN KEY clause in the CREATE TABLE statement.

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.

The column names you specify identify the columns of the parent key. The privilege set must include the ALTER or the REFERENCES privilege on the columns of the parent key. A unique index must exist on the parent key columns of the parent table.

The relationship name

You can 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. Figure 119 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 DSN8810.EMPPROJECT
  (EMPNO      CHAR(6)          NOT NULL,
   PROJNO     CHAR(6)          NOT NULL,
   ACTNO      SMALLINT         NOT NULL,
   CONSTRAINT REPAPA FOREIGN KEY (PROJNO, ACTNO)
     REFERENCES DSN8810.PROJECT ON DELETE RESTRICT,
   CONSTRAINT REPAE FOREIGN KEY (EMPNO)
     REFERENCES DSN8810.EMP ON DELETE RESTRICT)
IN DATABASE DSN8D81A;
```

Figure 119. Specifying constraint names for foreign keys

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 beginning with the same column (otherwise DB2 generates the name).

Indexes on foreign keys

Although not required, an index on a foreign key is strongly recommended if rows of the parent table are often deleted. The validity of the delete statement, and its possible effect on the dependent table, can be checked through the index.

You can create an index on the columns of a foreign key in the same way you create one on any other set of columns. Most often it is not a unique index. If you do create a unique index on a foreign key, it introduces an additional constraint on the values of the columns.

To let an index on the foreign key be used on the dependent table for a delete operation on a parent table, the columns of the index on the foreign key must be identical to and in the same order as the columns in the foreign key.

A foreign key can also be the primary key; then the primary index is also a unique index on the foreign key. In that case, every row of the parent table has at most one dependent row. The dependent table might be used to hold information that pertains to only a few of the occurrences of the entity described by the parent table. For example, a dependent of the employee table might contain information that applies only to employees working in a different country.

The primary key can share columns of the foreign key if the first n columns of the foreign key are the same as the primary key's columns. Again, the primary index serves as an index on the foreign key. In the sample project activity table, the primary index (on PROJNO, ACTNO, ACSTDATE) serves as an index on the foreign key on PROJNO. It does not serve as an index on the foreign key on ACTNO, because ACTNO is not the first column of the index.

The FOREIGN KEY clause in ALTER TABLE

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.

Restrictions on cycles of dependent tables

A *cycle* is a set of two or more tables that can be 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. 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 Figure 120 on page 266. 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.

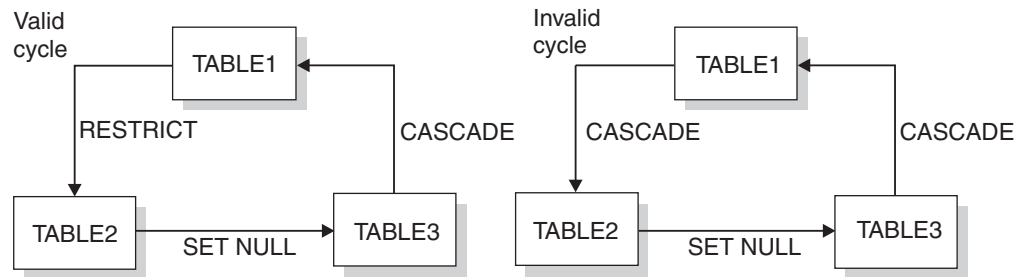


Figure 120. Valid and invalid delete cycles

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.

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 (INSERT statement) clause on a SELECT 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. See *DB2 SQL Reference* for more information about the SET ENCRYPTION PASSWORD statement and the ENCRYPT_TDES statement.

Referential constraints on tables with multilevel security with row-level granularity

The multilevel security check with row-level granularity is not enforced when DB2 is enforcing referential constraints. Although referential constraints cannot be used for a security label column, referential constraints can be used on other columns in the row. Referential constraints are enforced when the following situations occur:

- An INSERT statement is applied to a dependent table.
- An UPDATE statement is applied to a foreign key of a dependent table, or to the parent key of a parent table.
- A DELETE statement 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.

Refer to Part 3 (Volume 1) of *DB2 Administration Guide* for more information about multilevel security with row-level granularity.

Using informational referential constraints

An informational referential constraint is a referential constraint that is not enforced by DB2 during normal operations. 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. For more information about the NOT ENFORCED option, see Chapter 5 of *DB2 SQL Reference*.

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. For more information about materialized query tables and automatic query rewrite, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Chapter 11. Using DB2-generated values as keys

This chapter discusses how to use DB2-generated values as keys in applications:

- “Using ROWID columns as keys”
- “Using identity columns as keys” on page 270
- “Using values obtained from sequence objects as keys” on page 273

The chapter describes the characteristics of values in ROWID and identity columns and values that are obtained by referencing sequence objects. This chapter also evaluates the ability of these DB2-generated values to serve as primary keys or unique keys.

Using 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 allow you 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 section 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.

Defining a ROWID column

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.

For more information, see “Inserting data into a ROWID column” on page 30.

Direct row access

For some applications, you can use the value of a ROWID column to navigate directly to a row. 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. For more information about predicates and direct row access, see “Is direct row access possible? (PRIMARY_ACCESSTYPE = D)” on page 803.

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. For identity columns that are defined as GENERATED BY DEFAULT and NO CYCLE, only the values that DB2 generates are unique among each other.
- 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. In addition, to ensure that the values are unique, define a unique index on that column.

For more information, see “Inserting data into an identity column” on page 30.

The values that DB2 generates for an identity column depend on how the column is defined. The START WITH parameter determines the first value that DB2 generates. The values advance by the INCREMENT BY parameter in ascending or descending order.

The MINVALUE and MAXVALUE parameters determine the minimum and maximum values that DB2 generates. The CYCLE or NO CYCLE parameter 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: 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 INSERTs 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.

Parent keys and foreign keys

The SELECT from INSERT statement allows 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. For information about the SELECT from INSERT statement, see “Selecting values as you insert: SELECT FROM INSERT” on page 31.

In addition, you can use the IDENTITY_VAL_LOCAL function to return the most recently assigned value for an identity column that was generated by an INSERT with a VALUES clause within the current processing level. (A new level is initiated when a trigger, function, or stored procedure is invoked.)

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;
```

Example: Using IDENTITY_VAL_LOCAL: The following INSERT and UPDATE statements are equivalent to the INSERT and UPDATE statements of the previous example:

```
INSERT INTO EMPLOYEE (NAME, SALARY, WORKDEPT)
  VALUES ('New Employee', 75000.00, 11);

UPDATE DEPARTMENT
  SET MGRNO = IDENTITY_VAL_LOCAL()
  WHERE DEPTNO = 11;
```

The INSERT statement and the IDENTITY_VAL_LOCAL function must be at the same processing level.

Using values obtained from sequence objects as keys

A *sequence* is a user-defined object that generates a sequence of numeric values according to the specification with which the sequence was created. The sequence of numeric values is generated in a monotonically ascending or descending order. Sequences, unlike identity columns, are not associated with tables. Applications refer to a sequence object to get its current or next value. 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

Creating a sequence object

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 parameter determines the first value that DB2 generates. The values advance by the INCREMENT BY parameter in ascending or descending order.

The MINVALUE and MAXVALUE parameters determine the minimum and maximum values that DB2 generates. The CYCLE or NO CYCLE parameter 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.

Referencing a sequence object

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 statement, within the SET clause of an UPDATE statement (with certain restrictions), or within a SET host-variable statement. For more information about where you can use these expressions, see *DB2 SQL Reference*.

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.

Chapter 12. Using triggers for active data

Triggers are sets of SQL statements that execute when a certain event occurs in a DB2 table. Like constraints, triggers can be used to control changes in DB2 databases. Triggers are more powerful, however, because they can monitor a broader range of changes and perform a broader range of actions than constraints can.

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.

This chapter presents the following information about triggers:

- “Example of creating and using a trigger”
- “Parts of a trigger” on page 279
- “Invoking stored procedures and user-defined functions from triggers” on page 285
- “Passing transition tables to user-defined functions and stored procedures” on page 286
- “Trigger cascading” on page 286
- “Ordering of multiple triggers” on page 287
- “Interactions between triggers and referential constraints” on page 288
- “Interactions between triggers and tables that have multilevel security with row-level granularity” on page 289
- “Creating triggers to obtain consistent results” on page 290

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. Figure 121 on page 278 shows an example of a CREATE TRIGGER statement.

```

CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  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

```

Figure 121. Example of a trigger

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

This section gives you the information you need to code each of the trigger parts:

- “Trigger name”
- “Subject table”
- “Trigger activation time”
- “Triggering event”
- “Granularity” on page 280
- “Transition variables” on page 281
- “Transition tables” on page 282
- “Triggered action” on page 283

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 plan or package that contains the CREATE TRIGGER statement. If the bind command does not include the QUALIFIER option, DB2 uses the owner of the package or plan.
- If you use dynamic SQL to execute the CREATE TRIGGER statement, DB2 uses the authorization ID in special register CURRENT SQLID.

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. See *DB2 Utility Guide and Reference* for more information about the LOAD utility.

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. Table 27 summarizes which SQL statements you can use in which types of triggers.

Table 27. Valid SQL statements for triggers and trigger activation times

SQL statement	Valid before activation time	Valid after activation time
fullselect	Yes	Yes
CALL	Yes	Yes
SIGNAL SQLSTATE	Yes	Yes
VALUES	Yes	Yes
SET <i>transition-variable</i>	Yes	No
INSERT	No	Yes
DELETE (searched)	No	Yes

Table 27. Valid SQL statements for triggers and trigger activation times (continued)

SQL statement	Valid before activation time	Valid after activation time
UPDATE (searched)	No	Yes

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 stored procedures and user-defined functions from triggers” on page 285 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 SQLSTATE

Use the SIGNAL SQLSTATE 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 SQLSTATE statement, it returns an SQLCA to the application with SQLCODE -438. The SQLCA also includes the following values, which you supply in the SIGNAL SQLSTATE 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 SQLSTATE 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 fill in 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), and UPDATE (searched)

Because you can include INSERT, DELETE (searched), and UPDATE (searched) statements in your trigger body, execution of the trigger body might cause activation of other triggers. See “Trigger cascading” on page 286 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.

Invoking stored procedures and user-defined functions from triggers

A trigger body can include only SQL statements and built-in functions. Therefore, if you want the trigger to perform actions or use logic that is not available in SQL statements or built-in functions, you need to write a user-defined function or stored procedure and invoke that function or stored procedure from the trigger body. Chapter 15, “Creating and using user-defined functions,” on page 311 and Chapter 25, “Using stored procedures for client/server processing,” on page 629 contain detailed information on how to write and prepare user-defined functions and stored procedures.

Because a before trigger must not modify any table, functions and procedures that you invoke from a trigger cannot include INSERT, UPDATE, or DELETE statements that modify the subject table.

To invoke a user-defined function from a trigger, code a SELECT statement or VALUES statement. Use a SELECT statement to execute the function conditionally. The number of times the user-defined function executes depends on the number of rows in the result table of the SELECT statement. For example, in this trigger, the SELECT statement causes user-defined function LARGE_ORDER_ALERT to execute for each row in transition table N_TABLE with an order 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
```

Use the VALUES statement to execute a function unconditionally; that is, once for each execution of a statement trigger or once for each row in a row trigger. In this

example, user-defined function PAYROLL_LOG executes every time an update operation occurs that activates trigger PAYROLL1:

```
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 from a trigger, use a CALL statement. The parameters of this stored procedure call must be literals, transition variables, table locators, or expressions.

Passing transition tables to user-defined functions and stored procedures

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. That is, you want to pass a pointer to the old or new transition table. You do this using table locators.

Most of the code for using a table locator is in the function or stored procedure that receives the locator. “Accessing transition tables in a user-defined function or stored procedure” on page 345 explains how a function defines a table locator and uses it to receive a transition table. To pass the transition table from a trigger, specify the parameter TABLE *transition-table-name* when you invoke the function or stored procedure. This causes DB2 to pass a table locator for the transition table to the user-defined function or stored procedure. For example, this 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
```

Trigger cascading

An SQL operation that a trigger performs might modify the subject table or other tables with triggers, so DB2 also activates those triggers. 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. See Appendixes (Volume 2) of *DB2 Administration Guide* for information on how to write a monitor program.

Ordering 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 recreates 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 and 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.
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

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 statement, 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. Refer to Part 3 (Volume 1) of *DB2 Administration Guide* for a discussion about multilevel security with row-level granularity.

Creating triggers to obtain consistent results

When you create triggers and write SQL statements that activate those triggers, you need to ensure that executing those statements on the same set of data 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

Part 3. Using DB2 object-relational extensions

Chapter 13. Introduction to DB2 object-relational extensions	295
Chapter 14. Programming for large objects	297
Introduction to LOBs	297
Declaring LOB host variables and LOB locators	300
LOB materialization	305
Using LOB locators to save storage	305
Deferring evaluation of a LOB expression to improve performance	306
Indicator variables and LOB locators	308
Valid assignments for LOB locators	309
# Avoiding character conversion for LOB locators	309
Chapter 15. Creating and using user-defined functions	311
Overview of user-defined function definition, implementation, and invocation	311
Example of creating and using a user-defined scalar function	312
User-defined function samples shipped with DB2	313
Defining a user-defined function	314
Components of a user-defined function definition	314
Examples of user-defined function definitions	316
Implementing an external user-defined function	318
Writing a user-defined function	318
Restrictions on user-defined function programs	319
Coding your user-defined function as a main program or as a subprogram	319
Parallelism considerations	320
Passing parameter values to and from a user-defined function	321
Examples of receiving parameters in a user-defined function	333
Using special registers in a user-defined function	342
Using a scratchpad in a user-defined function	344
Accessing transition tables in a user-defined function or stored procedure	345
Preparing a user-defined function for execution	350
Making a user-defined function reentrant	351
Determining the authorization ID for user-defined function invocation	351
Preparing user-defined functions to run concurrently	352
Testing a user-defined function	352
Implementing an SQL scalar function	355
Invoking a user-defined function	355
Syntax for user-defined function invocation	355
Ensuring that DB2 executes the intended user-defined function	356
How DB2 chooses candidate functions	357
How DB2 chooses the best fit among candidate functions	359
How you can simplify function resolution	360
Using DSN_FUNCTION_TABLE to see how DB2 resolves a function	361
Casting of user-defined function arguments	362
What happens when a user-defined function abnormally terminates	363
Nesting SQL statements	363
Recommendations for user-defined function invocation	365
Chapter 16. Creating and using distinct types	367
Introduction to distinct types	367
Using distinct types in application programs	368
Comparing distinct types	368
Assigning distinct types	369
Assigning column values to columns with different distinct types	369
Assigning column values with distinct types to host variables	370
Assigning host variable values to columns with distinct types	370
Using distinct types in UNIONS	371

Invoking functions with distinct types	371
Combining distinct types with user-defined functions and LOBs	372

Chapter 13. Introduction to 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 and VARGRAPHIC 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 Chapter 14, “Programming for large objects,” on page 297.

- 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 Chapter 16, “Creating and using distinct types,” on page 367.

- 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 there is no built-in function. In that situation, you can define and write an *external* user-defined function.

For a detailed discussion of user-defined functions, see Chapter 15, “Creating and using user-defined functions,” on page 311.

Chapter 14. Programming for large objects

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.

This chapter presents the following information about LOBs:

- "Introduction to LOBs"
- "Declaring LOB host variables and LOB locators" on page 300
- "LOB materialization" on page 305
- "Using LOB locators to save storage" on page 305

Introduction to LOBs

Working with LOBs involves defining the LOBs to DB2, moving the LOB data into DB2 tables, then using SQL operations to manipulate the data. This chapter concentrates on manipulating LOB data using SQL statements. For information on defining LOBs to DB2, see Chapter 5 of *DB2 SQL Reference*. For information on how DB2 utilities manipulate LOB data, see Part 2 of *DB2 Utility Guide and Reference*.

These are the basic steps for defining LOBs and moving the data into DB2:

1. Define a column of the appropriate LOB type and optionally a row identifier (ROWID) column in a DB2 table. Define only one ROWID column, even if there are multiple LOB columns in the table. If you do not create a ROWID column before you define a LOB column, DB2 creates a *hidden* ROWID column and appends it as the last column of the table. For information about what hidden ROWID columns are, see the description on page 298.

The LOB column holds information about the LOB, not the LOB data itself. The table that contains the LOB information is called the *base table*. DB2 uses the ROWID column to locate your LOB data. You can define the LOB column (and optionally the ROWID column) in a CREATE TABLE or ALTER TABLE statement.

You can add both a LOB column and a ROWID column to an existing table by using two ALTER TABLE statements: add the ROWID column with the first ALTER TABLE statement and the LOB column with the second. If you add a LOB column first, DB2 generates a hidden ROWID column.

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.

2. Create a table space and table to hold the LOB data.

The table space and table are called a LOB table space and an auxiliary table. If your base table is nonpartitioned, you must create one LOB table space and one auxiliary table for each LOB column. If your base table is partitioned, for each LOB column, you must create one LOB table space and one auxiliary table for each partition. For example, if your base table has three partitions, you must create three LOB table spaces and three auxiliary tables for each LOB column. Create these objects using the CREATE LOB TABLESPACE and CREATE AUXILIARY TABLE statements.

3. Create an index on the auxiliary table.

Each auxiliary table must have exactly one index. Use CREATE INDEX for this task.

4. Put the LOB data into DB2.

If the total length of a LOB column and the base table row is less than 32 KB, you can use the LOAD utility to put the data in DB2. Otherwise, you must use INSERT or UPDATE statements. Even though the data is stored in the auxiliary table, the LOAD utility statement or INSERT statement specifies the base table. Using INSERT can be difficult because your application needs enough storage to hold the entire value that goes into the LOB column.

Hidden ROWID column: If you do not create a ROWID column before you define a LOB column, DB2 creates a *hidden* ROWID column for you. A hidden ROWID column is not visible in the results of SELECT * statements, including those in DESCRIBE and CREATE VIEW statements. However, it is visible to all statements that refer to the column directly. DB2 assigns the GENERATED ALWAYS attribute and the name DB2_GENERATED_ROWID_FOR_LOBS nn to a hidden ROWID column. DB2 appends the identifier nn 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.

Example: Adding a CLOB column: Suppose that you want to add a resume for each employee to the employee table. Employee resumes are no more than 5 MB in size. The employee resumes contain single-byte characters, so 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.

Execute an ALTER TABLE statement to add the ROWID column, and then execute another ALTER TABLE statement to add the CLOB column. Use statements like this:

```
ALTER TABLE EMP
  ADD ROW_ID ROWID NOT NULL GENERATED ALWAYS;
COMMIT;
ALTER TABLE EMP
  ADD EMP_RESUME CLOB(5M);
COMMIT;
```

Next, you 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. You can use statements like this:

```
CREATE LOB TABLESPACE RESUMETS
  IN DSN8D81A
  LOG NO;
COMMIT;
CREATE AUXILIARY TABLE EMP_RESUME_TAB
  IN DSN8D81A.RESUMETS
```

```

STORES DSN8810.EMP
COLUMN EMP_RESUME;
CREATE UNIQUE INDEX XEMP_RESUME
ON EMP_RESUME_TAB;
COMMIT;

```

If the value of bind option SQLRULES is STD, or if special register CURRENT RULES has been set in the program and has the value STD, DB2 creates the LOB table space, auxiliary table, and auxiliary index for you when you execute the ALTER statement to add the LOB column.

Now that your DB2 objects for the LOB data are defined, you can load your employee resumes into DB2. To do this in an SQL 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 data goes into the auxiliary table, your UPDATE statement specifies the name of the base table. The C language declaration of the host variable might be:

```
SQL TYPE is CLOB (5M) resumedata;
```

The UPDATE statement looks like this:

```

UPDATE EMP SET EMP_RESUME=:resumedata
WHERE EMPNO=:employeenum;

```

In this example, employeenum is a host variable that identifies the employee who is associated with a resume.

After your LOB data is in DB2, you can write SQL applications to manipulate the data. You can use most SQL statements with LOBs. For example, you can use statements like these 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(:resumedata, 'Department Information');

EXEC SQL SET :deptInfoEndLoc =
POSSTR(:resumedata, '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. LOB locators are discussed in "Using LOB locators to save storage" on page 305.

Sample LOB applications: Table 28 lists the sample programs that DB2 provides to assist you in writing applications to manipulate LOB data. All programs reside in data set DSN810.SDSNSAMP.

Table 28. 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 32KB 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.

For instructions on how to prepare and run the sample LOB applications, see Part 2 of *DB2 Installation Guide*.

Declaring LOB host variables and LOB locators

When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator variables to point to the LOB data. See “Using LOB locators to save storage” on page 305 for information on what LOB locators are and when you should use them instead of host variables.

You can declare LOB host variables and LOB locators in assembler, C, C++, COBOL, Fortran, and PL/I. For each host variable or locator 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 or locator in an SQL statement, you must use the variable 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. See Chapter 9, “Embedding SQL statements in host languages,” on page 143 for the syntax of LOB declarations in each language and for host language equivalents for each LOB type.

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.

- #
- #
- #
- #
- #
- #
- #
- LOB host variables that are referenced by the precompiler in SQL statements must be declared in one of the following ways:
- By using the DB2 provisions to declare LOB host variables, such as the keywords SQL TYPE IS followed by the appropriate declaration for each language.
 - By using a LOB host variable that contains a 31-bit length, followed by the data, to be compatible with the output that is generated by the precompiler.

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: Table 29 shows assembler language declarations for some typical LOB types.

Table 29. 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_var SQL TYPE IS DBCLOB_LOCATOR	dbclob_loc DS FL4
blob_loc SQL TYPE IS BLOB_LOCATOR	blob_loc 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 30. 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;

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 31. Examples of COBOL variable declarations by the DB2 precompiler

You declare this variable	DB2 precompiler generates this variable
01 BLOB-VAR USAGE IS SQL TYPE IS BLOB(1M).	01 BLOB-VAR. 02 BLOB-VAR-LENGTH PIC 9(9) COMP. 02 BLOB-VAR-DATA. 49 FILLER PIC X(32767). ¹ 49 FILLER PIC X(32767). Repeat 30 times : 49 FILLER PIC X(1048576-32*32767).
01 CLOB-VAR USAGE IS SQL TYPE IS CLOB(40000K).	01 CLOB-VAR. 02 CLOB-VAR-LENGTH PIC 9(9) COMP. 02 CLOB-VAR-DATA. 49 FILLER PIC X(32767). ¹ 49 FILLER PIC X(32767). Repeat 1248 times : 49 FILLER PIC X(40960000-1250*32767).

Table 31. Examples of COBOL variable declarations by the DB2 precompiler (continued)

You declare this variable	DB2 precompiler generates this variable
01 DBCLOB-VAR USAGE IS SQL TYPE IS DBCLOB(4000K).	01 DBCLOB-VAR. 02 DBCLOB-VAR-LENGTH PIC 9(9) COMP. 02 DBCLOB-VAR-DATA. 49 FILLER PIC G(32767) USAGE DISPLAY-1. ² 49 FILLER PIC G(32767) USAGE DISPLAY-1. Repeat 123 times : : 49 FILLER PIC G(20480000-125*32767) USAGE DISPLAY-1.
01 BLOB-LOC USAGE IS SQL TYPE IS BLOB-LOCATOR.	01 BLOB-LOC PIC S9(9) USAGE IS BINARY.
01 CLOB-LOC USAGE IS SQL TYPE IS CLOB-LOCATOR.	01 CLOB-LOC PIC S9(9) USAGE IS BINARY.
01 DBCLOB-LOC USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	01 DBCLOB-LOC PIC S9(9) USAGE IS BINARY.

Notes:

1. Because the COBOL language allows character declarations of no more than 32767 bytes, for BLOB or CLOB host variables that are greater than 32767 bytes in length, DB2 creates multiple host language declarations of 32767 or fewer bytes.
2. Because the COBOL language allows graphic declarations of no more than 32767 double-byte characters, for 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 LOB host variables in Fortran: Table 32 shows Fortran declarations for some typical LOB types.

Table 32. 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 33. 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(40960000-1250*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(40960000-250*16383);
DCL blob_loc SQL TYPE IS BLOB_LOCATOR;	DCL blob_loc FIXED BINARY(31);
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);

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 $length-m*16383$.

LOB materialization

LOB materialization means that DB2 places a LOB value into contiguous storage.
Because LOB values can be very large, DB2 avoids materializing LOB data until
absolutely necessary. However, DB2 must materialize LOBs when your application
program:

- 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
- Converts a LOB from one CCSID to another

The amount of storage that is used for LOB 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. See “Using LOB locators to save storage” for information on how to use LOB locators.

Using LOB locators to save storage

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`. For more information on `HOLD LOCATOR` and `FREE LOCATOR`, see Chapter 5 of *DB2 SQL Reference*.

Deferring evaluation of a LOB expression to improve performance

DB2 moves no bytes of a LOB value until a program assigns a LOB expression to a target destination. This means that when you use a LOB locator with string functions and operators, you can create an expression that DB2 does not evaluate until the time of assignment. This is called *deferring evaluation* of a LOB expression. Deferring evaluation can improve LOB I/O performance.

Figure 122 on page 307 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 Figure 122 on page 307 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');

```

Figure 122. Example of deferring evaluation of LOB expressions (Part 1 of 2)

```

/*****
/* 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);

/*****
/* Free the locators
/*****
EXEC SQL FREE LOCATOR :HV_DOC_LOCATOR1, :HV_DOC_LOCATOR2, :HV_DOC_LOCATOR3;

```

Figure 122. Example of deferring evaluation of LOB expressions (Part 2 of 2)

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.

Indicator variables and LOB locators

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 for assigning data to and retrieving data from LOB columns, you can also use LOB locators to assign data to CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC columns. However, you cannot fetch data from CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC columns into LOB locators.

Avoiding character conversion for LOB locators

You can use a VALUES INTO or SET statement to obtain the results of functions that operate on LOB locators, such as LEN 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.

Example: 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;
```

Chapter 15. Creating and using user-defined functions

A user-defined function is an extension to the SQL language. A user-defined function is 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 a user-defined function in an SQL statement.

This chapter presents the following information about user-defined functions:

- “Overview of user-defined function definition, implementation, and invocation”
- “Defining a user-defined function” on page 314
- “Implementing an external user-defined function” on page 318
- “Implementing an SQL scalar function” on page 355
- “Invoking a user-defined function” on page 355

This chapter contains information that applies to all user-defined functions and specific information about user-defined functions in languages other than Java. For information about writing, preparing, and running Java user-defined functions, see *DB2 Application Programming Guide and Reference for Java*.

Overview of user-defined function definition, implementation, and invocation

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

External user-defined functions can be user-defined scalar functions or user-defined table functions. Sourced and SQL user-defined functions can only be user-defined scalar functions.

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 Figure 123 on page 312. 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 stored. 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. See Chapter 25, “Using stored procedures for client/server processing,” on page 629 for this information.

- 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*.

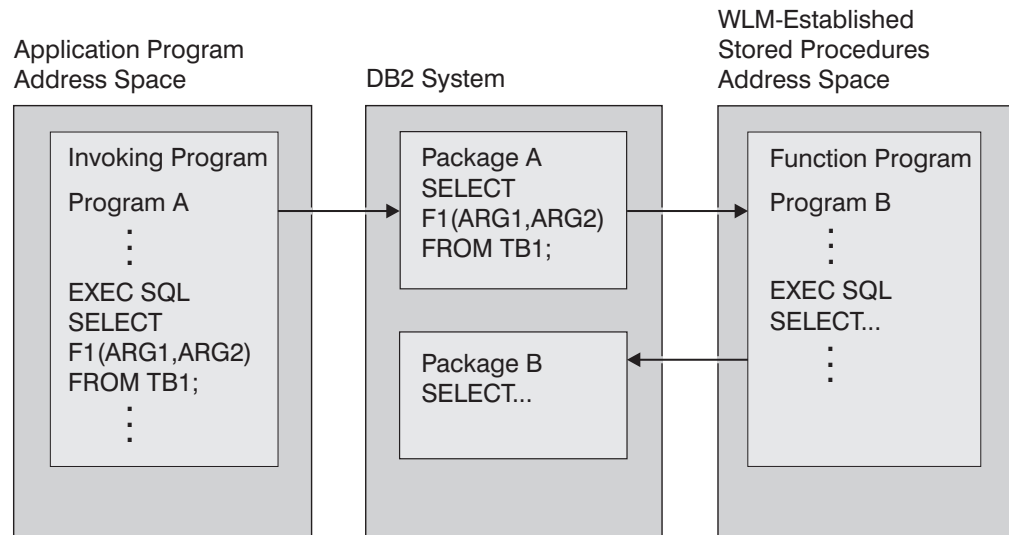


Figure 123. The user-defined function environment

Example of creating and using a user-defined scalar function

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 shipped with DB2

To assist you in defining, implementing, and invoking your user-defined functions, DB2 provides a number of sample user-defined functions. All user-defined function code is in data set DSN810.SDSNSAMP.

Table 34 summarizes the characteristics of the sample user-defined functions.

Table 34. User-defined function samples shipped with DB2

User-defined function name	Language	Member that contains source code	Purpose
ALTDAT ¹	C	DSN8DUAD	Converts the current date to a user-specified format
ALTDAT 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 ALTDAT has one input parameter, of type VARCHAR(13).
2. This version of ALTDAT 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.

Defining a user-defined function

To define a user-defined function, perform the following steps:

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. For information about the characteristics that you can define for a user-defined function, see “Components of a user-defined function definition.”
2. Execute a CREATE FUNCTION statement to register the information in the DB2 catalog. For examples of CREATE FUNCTION statements, see “Examples of user-defined function definitions” on page 316.

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. See Chapter 5 of *DB2 SQL Reference* for information about which characteristics you can change with ALTER FUNCTION.

Components of a user-defined function definition

The characteristics you include in a CREATE FUNCTION or ALTER FUNCTION statement depend on whether the user-defined function is sourced, external, or SQL. Table 35 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 35. 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	FUNCTION	Yes	Yes	Yes
Input parameter types and encoding schemes	FUNCTION	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 not deterministic	NOT DETERMINISTIC DETERMINISTIC	No	Yes	Yes

Table 35. 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?
Types of SQL statements in the function	NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA	No	Yes ⁵	Yes ⁶
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	RETURN <i>expression</i>	No	No	Yes
Encoding scheme for all string parameters	PARAMETER CCSID EBCDIC PARAMETER CCSID ASCII PARAMETER CCSID UNICODE	No	Yes	Yes

Table 35. 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

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. For additional information, see “Special techniques to influence access path selection” on page 774.
- | 2. An SQL user-defined function can return only one parameter.
- | 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.

For a complete explanation of the parameters in a CREATE FUNCTION or ALTER FUNCTION statement, see Chapter 5 of *DB2 SQL Reference*.

Examples of user-defined function definitions

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;
```

Implementing an external user-defined function

This section discusses these steps in implementing an external user-defined function:

- “Writing a user-defined function”
- “Preparing a user-defined function for execution” on page 350
- “Testing a user-defined function” on page 352

Writing a user-defined function

A user-defined function is similar to any other SQL program. When you write a user-defined function, you can include static or dynamic SQL statements, IFI calls, and DB2 commands issued through IFI calls.

Your user-defined function can also access remote data using the following methods:

- DB2 private protocol access using three-part names or aliases for three-part names

- DRDA[®] access using three-part names or aliases for three-part names
- DRDA access using CONNECT or SET CONNECTION statements

The user-defined function and the application that calls it can access the same remote site if both use the same protocol.

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 *DB2 Application Programming Guide and Reference for Java*.

The following sections include additional information that you need when you write a user-defined function:

- “Restrictions on user-defined function programs”
- “Coding your user-defined function as a main program or as a subprogram”
- “Parallelism considerations” on page 320
- “Passing parameter values to and from a user-defined function” on page 321
- “Examples of receiving parameters in a user-defined function” on page 333
- “Using special registers in a user-defined function” on page 342
- “Using a scratchpad in a user-defined function” on page 344
- “Accessing transition tables in a user-defined function or stored procedure” on page 345

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.

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 desired result.

- **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 not 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 nondeterministic 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:

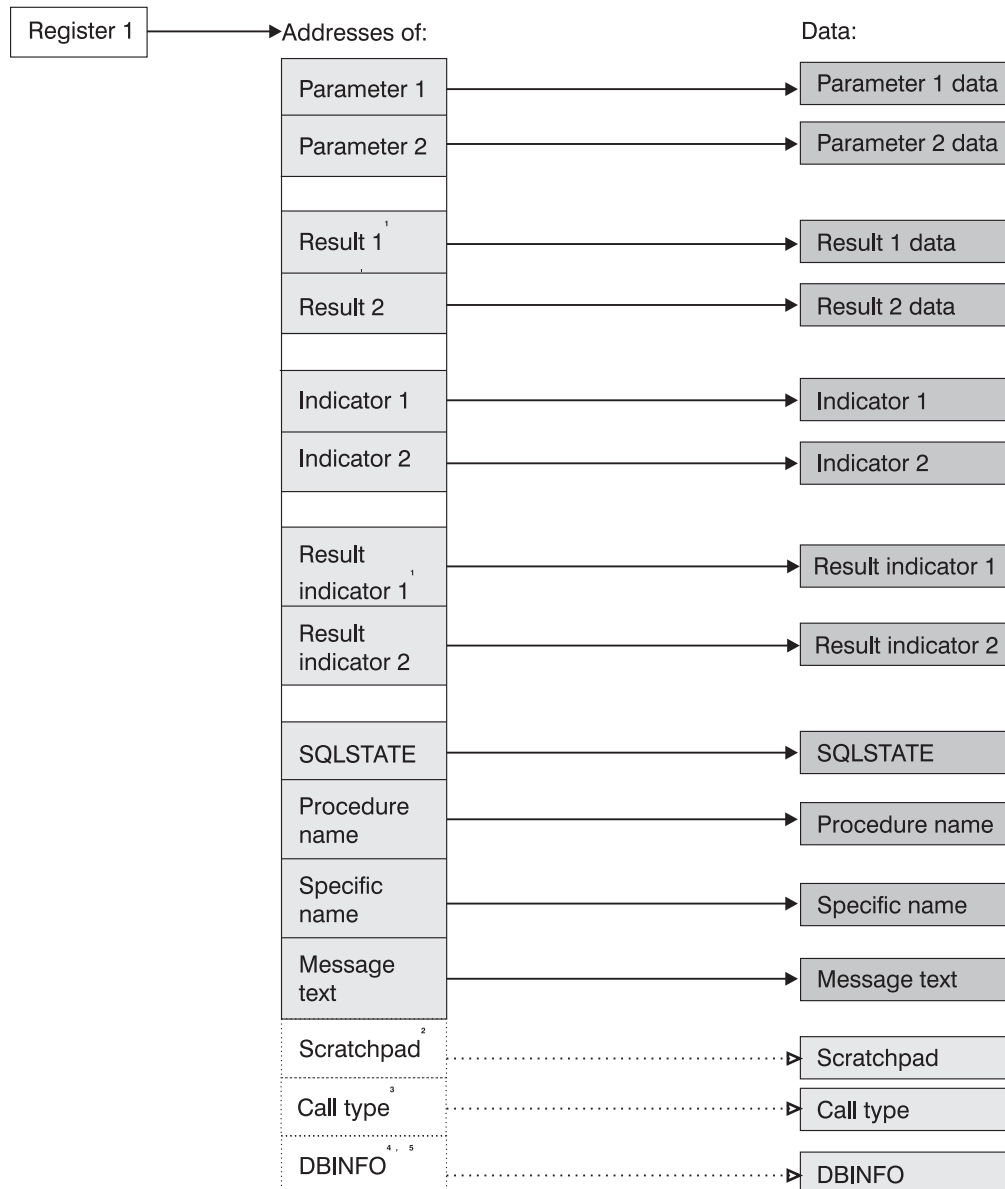
- When parallel task 1 executes, C1 has values 1, 2, and 3, and COUNTER returns values 1, 2, and 3, so the query returns values 1, 2, and 3.
- When parallel task 2 executes, C1 has values 4, 5, and 6, but COUNTER returns values 1, 2, and 3, so the query returns no rows.
- When parallel task 3, executes, 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.

Passing parameter values to and from a user-defined function

To receive parameters from and pass parameters to a function invoker, you must understand the structure of the parameter list, the meaning of each parameter, and whether DB2 or your user-defined function sets the value of each parameter. This section explains the parameters and gives examples of how a user-defined function in each host language receives the parameter list.

Figure 124 on page 322 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 124. 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 Table 36 for the host data types that are compatible with the data types in the user-defined function definition.

Table 36. Listing of tables of compatible data types

Language	Compatible data types table
Assembler	Table 12 on page 153
C	Table 14 on page 177
COBOL	Table 17 on page 211
PL/I	Table 21 on page 243

For LOBs, ROWIDs, and locators, see Table 37 for the assembler data types that are compatible with the data types in the user-defined function definition.

Table 37. 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 CLn 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 CLn 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>n</i> (=2* <i>n</i>) <= 65534: var DS 0FL4 var_length DS FL4 var_data DS CLm If <i>n</i> > 65534: var DS 0FL4 var_length DS FL4 var_data DS CL65534 ORG var_data+(<i>m</i> -65534)
ROWID	DS HL2,CL40

#

For LOBs, ROWIDs, VARCHARs, and locators see Table 38 for the C data types that are compatible with the data types in the user-defined function definition.

Table 38. 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;

|

#

Notes:

1. The SQLUDF file, which is in data set DSN810.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 Table 39 for the COBOL data types that are compatible with the data types in the user-defined function definition.

Table 39. 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) USAGE IS BINARY.

Table 39. Compatible COBOL declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	COBOL declaration
BLOB(<i>n</i>)	<pre> If n <= 32767: 01 var. 49 var-LENGTH PIC 9(9) USAGE COMP. 49 var-DATA PIC X(<i>n</i>). If length > 32767: 01 var. 02 var-LENGTH PIC S9(9) USAGE COMP. 02 var-DATA. 49 FILLER PIC X(32767). 49 FILLER PIC X(32767). : 49 FILLER PIC X(mod(<i>n</i>,32767)). </pre>
CLOB(<i>n</i>)	<pre> If n <= 32767: 01 var. 49 var-LENGTH PIC 9(9) USAGE COMP. 49 var-DATA PIC X(<i>n</i>). If length > 32767: 01 var. 02 var-LENGTH PIC S9(9) USAGE COMP. 02 var-DATA. 49 FILLER PIC X(32767). 49 FILLER PIC X(32767). : 49 FILLER PIC X(mod(<i>n</i>,32767)). </pre>
DBCLOB(<i>n</i>)	<pre> If n <= 32767: 01 var. 49 var-LENGTH PIC 9(9) USAGE COMP. 49 var-DATA PIC G(<i>n</i>) USAGE DISPLAY-1. If length > 32767: 01 var. 02 var-LENGTH PIC S9(9) USAGE COMP. 02 var-DATA. 49 FILLER PIC G(32767) USAGE DISPLAY-1. 49 FILLER PIC G(32767). USAGE DISPLAY-1. : 49 FILLER PIC G(mod(<i>n</i>,32767)) USAGE DISPLAY-1. </pre>

Table 39. Compatible COBOL declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	COBOL declaration
ROWID	01 var. 49 var-LEN PIC 9(4) USAGE COMP. 49 var-DATA PIC X(40).

For LOBs, ROWIDs, and locators, see Table 40 for the PL/I data types that are compatible with the data types in the user-defined function definition.

Table 40. 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));
CLOB(<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 40. Compatible PL/I declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	PL/I
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 “Passing parameter values to and from a user-defined function” on page 321 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 the explanation of DBINFO on page 331 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.

Refer to *DB2 Codes* for more information about the valid SQLSTATE values that a program may generate.

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 70 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 that the UPDATE or INSERT 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 statement
- In the VALUES list of an INSERT statement

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 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 statement
- In the VALUES list of an INSERT statement

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 *pppvvrrm*, where:

- *ppp* is a 3-byte product code:
 - ARI** DB2 Server for VSE & VM
 - DSN** DB2 UDB for z/OS
 - QSQ** DB2 UDB for iSeries
 - SQL** DB2 UDB 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	OS/390 or z/OS
13	Siemens Nixdorf
15	Windows 95
16	SCO UNIX
18	Linux
19	DYNIX/ptx
24	Linux for S/390
25	Linux for zSeries
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.

"Examples of receiving parameters in a user-defined function" has examples of declarations of passed parameters in each language. If you write your user-defined function in C or C++, you can use the declarations in member SQLUDF of DSN810.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 DSN810.SDSNC.H data set in the SYSLIB concatenation for the compile step of your program preparation job.
- Specify the NOMARGINS and NOSEQUENCE options in the compile 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: Figure 125 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=PROG SIZE,MAIN=YES,PLIST=0S
        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

PROG SIZE EQU      *-PROGAREA
        CEEDSA   ,                MAPPING OF THE DYNAMIC SAVE AREA
        CEECAA   ,                MAPPING OF THE COMMON ANCHOR AREA
        END      MYMAIN
```

Figure 125. How an assembler language user-defined function receives parameters

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].

Figure 126 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. */
    /******
  }
}
```

Figure 126. How a C or C++ user-defined function that is written as a main program receives parameters (Part 1 of 2)


```

/*****
/* 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/0");

/*****
/* 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         */
/* 'yyyyyyyy' to the diagnostic string returned   */
/* in the SQLCA associated with the invoked       */
/* user-defined function.                         */
*****/
strcpy(argv[10],"yyyyyyyy/0");

/*****
/* Use the following expression to assign the     */
/* result of the function.                        */
*****/
char l_result[11];
strcpy(argv[3],l_result);

:
}

```

Figure 126. How a C or C++ user-defined function that is written as a main program receives parameters (Part 2 of 2)

Figure 127 on page 337 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_ind2 = *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));

    :
}

```

Figure 127. How a C language user-defined function that is written as a subprogram receives parameters

Figure 128 on page 338 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_udbinfo *udf_dbinf);

{
    /******
    /* 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_dbinf *ludf_dbinf /* dbinf */
    /******
    /* 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_ind2 = *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_dbinf,udf_dbinf,sizeof(ludf_dbinf));

    :
}

```

Figure 128. How a C++ user-defined function that is written as a subprogram receives parameters

COBOL: Figure 129 on page 339 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(70).

```

Figure 129. How a COBOL user-defined function receives parameters (Part 1 of 3)

```

*****
* 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.

```

Figure 129. How a COBOL user-defined function receives parameters (Part 2 of 3)

```

*      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 UDFPARM1, UDFPARM2, 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.

```

Figure 129. How a COBOL user-defined function receives parameters (Part 3 of 3)

PL/I: Figure 130 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);

```

Figure 130. How a PL/I user-defined function receives parameters (Part 1 of 2)

```

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_CCIDS(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_CCSID_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_RESERVED1 CHAR(26),    /* reserved         */
      03 UDF_DBINFO_TFCOLUMN PTR,          /* -> TFcolumn list */
      03 UDF_DBINFO_APPLID PTR,            /* -> application id */
      03 UDF_DBINFO_RESERVED2 CHAR(20);    /* reserved         */

:

```

Figure 130. How a PL/I user-defined function receives parameters (Part 2 of 2)

Using special registers in a user-defined function

You can use all special registers in a user-defined function. However, you can modify only some of those special registers. After a user-defined function completes, DB2 restores all special registers to the values they had before invocation.

Table 41 shows information that you need when you use special registers in a user-defined function.

Table 41. Characteristics of special registers in a user-defined function

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Function can use SET to modify?
CURRENT CLIENT_ACCTNG	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT CLIENT_APPLNAME	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT CLIENT_USERID	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT CLIENT_WRKSTNNAME	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT APPLICATION ENCODING SCHEME	The value of bind option ENCODING for the user-defined function package ¹	The value of bind option ENCODING for the user-defined function package ¹	Yes

Table 41. Characteristics of special registers in a user-defined function (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Function can use SET to modify?
CURRENT DATE	New value for each SQL statement in the user-defined function package ²	New value for each SQL statement in the user-defined function package ²	Not applicable ⁵
CURRENT DEGREE	Inherited from invoking application ³	The value of field CURRENT DEGREE on installation panel DSNTIP8	Yes
CURRENT LOCALE LC_CTYPE	Inherited from invoking application	The value of field CURRENT DEGREE on installation panel DSNTIP8	Yes
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Inherited from invoking application	The value of field CURRENT MAINT TYPES on installation panel DSNTIP8	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	No
CURRENT OPTIMIZATION HINT	The value of bind option OPTHINT for the user-defined function package or inherited from invoking application ⁶	The value of bind option OPTHINT for the user-defined function package	Yes
CURRENT PACKAGESET	Inherited from invoking application ⁴	Inherited from invoking application ⁴	Yes
CURRENT PACKAGE PATH	Inherited from invoking application ⁹	Inherited from invoking application ⁹	Yes
CURRENT PATH	The value of bind option PATH for the user-defined function package or inherited from invoking application ⁶	The value of bind option PATH for the user-defined function package	Yes
CURRENT PRECISION	Inherited from invoking application	The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4	Yes
CURRENT REFRESH AGE	Inherited from invoking application	The value of field CURRENT REFRESH AGE on installation panel DSNTIP8	Yes
CURRENT RULES	Inherited from invoking application	The value of bind option SQLRULES for the user-defined function package	Yes
CURRENT SCHEMA	Inherited from invoking application	The value of CURRENT SQLID when the user defined function is entered	Yes
CURRENT SERVER	Inherited from invoking application	Inherited from invoking application	Yes
CURRENT SQLID	The primary authorization ID of the application process or inherited from 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 package ²	New value for each SQL statement in the user-defined function package ²	Not applicable ⁵

Table 41. Characteristics of special registers in a user-defined function (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Function can use SET to modify?
CURRENT TIMESTAMP	New value for each SQL statement in the user-defined function package ²	New value for each SQL statement in the user-defined function package ²	Not applicable ⁵
CURRENT TIMEZONE	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
I ENCRYPTION PASSWORD	Inherited from invoking application	A string of 0 length	Yes
USER	Primary authorization ID of the application process	Primary authorization ID of the application process	Not applicable ⁵

Notes:

1. If the ENCODING bind option is not specified, the initial value is the value that was specified in field APPLICATION ENCODING of installation panel DSNTIPF.
2. If the user-defined function 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 function package.
3. 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.
4. If the user-defined function definer specifies a value for COLLID in the CREATE FUNCTION statement, DB2 sets CURRENT PACKAGESET to the value of COLLID.
5. Not applicable because no SET statement exists for the special register.
6. If a program within the scope of the invoking application issues a SET statement for the special register before the user-defined function 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 package.
7. If a program within the scope of the invoking application issues a SET CURRENT SQLID statement before the user-defined function 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 package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed but does not affect the authorization ID that is used for the dynamic SQL statements in the user-defined function package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements. See “Using DYNAMICRULES to specify behavior of dynamic SQL statements” on page 502 for more information about DYNAMICRULES values and authorization IDs.
9. If the user-defined function definer specifies a value for COLLID in the CREATE FUNCTION statement, DB2 sets CURRENT PACKAGE PATH to an empty string.

Using a scratchpad in a user-defined 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.

Figure 131 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 *scratchp) /* Scratchpad              */
{
    *out = ++scratchp->countr; /* Increment counter and
    /* copy to output variable */
    *outnull = 0;             /* Set output null indicator*/
    return;
}
/* end of user-defined function ctr */
```

Figure 131. Example of coding a scratchpad in a user-defined function

Accessing transition tables in a user-defined function or stored procedure

When you write a user-defined function, external stored procedure, or SQL procedure that is invoked from a trigger, you might need access to transition tables for the trigger. This section 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. See Chapter 5 of *DB2 SQL Reference* for more information.

The five basic steps to accessing transition tables in a user-defined function are:

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. The syntax for declaring table locators in C, C++, COBOL, and PL/I is described in Chapter 9, “Embedding SQL statements in host languages,” on page 143. The syntax for declaring table locators in an SQL procedure is described in Chapter 6 of *DB2 SQL Reference*.
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: Figure 132 on page 347 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,PROG SIZE      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,GETLENT H     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
    :
    :

```

Figure 132. How an assembler user-defined function accesses a transition table (Part 1 of 2)

```

PROGAREA DSECT
SAVEAREA DS 18F          WORKING STORAGE FOR THE PROGRAM
                        THIS ROUTINE'S SAVE AREA
GETLENT H DS A           GETMAIN LENGTH FOR THIS AREA
    :
NAME      DS CL24
    :
    :
    DS 0D
PROG SIZE EQU *-PROGAREA DYNAMIC WORKAREA SIZE
PARMAREA DSECT
TABLOC DS A             INPUT PARAMETER FOR TABLE LOCATOR
    :
END CHECKEMP

```

Figure 132. How an assembler user-defined function accesses a transition table (Part 2 of 2)

C or C++: Figure 133 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;

    :
}
```

Figure 133. How a C or C++ user-defined function accesses a transition table

COBOL: Figure 134 on page 349 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.

```

Figure 134. How a COBOL user-defined function accesses a transition table

PL/I: Figure 135 on page 350 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;

```

Figure 135. How a PL/I user-defined function accesses a transition table

Preparing a user-defined function for execution

To prepare a user-defined function for execution, perform these steps:

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 RRSAF.

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 RRSAF.

For the minimum compiler and Language Environment requirements for user-defined functions, see *DB2 Release Planning Guide*.

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. If your DB2 subsystem has been installed to work with Language Environment, you can use this sample JCL when you prepare your user-defined functions. For object-oriented programs in C++ or COBOL, see JCL samples DSNHCPP2 and DSNHICB2 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.

Making a user-defined function reentrant

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, 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 "Specifying the package list for the PKLIST option of BIND PLAN" on page 499 to search for the package.

Determining the authorization ID for user-defined function invocation

If your user-defined function is invoked statically, the authorization ID under which the user-defined function is invoked is the owner of the package that contains the user-defined function invocation.

If the user-defined function is invoked dynamically, the authorization ID under which the user-defined function is invoked depends on the value of bind parameter DYNAMICRULES for the package that contains the function invocation.

While a user-defined function is executing, the authorization ID under which static SQL statements in the user-defined function package execute is the owner of the user-defined function package. The authorization ID under which dynamic SQL statements in the user-defined function package execute depends on the value of DYNAMICRULES with which the user-defined function package was bound.

DYNAMICRULES influences a number of features of an application program. For information about how DYNAMICRULES works, see "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 502. For more information about the authorization needed to invoke and execute SQL statements in a user-defined function, see Chapter 5 of *DB2 SQL Reference* and Part 3 (Volume 1) of *DB2 Administration Guide*.

Preparing user-defined functions to run concurrently

Multiple user-defined functions and stored procedures can run concurrently, each under its own z/OS task (TCB).

To maximize the number of user-defined functions and stored procedures that can run concurrently, follow these preparation recommendations:

- Ask the system administrator to set the region size parameter in the startup procedures for the WLM-established stored procedures address spaces to REGION=0. This lets an address space obtain the largest possible amount of storage below the 16-MB line.
- Limit storage required by application programs below the 16-MB line by:
 - Link-editing programs with the AMODE(31) and RMODE(ANY) attributes
 - Compiling COBOL programs with the RES and DATA(31) options
- Limit storage that is required by Language Environment by using these run-time options:

HEAP(,,ANY)	Allocates program heap storage above the 16-MB line
STACK(,,ANY,)	Allocates program stack storage above the 16-MB line
STORAGE(,,,4K)	Reduces reserve storage area below the line to 4 KB
BELOWHEAP(4K,,)	Reduces the heap storage below the line to 4 KB
LIBSTACK(4K,,)	Reduces the library stack below the line to 4 KB
ALL31(ON)	Causes all programs contained in the external user-defined function to execute with AMODE(31) and RMODE(ANY)

The definer can list these options as values of the RUN OPTIONS parameter of CREATE FUNCTION, or the system administrator can establish these options as defaults during Language Environment installation.

For example, the RUN OPTIONS option parameter could contain:

```
H(,,ANY),STAC(,,ANY,),STO(,,,4K),BE(4K,,),LIBS(4K,,),ALL31(ON)
```

- Ask the system administrator to set the NUMTCB parameter for WLM-established stored procedures address spaces to a value greater than 1. This lets more than one TCB run in an address space. Be aware that setting NUMTCB to a value greater than 1 also reduces your level of application program isolation. For example, a bad pointer in one application can overwrite memory that is allocated by another application.

Testing a user-defined function

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where user-defined functions run. This section describes some alternative testing strategies.

Debug Tool for z/OS: You can use the Debug Tool for z/OS, which works with Language Environment, to test DB2 UDB for z/OS user-defined functions written in any of the supported languages. You can use the Debug Tool either interactively or in batch mode.

Using the Debug Tool interactively: To test a user-defined function interactively using the Debug Tool, you must have the Debug Tool installed on the z/OS system where the user-defined function runs. To debug your user-defined function using the Debug Tool, do the following:

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 parameter of CREATE FUNCTION or ALTER FUNCTION. See “Components of a user-defined function definition” on page 314 for more information about the RUN OPTIONS parameter.

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.

Using the Debug Tool in batch mode: To test your user-defined function in batch mode, you must have the Debug Tool installed on the z/OS system where the user-defined function runs. To debug your user-defined function in batch mode using the Debug Tool, do the following:

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 CEESTEST 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 CEESTEST calls. For example, you might want to use TEST to name the commands data set but use CEESTEST calls to control when the Debug Tool takes control.

You can combine the Language Environment run-time TEST option with CEESTEST calls. For example, you might want to use TEST to name the commands data set but use CEESTEST calls to control when the Debug Tool takes control.

For more information about the Debug Tool, see *Debug Tool User's Guide and Reference*.

Route 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.

Driver applications: You can write a small driver application that calls the 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.

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 a workstation) to easily access the data in the table 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.

Implementing an SQL scalar function

An SQL scalar function is a user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL scalar function can return only one parameter. You specify the SQL expression in the RETURN clause of the CREATE FUNCTION statement. The value of the SQL expression must be compatible with the data type of the parameter in the RETURNS clause.

See “Defining a user-defined function” on page 314 and Chapter 5 of *DB2 SQL Reference* for a description of the parameters that you can specify in the CREATE FUNCTION statement for an SQL scalar function.

To prepare an SQL scalar function for execution, you execute the CREATE FUNCTION statement, either statically or dynamically.

Invoking a user-defined function

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.

See the following sections for details you should know before you invoke a user-defined function:

- “Syntax for user-defined function invocation”
- “Ensuring that DB2 executes the intended user-defined function” on page 356
- “Casting of user-defined function arguments” on page 362
- “What happens when a user-defined function abnormally terminates” on page 363

Syntax for user-defined function invocation

Use the syntax shown in Figure 136 when you invoke a user-defined scalar function:

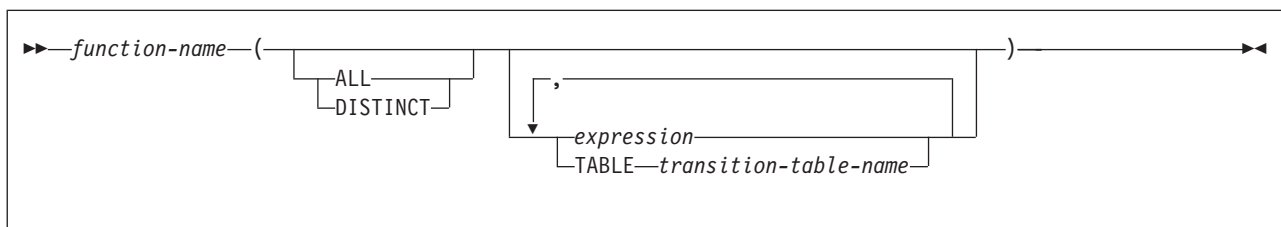


Figure 136. Syntax for user-defined scalar function invocation

Use the syntax shown in Figure 137 on page 356 when you invoke a table function:

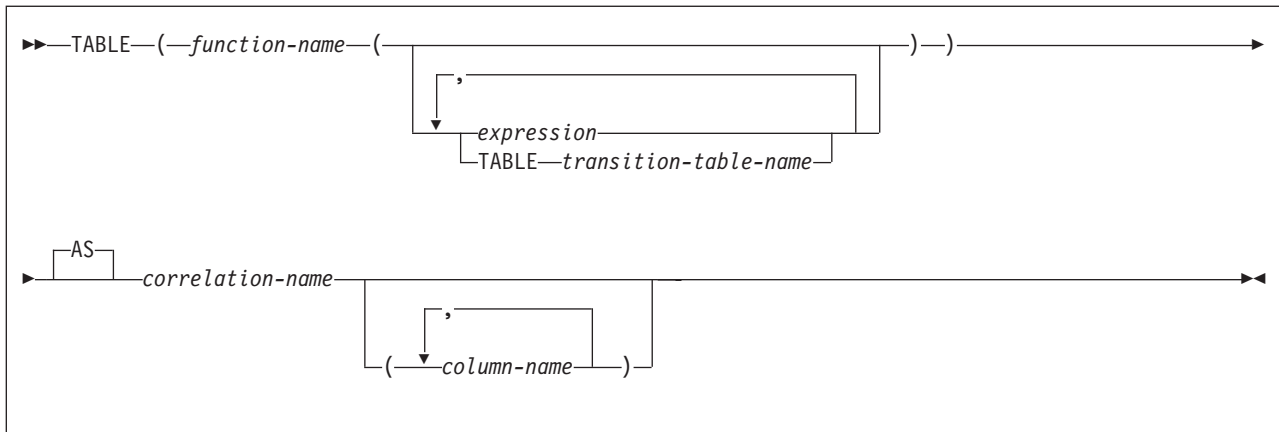


Figure 137. Syntax for table function invocation

See Chapter 2 of *DB2 SQL Reference* for more information about the syntax of user-defined function invocation.

Ensuring that DB2 executes the intended user-defined function

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. This process is known as *function resolution*. You need to understand DB2's function resolution process to ensure that you invoke the user-defined function that you want to invoke.

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 Chapter 12, "Using triggers for active data," on page 277.

- 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. See Appendix B (Volume 2) of *DB2 Administration Guide* for information about function resolution with access control authorization exit routines.

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.

To determine whether a data type is promotable to another data type, see Table 42. The first column lists data types in function invocations. The second column lists data types to which the types in the first column can be promoted, in order from best fit to worst fit. For 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.

Table 42. Promotion of data types

Data type in function invocation	Possible fits (in best-to-worst order)
SMALLINT	SMALLINT INTEGER DECIMAL REAL DOUBLE
INTEGER	INTEGER DECIMAL REAL DOUBLE
DECIMAL	DECIMAL REAL DOUBLE
REAL ²	REAL DOUBLE
DOUBLE ³	DOUBLE
CHAR or GRAPHIC	CHAR or GRAPHIC VARCHAR or VARGRAPHIC CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC CLOB or DBCLOB
CLOB or DBCLOB ¹	CLOB or DBCLOB
BLOB ¹	BLOB
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
ROWID	ROWID
Distinct type	Distinct type with same name

Table 42. Promotion of data types (continued)

Data type in function invocation	Possible fits (in best-to-worst order)
Notes:	
1.	This promotion also applies if the parameter type in the invocation is a LOB locator for a LOB with this data type.
2.	The FLOAT type with a length of less than 22 is equivalent to REAL.
3.	The FLOAT type with a length of greater than or equal to 22 is equivalent to DOUBLE.

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:

- 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.
- 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. Table 42 on page 358 shows the possible fits for each data type, in best-to-worst order.
- 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 criteria in “How DB2 chooses candidate functions” on page 357 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.

How you can simplify function resolution

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;
```

- Avoid defining user-defined function numeric parameters as SMALLINT or REAL. Use INTEGER or DOUBLE instead. An invocation of a user-defined function defined with parameters of type SMALLINT or REAL must use 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. An invocation like this 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 or GRAPHIC), 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 or VARGRAPHIC), 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, 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 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;
```

Using DSN_FUNCTION_TABLE to see how DB2 resolves a function

You can use the DB2 EXPLAIN tool to obtain information about how DB2 resolves functions. DB2 stores the information in a table called DSN_FUNCTION_TABLE, which you create. DB2 puts a row in DSN_FUNCTION_TABLE for each function that is referenced in an SQL statement when one of the following events occurs:

- You execute the SQL EXPLAIN statement on an SQL statement that contains user-defined function invocations.
- You run a program whose plan is bound with EXPLAIN(YES), and the program executes an SQL statement that contains user-defined function invocations.

Before you use EXPLAIN to obtain information about function resolution, create DSN_FUNCTION_TABLE. The table definition looks like this:

```
CREATE TABLE DSN_FUNCTION_TABLE
(QUERYNO          INTEGER          NOT NULL WITH DEFAULT,
 QBLOCKNO         INTEGER          NOT NULL WITH DEFAULT,
 APPLNAME         CHAR(8)          NOT NULL WITH DEFAULT,
 PROGNAME         VARCHAR(128)     NOT NULL WITH DEFAULT,
 COLLID          VARCHAR(128)     NOT NULL WITH DEFAULT,
 GROUP_MEMBER     CHAR(8)          NOT NULL WITH DEFAULT,
 EXPLAIN_TIME     TIMESTAMP        NOT NULL WITH DEFAULT,
 SCHEMA_NAME      VARCHAR(128)     NOT NULL WITH DEFAULT,
 FUNCTION_NAME    VARCHAR(128)     NOT NULL WITH DEFAULT,
 SPEC_FUNC_NAME   VARCHAR(128)     NOT NULL WITH DEFAULT,
 FUNCTION_TYPE    CHAR(2)          NOT NULL WITH DEFAULT,
 VIEW_CREATOR     VARCHAR(128)     NOT NULL WITH DEFAULT,
 VIEW_NAME        VARCHAR(128)     NOT NULL WITH DEFAULT,
 PATH             VARCHAR(2048)    NOT NULL WITH DEFAULT,
 FUNCTION_TEXT    VARCHAR(1500)    NOT NULL WITH DEFAULT);
```

Columns QUERYNO, QBLOCKNO, APPLNAME, PROGNAME, COLLID, and GROUP_MEMBER have the same meanings as in the PLAN_TABLE. See Chapter 27, “Using EXPLAIN to improve SQL performance,” on page 789 for explanations of those columns. The meanings of the other columns are:

EXPLAIN_TIME

Timestamp when the EXPLAIN statement was executed.

SCHEMA_NAME

Schema name of the function that is invoked in the explained statement.

FUNCTION_NAME

Name of the function that is invoked in the explained statement.

SPEC_FUNC_NAME

Specific name of the function that is invoked in the explained statement.

FUNCTION_TYPE

The type of function that is invoked in the explained statement. Possible values are:

SU Scalar function

TU Table function

VIEW_CREATOR

The creator of the view, if the function that is specified in the FUNCTION_NAME column is referenced in a view definition. Otherwise, this field is blank.

VIEW_NAME

The name of the view, if the function that is specified in the FUNCTION_NAME column is referenced in a view definition. Otherwise, this field is blank.

PATH

The value of the SQL path when DB2 resolved the function reference.

FUNCTION_TEXT

The text of the function reference (the function name and parameters). If the function reference exceeds 1500 bytes, this column contains the first 1500 bytes.

For a function specified in infix notation, FUNCTION_TEXT contains only the function name. For example, suppose a user-defined function named / is in the function reference A/B. Then FUNCTION_TEXT contains only /, not A/B.

Casting of user-defined function arguments

Whenever you invoke a user-defined function, DB2 assigns your input parameter values to parameters with the data types and lengths in the user-defined function definition. See Chapter 2 of *DB2 SQL Reference* for information about how DB2 assigns values when the data types of the source and target differ.

When you invoke a user-defined function that is sourced on another function, DB2 casts your parameters 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:

```
SELECT FX(CAST(? AS DECIMAL(9,2))) FROM T1;
```

What happens when a user-defined function abnormally terminates

If an external user-defined function abnormally terminates:

- Your program receives SQLCODE -430 for the invoking statement.
- DB2 places the unit of work that contains the invoking statement in a must-rollback state.
- DB2 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.

Nesting 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 is known as *nesting* of SQL statements. DB2 supports up to 16 levels of nesting. Figure 138 on page 364 shows an example of 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

```

Figure 138. Nested SQL statements

DB2 has the following restrictions on nested SQL statements:

- Restrictions for SELECT statements:

When you execute a SELECT statement on a table, you cannot execute INSERT, UPDATE, 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 INSERT, UPDATE, and DELETE statements:

When you execute an INSERT, DELETE, or UPDATE 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;
```

```
#
#
#
#
#
#
#
```

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.

Recommendations for user-defined function invocation

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 set, 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.

Chapter 16. Creating and using distinct types

A distinct type is a data type that you define using the CREATE DISTINCT TYPE statement. Each distinct type has the same internal representation as a built-in data type. You can use distinct types in the same way that you use built-in data types, in any type of SQL application except for a DB2 private protocol application.

This chapter presents the following information about distinct types:

- “Introduction to distinct types”
- “Using distinct types in application programs” on page 368
- “Combining distinct types with user-defined functions and LOBs” on page 372

Introduction to distinct types

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 Chapter 14, “Programming for large objects,” on page 297.

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.

Comparing distinct types

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.

DB2 does not let you 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.

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 “Using parameter markers with PREPARE and EXECUTE” on page 606, 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)

```

Assigning distinct types

For assignments from columns to columns or from constants to columns of distinct types, the type of that value to be assigned must match the type of the object to which the value is assigned, or you must be able to cast one type to the other.

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. For a table of base data types and the base data types to which they can be cast, see Table 42 on page 358.

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   */
```

Using distinct types in UNIONS

As with comparisons, DB2 enforces strong typing of distinct types in UNIONS. When you use a UNION to combine column values from several tables, the combined columns must be of the same types. For example, suppose 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, so 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 that converts values of type EURO to US_DOLLAR
- A function that converts values of type JAPANESE_YEN to US_DOLLAR

Assume that these functions exist, and that both are called 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, US_DOLLAR(TOTAL)
FROM EUROPEAN_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR(TOTAL)
FROM JAPAN_SALES;
```

Because the result type of both US_DOLLAR functions is US_DOLLAR, you have satisfied the requirement that the distinct types of the combined columns are the same.

Invoking functions with distinct types

DB2 enforces strong typing when you pass arguments to a function. This means that:

- 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));
```

Combining distinct types with user-defined functions and LOBs

The example in this section 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));
:
:

```

Part 4. Designing a DB2 database application

Chapter 17. Planning for DB2 program preparation	381
Planning to process SQL statements	383
Planning to bind	384
Binding DBRMs with packages and plans	384
Binding with a package list only	384
Binding all DBRMs to a plan	385
Binding with both DBRMs and a package list	385
Advantages of packages	385
Planning for changes to your application	386
Dropping objects	387
Rebinding a package	387
Rebinding a plan	388
Rebinding lists of plans and packages	389
Working with trigger packages	389
Automatic rebinding	389
 Chapter 18. Planning for concurrency	 393
Definitions of concurrency and locks	393
Effects of DB2 locks	394
Suspension	394
Timeout	394
Deadlock	395
Basic recommendations to promote concurrency	397
Recommendations for database design	398
Recommendations for application design	399
Aspects of transaction locks	402
The size of a lock	402
Definition	402
Hierarchy of lock sizes	402
General effects of size	403
Effects of table spaces of different types	403
The duration of a lock	404
Effects	404
The mode of a lock	404
Modes of page and row locks	405
Modes of table, partition, and table space locks	405
Lock mode compatibility	406
The object of a lock	407
Definition and examples	407
Indexes and data-only locking	407
Options for tuning locks	408
Bind options	408
The ACQUIRE and RELEASE options	408
Advantages and disadvantages of the combinations	410
The ISOLATION option	412
Advantages and disadvantages of the isolation values	412
The CURRENTDATA option	417
When plan and package options differ	420
The effect of WITH HOLD for a cursor	420
Isolation overriding with SQL statements	421
The LOCK TABLE statement	422
The purpose of LOCK TABLE	422
The effect of LOCK TABLE	422
Recommendations for using LOCK TABLE	423
Access paths	424
LOB locks	425

Relationship between transaction locks and LOB locks	425
Hierarchy of LOB locks	426
LOB and LOB table space lock modes	426
Modes of LOB locks	427
Modes of LOB table space locks	427
LOB lock and LOB table space lock duration	427
The duration of LOB locks	427
The duration of LOB table space locks	428
Instances when LOB table space locks are not taken	428
The LOCK TABLE statement for LOBs	428
Chapter 19. Planning for recovery.	431
Unit of work in TSO batch and online	432
Unit of work in CICS.	432
Unit of work in IMS online programs	433
Planning ahead for program recovery: Checkpoint and restart	435
What symbolic checkpoint does	435
What restart does	435
When are checkpoints important?	436
Checkpoints in MPPs and transaction-oriented BMPs	436
Checkpoints in batch-oriented BMPs	437
Specifying checkpoint frequency	438
Unit of work in DL/I and IMS batch programs	438
Commit and rollback coordination	438
Using ROLL.	439
Using ROLB.	439
In batch programs.	439
Restart and recovery in IMS batch	439
Using savepoints to undo selected changes within a unit of work	439
Chapter 20. Planning to access distributed data.	441
Planning for DRDA and DB2 private protocol access	441
Advantages of DRDA access	442
Moving from DB2 private protocol access to DRDA access	442
Bind processes for DRDA and DB2 private protocol access	444
Precompiler and bind options for DRDA access	445
Precompiler options for DRDA access	445
BIND PLAN options for DRDA access	445
BIND PACKAGE options for DRDA access	446
Checking BIND PACKAGE options	447
Coding methods for distributed data	448
Using three-part table names to access distributed data	448
Three-part names and multiple servers	449
Accessing declared temporary tables by using three-part names	449
Using explicit CONNECT statements to access distributed data	450
Using a location alias name for multiple sites.	451
Releasing connections	451
Coordinating updates to two or more data sources	452
Working without two-phase commit.	452
Update restrictions on servers that do not support two-phase commit	453
Forcing update restrictions by using CONNECT (Type 1).	453
Maximizing performance for distributed data	454
Coding efficient queries	454
Maximizing LOB performance in a distributed environment.	454
Using bind options to improve performance for distributed applications	456
DEFER(PREPARE).	456
PKLIST	456
REOPT(ALWAYS)	457
CURRENTDATA(NO)	458
KEEPDYNAMIC(YES)	458
DBPROTOCOL(DRDA)	458

Using block fetch in distributed applications	458
When DB2 uses block fetch for non-scrollable cursors	459
When DB2 uses block fetch for scrollable cursors	459
Limiting the number of DRDA network transmissions.	461
Limiting the number of rows returned to DRDA clients	464
Fast implicit close and FETCH FIRST n ROWS ONLY	464
Example of FETCH FIRST n ROWS ONLY	464
Limiting the number of rows with the rowset parameter	465
Working with distributed data.	465
SQL limitations at dissimilar servers.	465
I Executing long SQL statements in a distributed environment	466
Retrieving data from ASCII or Unicode tables	466
Accessing data with a scrollable cursor when the requester is down-level	467
I Accessing data with a rowset-positioned cursor when the requester is down-level	467
Maintaining data currency by using cursors	467
Copying a table from a remote location.	467
Transmitting mixed data.	467
Converting mixed data	467
Identifying the server at run time	468

Chapter 17. Planning for DB2 program preparation

DB2 application programs include SQL statements. You need to process those SQL
statements, using either the DB2 precompiler or the DB2 coprocessor that is
provided with a compiler. Either type of SQL statement processor does the
following things:

- # • 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

For specific information about accomplishing the steps in program preparation, see Chapter 21, “Preparing an application program to run,” on page 471.

| Figure 139 on page 382 illustrates the program preparation process when you use
| the DB2 precompiler. After you process SQL statements in your source program
| 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 or
| BIND PLAN commands.

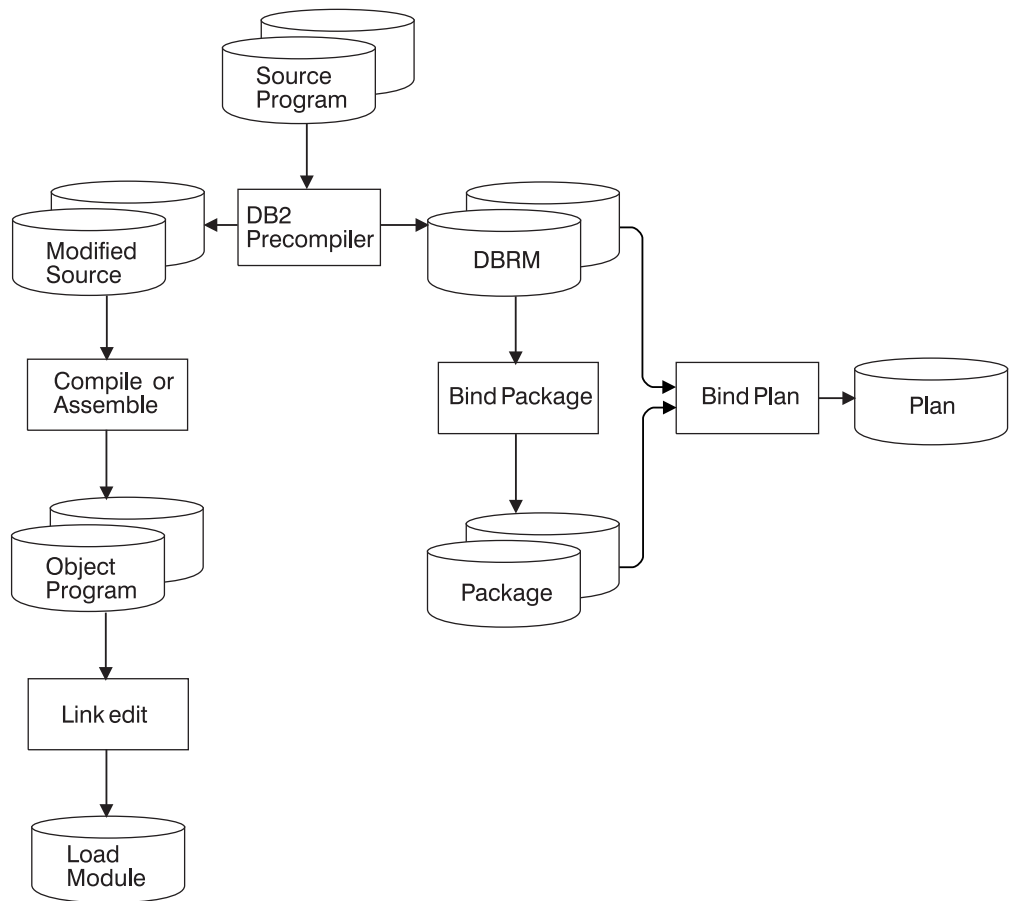


Figure 139. Program preparation with the DB2 precompiler

|
|
|
|

Figure 140 on page 383 illustrates the program preparation process when you use the DB2 coprocessor. The process is similar to the process used with the DB2 precompiler, except that the DB2 coprocessor does not create modified source for your application program.

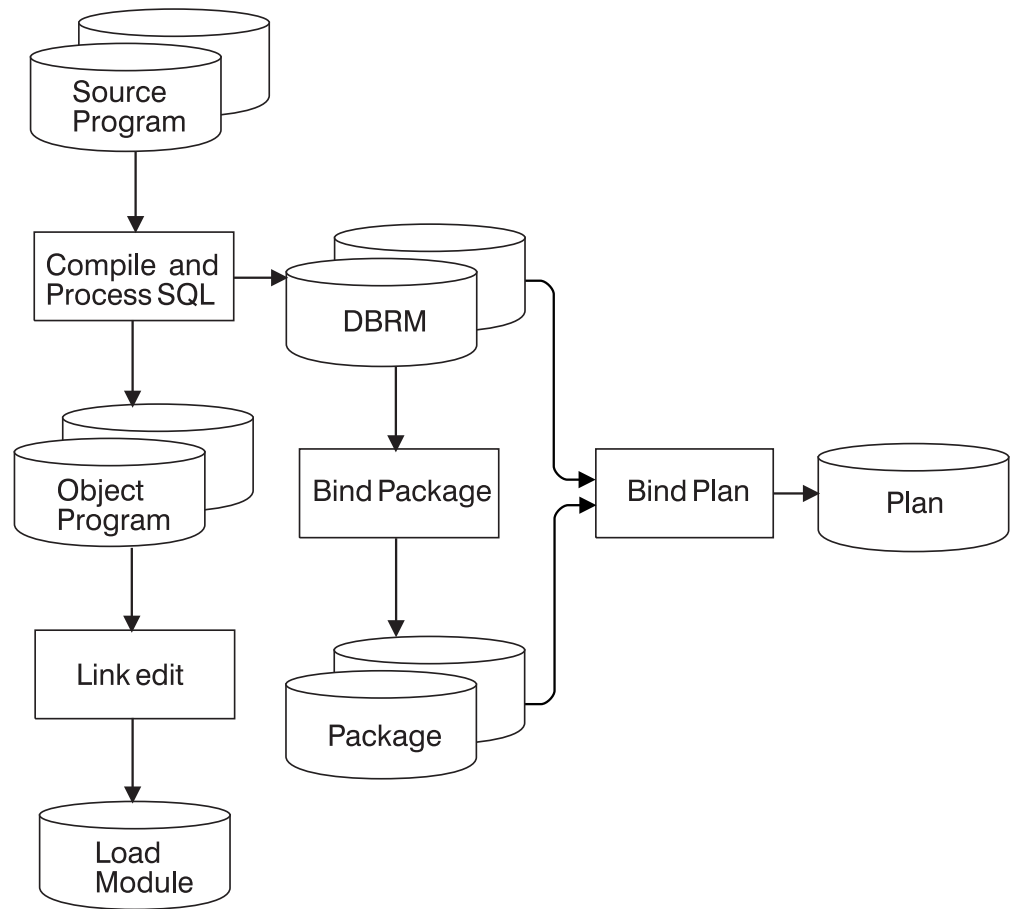


Figure 140. Program preparation with the DB2 coprocessor

Planning to process SQL statements

When you process SQL statements in an application program, you can specify a number of options. Most of the options do not affect the way you design or code the program. Those options describe the basic characteristics of the source program or indicate how you want the output listings to look. For example, there are options that specify:

- 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.

A few options, however, can 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.

Before you begin writing your program, review the list of options in Table 64 on page 484. You can specify any of those options whether you use the DB2 precompiler or an DB2 coprocessor. However, the DB2 coprocessor might ignore certain options because there are host language compiler options that provide the same information.

#

Planning to bind

Depending on how you design your DB2 application, you might bind all your DBRMs in one operation, creating only a single application plan. Alternatively, you might bind some or all of your DBRMs into separate packages in separate operations. After that, you must still bind the entire application as a single plan, listing the included packages or collections and binding any DBRMs that are not already bound into packages. Regardless of what the plan contains, **you must bind a plan before the application can run.**

Binding or rebinding a package or plan in use: Packages and plans are locked when you bind or run them. Packages that run under a plan are not locked until the plan uses them. If you run a plan and some packages in the package list never run, those packages are never locked.

You cannot bind or rebind a package or a plan while it is running. However, you can bind a different version of a package that is running.

Options for binding and rebinding: Several of the options of BIND PACKAGE and BIND PLAN 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 a particular IMS region—you do not need to enforce this in your code. Several other options are discussed at length in later chapters, particularly the ones that affect your program's use of locks, such as the ISOLATION option. Before you finish reading this chapter, you might want to review those options in Chapter 2 of *DB2 Command Reference*.

Preliminary steps: Before you bind, consider the following:

- Determine how you want to bind the DBRMs. You can bind them into packages or directly into plans, or you can use a combination of both methods.
- Develop a naming convention and strategy for the most effective and efficient use of your plans and packages.
- Determine when your application should acquire locks on the objects it uses: on all objects when the plan is first allocated, or on each object in turn when that object is first used. For a description of the consequences of these choices, see "The ACQUIRE and RELEASE options" on page 408.

Binding DBRMs with packages and plans

The question of whether to use packages affects your application design from the beginning. For example, you might decide to put certain SQL statements together in the same program, precompiling them into the same DBRM and then binding them into a single package.

Input to binding the plan can include DBRMs only, a package list only, or a combination of the two. When choosing one of those alternatives for your application, consider the impact of rebinding; see "Planning for changes to your application" on page 386.

Binding with a package list only

At one extreme, you can bind each DBRM into its own package. Input to binding a package is a single DBRM only. A one-to-one correspondence between programs and packages might easily allow you to keep track of each. However, your application might consist of too many packages to track easily.

Binding a plan that includes only a package list makes maintenance easier when the application changes significantly over time.

Binding all DBRMs to a plan

At the other extreme, you can bind all your DBRMs into a single plan. This approach has the disadvantage that a change to even one DBRM requires rebinding the entire plan, even though most DBRMs are unchanged.

Binding all DBRMs to a plan is suitable for small applications that are unlikely to change or that require all resources to be acquired when the plan is allocated rather than when your program first uses them.

Binding with both DBRMs and a package list

Binding DBRMs directly to the plan and specifying a package list is suitable for maintaining existing applications. You can add a package list when rebinding an existing plan. To migrate gradually to the use of packages, bind DBRMs as packages when you need to make changes.

Advantages of packages

You must decide how to use packages based on your application design and your operational objectives. The following are advantages of using packages:

Ease of maintenance: When you use packages, you do not need to bind the entire plan again when you change one SQL statement. You need to bind only the package that is associated with the changed SQL statement.

Incremental development of your program: Binding packages into package collections allows you to add packages to an existing application plan without having to bind the entire plan again. A *collection* is a group of associated packages. If you include a collection name in the package list when you bind a plan, any package in the collection becomes available to the plan. The collection can even be empty when you first bind the plan. Later, you can add packages to the collection, and drop or replace existing packages, without binding the plan again.

Versioning: Maintaining several versions of a plan without using packages requires a separate plan for each version, and therefore separate plan names and RUN commands. Isolating separate versions of a program into packages requires only one plan and helps to simplify program migration and fallback. For example, you can maintain separate development, test, and production levels of a program by binding each level of the program as a separate version of a package, all within a single plan.

Flexibility in using bind options: The options of BIND PLAN apply to all DBRMs that are bound directly to the plan. The options of BIND PACKAGE apply only to the single DBRM that is bound to that package. The package options need not all be the same as the plan options, and they need not be the same as the options for other packages that are used by the same plan.

Flexibility in using name qualifiers: You can use a bind option to name a qualifier for the unqualified object names in SQL statements in a plan or package. By using packages, you can use different qualifiers for SQL statements in different parts of your application. By rebinding, you can redirect your SQL statements, for example, from a test table to a production table.

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. See “Using packages with dynamic plan selection” on page 507 for information about using packages with dynamic plan selection.

Planning for changes to your application

As you design your application, consider what will happen to your plans and packages when you make changes to your application.

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

- 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, recreate the trigger associated with the trigger package.

- To rebind an existing plan or package, other than a trigger package, use the `REBIND` subcommand.

To rebind trigger package, use the `REBIND TRIGGER PACKAGE` subcommand.

Table 43 tells which action particular types of change require. For more information about trigger packages, see “Working with trigger packages” on page 389.

If you want to change the bind options in effect when the plan or package runs, review the descriptions of those bind options in Part 3 of *DB2 Command Reference*. Some options of `BIND` are not available on `REBIND`.

A plan or package can also become invalid for reasons that do not depend on operations in your program (such as when an index is dropped that is used as an access path by one of your queries). In those cases, DB2 might rebind the plan or package automatically, the next time it is used. (For details about automatic rebinding, see “Automatic rebinding” on page 389.)

Table 43. Changes requiring BIND or REBIND

Change made	Minimum action necessary
Drop a table, index, or other object, and recreate the object	If a table with a trigger is dropped, recreate the trigger if you recreate the table. Otherwise, no change is required; automatic rebind is attempted at the next run.
Revoke an authorization to use an object	None required; automatic rebind is attempted at the next run. Automatic rebind fails if authorization is still not available; then you must issue <code>REBIND</code> for the package or plan.
Run <code>RUNSTATS</code> to update catalog statistics	Issue <code>REBIND</code> for the package or plan to possibly change the access path that DB2 uses.

Table 43. Changes requiring BIND or REBIND (continued)

Change made	Minimum action necessary
Add an index to a table	Issue REBIND for the package or plan to use the index.
Change bind options	Issue REBIND for the package or plan, or issue BIND with ACTION(REPLACE) if the option you want is not available on REBIND.
Change statements in host language and SQL statements	Precompile, compile, and link the application program. Issue BIND with ACTION(REPLACE) for the package or plan.

Dropping objects

If you drop an object that a package depends on, the package might become invalid for the following reasons:

- If the package is not appended to any running plan, the package becomes invalid.
- If the package is appended to a running plan, and the drop occurs within that plan, the package becomes invalid.

However, if the package is appended to a running plan, and the drop occurs outside of that plan, the object is not dropped, and the package does not become invalid.

In all cases, the plan does not become invalid until it has a DBRM that references the dropped object. If the package or plan becomes invalid, automatic rebind occurs the next time the package or plan is allocated.

Rebinding a package

Table 44 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. For syntax and descriptions of this subcommand, see Part 3 of *DB2 Command Reference*.

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 44. 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

Table 44. Behavior of REBIND PACKAGE specification (continued). "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
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. For information about the REBIND subcommand options, see *DB2 Command Reference*.

```
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.())
```

Rebinding a 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
```

Rebinding lists of plans and packages

You can generate a list of REBIND subcommands for a set of plans or packages that cannot be described by using asterisks, by using information in the DB2 catalog. You can then issue the list of subcommands through DSN.

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 Appendix F, “REBIND subcommands for lists of plans or packages,” on page 1103.

Working with 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 recreate 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. The default values for the options that you can change are:

- CURRENTDATA(YES)
- EXPLAIN(YES)
- FLAG(I)
- ISOLATION(RR)
- IMMEDIATEWRITE(NO)
- RELEASE(COMMIT)

When you run REBIND TRIGGER PACKAGE, you can change only the values of options CURRENTDATA, EXPLAIN, FLAG, IMMEDIATEWRITE, ISOLATION, and RELEASE.

Automatic rebinding

Automatic rebinding might occur if an authorized user invokes a plan or package when the attributes of the data on which the plan or package depends change, or if the environment in which the package executes changes. Whether the automatic rebinding occurs depends on the value of the field AUTO BIND on installation panel DSN TIPO. The options used for an automatic rebinding are the options used during the most recent binding process.

In most cases, DB2 marks a plan or package that needs to be automatically rebound as invalid. A few common situations in which DB2 marks a plan or 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 plan or package depends is dropped
- When the authorization of the owner to access any of those objects is revoked
- When the authorization to execute a stored procedure is revoked from a plan or package owner, and the plan or package uses the CALL *literal* form of the CALL statement to call the stored procedure
- When a table on which the plan or 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 plan or package depends is altered
- When the definition of an index on which the plan or package depends is altered from NOT PADDED to PADDED
- When the AUDIT attribute of a table on which the plan or package depends is altered
- When the length attribute of a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC column in a table on which the plan or package depends is altered
- When the data type, precision, or scale of a column in a table on which the plan or package depends is altered
- When a plan or 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 plan or package depends is altered to add a column
- When a user-defined function on which the plan or package depends is altered

Whether a plan or package is valid is recorded in column VALID of catalog tables SYSPLAN and SYSPACKAGE.

In the following cases, DB2 might automatically rebind a plan or package that has not been marked as invalid:

- A plan or package that is bound on release of DB2 that is more recent than the release in which it is being run. This can happen in a data sharing environment, or it can happen after a DB2 subsystem has fallen back to a previous release of DB2.
- A plan or package that was bound prior to DB2 Version 2 Release 3. Plans and packages that are bound prior to Version 2 Release 3 will be automatically rebound when they are run on the current release of DB2.
- A plan or package that has a location dependency and runs at a location other than the one at which it was bound. This 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 plan or package that has not been marked as invalid if the ABIND subsystem parameter is set to COEXIST:

- The subsystem on which the plan or package runs is in a data sharing group.

#

- The plan or package was previously bound on the current DB2 release and is now running on the previous DB2 release.

DB2 marks a plan or package as *inoperative* if an automatic rebind fails. Whether a plan or 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.

Chapter 18. Planning for concurrency

This chapter begins with an overview of concurrency and locks in the following sections:

- “Definitions of concurrency and locks”
- “Effects of DB2 locks” on page 394
- “Basic recommendations to promote concurrency” on page 397

After the basic recommendations, the chapter covers some of the major techniques that DB2 uses to control concurrency:

- **Transaction locks** mainly control access by SQL statements. Those locks are the ones over which you have the most control.
 - “Aspects of transaction locks” on page 402 describes the various types of transaction locks that DB2 uses and how they interact.
 - “Options for tuning locks” on page 408 describes what you can change to control locking. Your choices include:
 - “Bind options” on page 408
 - “Isolation overriding with SQL statements” on page 421
 - “The LOCK TABLE statement” on page 422

Under those headings, *lock* (with no qualifier) refers to *transaction lock*.

- **LOB locks** are described separately from transaction locks because the purpose of LOB locks is different than that of regular transition locks. See “LOB locks” on page 425.
- **Claims and drains** control access by DB2 utilities and commands. For information about them, see *DB2 Administration Guide*.
- **Physical locks** are of concern only if you are using DB2 data sharing. For information about that, see *DB2 Data Sharing: Planning and Administration*.

Definitions of concurrency and locks

Definition: *Concurrency* is the ability of more than one application process to access the same data at essentially the same time.

Example: An application for order entry is used by many transactions simultaneously. Each transaction makes inserts in tables of invoices and invoice items, reads a table of data about customers, and reads and updates data about items on hand. Two operations on the same data, by two simultaneous transactions, might be separated only by microseconds. To the users, the operations appear concurrent.

Conceptual background: Concurrency must be controlled to prevent lost updates and such possibly undesirable effects as unrepeatable reads and access to uncommitted data.

Lost updates. Without concurrency control, two processes, A and B, might both read the same row from the database, and both calculate new values for one of its columns, based on what they read. If A updates the row with its new value, and then B updates the same row, A’s update is lost.

Access to uncommitted data. Also without concurrency control, process A might update a value in the database, and process B might read that value

before it was committed. Then, if A's value is not later committed, but backed out, B's calculations are based on uncommitted (and presumably incorrect) data.

Unrepeatable reads. Some processes require the following sequence of events: A reads a row from the database and then goes on to process other SQL requests. Later, A reads the first row again and must find the same values it read the first time. Without control, process B could have changed the row between the two read operations.

To prevent those situations from occurring unless they are specifically allowed, DB2 might use *locks* to control concurrency.

What do locks do? A lock associates a DB2 resource with an application process in a way that affects how other processes can access the same resource. The process associated with the resource is said to "hold" or "own" the lock. DB2 uses locks to ensure that no process accesses data that has been changed, but not yet committed, by another process.

What do you do about locks? To preserve data integrity, your application process acquires locks implicitly, that is, under DB2 control. It is not necessary for a process to request a lock explicitly to conceal uncommitted data. Therefore, sometimes you need not do anything about DB2 locks. Nevertheless processes acquire, or avoid acquiring, locks based on certain general parameters. You can make better use of your resources and improve concurrency by understanding the effects of those parameters.

Effects of DB2 locks

The effects of locks that you want to minimize are *suspension*, *timeout*, and *deadlock*.

Suspension

Definition: An application process is *suspended* when it requests a lock that is already held by another application process and cannot be shared. The suspended process temporarily stops running.

Order of precedence for lock requests: Incoming lock requests are queued. Requests for lock promotion, and requests for a lock by an application process that already holds a lock on the same object, precede requests for locks by new applications. Within those groups, the request order is "first in, first out".

Example: Using an application for inventory control, two users attempt to reduce the quantity on hand of the same item at the same time. The two lock requests are queued. The second request in the queue is suspended and waits until the first request releases its lock.

Effects: The suspended process resumes running when:

- All processes that hold the conflicting lock release it.
- The requesting process times out or deadlocks and the process resumes to deal with an error condition.

Timeout

Definition: An application process is said to *time out* when it is terminated because it has been suspended for longer than a preset interval.

Example: An application process attempts to update a large table space that is being reorganized by the utility REORG TABLESPACE with SHRLEVEL NONE. It is likely that the utility job will not release control of the table space before the application process times out.

Effects: DB2 terminates the process, issues two messages to the console, and returns SQLCODE -911 or -913 to the process (SQLSTATES '40001' or '57033'). Reason code 00C9008E is returned in the SQLERRD(3) field of the SQLCA. Alternatively, you can use the GET DIAGNOSTICS statement to check the reason code. If statistics trace class 3 is active, DB2 writes a trace record with IFCID 0196.

IMS

If you are using IMS, and a timeout occurs, the following actions take place:

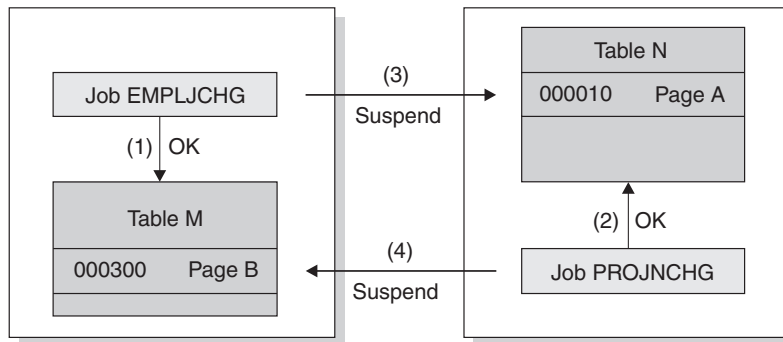
- In a DL/I batch application, the application process abnormally terminates with a completion code of 04E and a reason code of 00D44033 or 00D44050.
- In any IMS environment except DL/I batch:
 - DB2 performs a rollback operation on behalf of your application process to undo all DB2 updates that occurred during the current unit of work.
 - For a non-message driven BMP, IMS issues a rollback operation on behalf of your application. If this operation is successful, IMS returns control to your application, and the application receives SQLCODE -911. If the operation is unsuccessful, IMS issues user abend code 0777, and the application does not receive an SQLCODE.
 - For an MPP, IFP, or message driven BMP, IMS issues user abend code 0777, rolls back all uncommitted changes, and reschedules the transaction. The application does not receive an SQLCODE.

COMMIT and ROLLBACK operations do not time out. The command STOP DATABASE, however, may time out and send messages to the console, but it will retry up to 15 times.

Deadlock

Definition: A *deadlock* occurs when two or more application processes each hold locks on resources that the others need and without which they cannot proceed.

Example: Figure 141 on page 396 illustrates a deadlock between two transactions.



Notes:

1. Jobs EMPLJCHG and PROJNCHG are two transactions. Job EMPLJCHG accesses table M, and acquires an exclusive lock for page B, which contains record 000300.
2. Job PROJNCHG accesses table N, and acquires an exclusive lock for page A, which contains record 000010.
3. Job EMPLJCHG requests a lock for page A of table N while still holding the lock on page B of table M. The job is suspended, because job PROJNCHG is holding an exclusive lock on page A.
4. Job PROJNCHG requests a lock for page B of table M while still holding the lock on page A of table N. The job is suspended, because job EMPLJCHG is holding an exclusive lock on page B. The situation is a deadlock.

Figure 141. A deadlock example

Effects: After a preset time interval (the value of DEADLOCK TIME), DB2 can roll back the current unit of work for one of the processes or request a process to terminate. That frees the locks and allows the remaining processes to continue. If statistics trace class 3 is active, DB2 writes a trace record with IFCID 0172. Reason code 00C90088 is returned in the SQLERRD(3) field of the SQLCA. Alternatively, you can use the GET DIAGNOSTICS statement to check the reason code. (The codes that describe DB2's exact response depend on the operating environment; for details, see Part 5 of *DB2 Application Programming and SQL Guide*.)

It is possible for two processes to be running on distributed DB2 subsystems, each trying to access a resource at the other location. In that case, neither subsystem can detect that the two processes are in deadlock; the situation resolves only when one process times out.

Indications of deadlocks: In some cases, a deadlock can occur if two application processes attempt to update data in the same page or table space.

TSO, Batch, and CAF

When a deadlock or timeout occurs in these environments, DB2 attempts to roll back the SQL for one of the application processes. If the ROLLBACK is successful, that application receives SQLCODE -911. If the ROLLBACK fails, and the application does not abend, the application receives SQLCODE -913.

IMS

If you are using IMS, and a deadlock occurs, the following actions take place:

- In a DL/I batch application, the application process abnormally terminates with a completion code of 04E and a reason code of 00D44033 or 00D44050.
- In any IMS environment except DL/I batch:
 - DB2 performs a rollback operation on behalf of your application process to undo all DB2 updates that occurred during the current unit of work.
 - For a non-message driven BMP, IMS issues a rollback operation on behalf of your application. If this operation is successful, IMS returns control to your application, and the application receives SQLCODE -911. If the operation is unsuccessful, IMS issues user abend code 0777, and the application does not receive an SQLCODE.
 - For an MPP, IFP, or message driven BMP, IMS issues user abend code 0777, rolls back all uncommitted changes, and reschedules the transaction. The application does not receive an SQLCODE.

CICS

If you are using CICS and a deadlock occurs, the CICS attachment facility decides whether or not to roll back one of the application processes, based on the value of the ROLBE or ROLBI parameter. If your application process is chosen for rollback, it receives one of two SQLCODEs in the SQLCA:

-911	A SYNCPOINT command with the ROLLBACK option was issued on behalf of your application process. All updates (CICS commands and DL/I calls, as well as SQL statements) that occurred during the current unit of work have been undone. (SQLSTATE '40001')
-913	A SYNCPOINT command with the ROLLBACK option was not issued. DB2 rolls back only the incomplete SQL statement that encountered the deadlock or timed out. CICS does not roll back any resources. Your application process should either issue a SYNCPOINT command with the ROLLBACK option itself or terminate. (SQLSTATE '57033')

Consider using the DSNTIAC subroutine to check the SQLCODE and display the SQLCA. Your application must take appropriate actions before resuming.

Basic recommendations to promote concurrency

Recommendations are grouped by their scope:

- “Recommendations for database design” on page 398
- “Recommendations for application design” on page 399

Recommendations for database design

Keep like things together: Put tables relevant to the same application into the same database. Give each application process that creates private tables a private database. Put tables together in a segmented table space if they are similar in size and can be recovered together.

Keep unlike things apart: Use an adequate number of databases, schema or authorization-ID qualifiers, and table spaces. Concurrency and performance is improved for SQL data definition statements, GRANT statements, REVOKE statements, and utilities. For example, a general guideline is a maximum of 50 tables per database.

Plan for batch inserts: If your application does sequential batch insertions, excessive contention on the space map pages for the table space can occur. This problem is especially apparent in data sharing, where contention on the space map means the added overhead of page P-lock negotiation. For these types of applications, consider using the MEMBER CLUSTER option of CREATE TABLESPACE. This option causes DB2 to disregard the clustering index (or implicit clustering index) when assigning space for the SQL INSERT statement. For more information about using this option in data sharing, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*. For the syntax, see Chapter 5 of *DB2 SQL Reference*.

Use LOCKSIZE ANY until you have reason not to: LOCKSIZE ANY is the default for CREATE TABLESPACE. It allows DB2 to choose the lock size, and DB2 usually chooses LOCKSIZE PAGE and LOCKMAX SYSTEM for non-LOB table spaces. For LOB table spaces, it chooses LOCKSIZE LOB and LOCKMAX SYSTEM. You should use LOCKSIZE TABLESPACE or LOCKSIZE TABLE only for read-only table spaces or tables, or when concurrent access to the object is not needed. Before you choose LOCKSIZE ROW, you should estimate whether there will be an increase in overhead for locking and weigh that against the increase in concurrency.

Examine small tables: For small tables with high concurrency requirements, estimate the number of pages in the data and in the index. If the index entries are short or they have many duplicates, then the entire index can be one root page and a few leaf pages. In this case, spread out your data to improve concurrency, or consider it a reason to use row locks.

Partition the data: Large tables can be partitioned to take advantage of parallelism for online queries, batch jobs, and utilities. When batch jobs are run in parallel and each job goes after different partitions, lock contention is reduced. In addition, in a data sharing environment, data sharing overhead is reduced when applications that are running on different members go after different partitions.

Partition secondary indexes: The use of data-partitioned secondary indexes (DPSIs) promotes partition independence and, therefore, can reduce lock contention and improve index availability, especially for utility processing, partition-level operations (such as dropping or rotating partitions), and recovery of indexes.

However, the use of data-partitioned secondary indexes does not always improve the performance of queries. For example, for a query with a predicate that references only the columns of a data-partitioned secondary index, DB2 must probe each partition of the index for values that satisfy the predicate if index access is chosen as the access path. Therefore, take into account data access patterns and maintenance practices when deciding to use a data-partitioned secondary index.

Replace a nonpartitioned index with a partitioned index only if there are perceivable benefits such as improved data or index availability, easier data or index maintenance, or improved performance.

For examples of how query performance can be improved with data-partitioned secondary indexes, see “Writing efficient queries on tables with data-partitioned secondary indexes” on page 772.

Fewer rows of data per page: By using the MAXROWS clause of CREATE or ALTER TABLESPACE, you can specify the maximum number of rows that can be on a page. For example, if you use MAXROWS 1, each row occupies a whole page, and you confine a page lock to a single row. Consider this option if you have a reason to avoid using row locking, such as in a data sharing environment where row locking overhead can be greater.

Consider volatile tables to ensure index access: If multiple applications access the same table, consider defining the table as VOLATILE. DB2 uses index access whenever possible for volatile tables, even if index access does not appear to be the most efficient access method because of volatile statistics. Because each application generally accesses the rows in the table in the same order, lock contention can be reduced.

Recommendations for application design

Access data in a consistent order: When different applications access the same data, try to make them do so in the same sequence. For example, make both access rows 1,2,3,5 in that order. In that case, the first application to access the data delays the second, but the two applications cannot deadlock. For the same reason, try to make different applications access the same tables in the same order.

Commit work as soon as is practical: To avoid unnecessary lock contention, issue a COMMIT statement as soon as possible after reaching a point of consistency, even in read-only applications. To prevent unsuccessful SQL statements (such as PREPARE) from holding locks, issue a ROLLBACK statement after a failure. Statements issued through SPUFI can be committed immediately by the SPUFI autocommit feature.

Taking commit points frequently in a long running unit of recovery (UR) has the following benefits at the possible cost of more CPU usage and log write I/Os:

- Reduces lock contention, especially in a data sharing environment
- Improves the effectiveness of lock avoidance, especially in a data sharing environment
- Reduces the elapsed time for DB2 system restart following a system failure
- Reduces the elapsed time for a unit of recovery to rollback following an application failure or an explicit rollback request by the application
- Provides more opportunity for utilities, such as online REORG, to break in

Consider using the UR CHECK FREQ field or the UR LOG WRITE CHECK field of installation panel DSNTIPN to help you identify those applications that are not committing frequently. UR CHECK FREQ, which identifies when too many checkpoints have occurred without a UR issuing a commit, is helpful in monitoring overall system activity. UR LOG WRITE CHECK enables you to detect applications that might write too many log records between commit points, potentially creating a lengthy recovery situation for critical tables.

Even though an application might conform to the commit frequency standards of the installation under normal operational conditions, variation can occur based on system workload fluctuations. For example, a low-priority application might issue a commit frequently on a system that is lightly loaded. However, under a heavy system load, the use of the CPU by the application may be pre-empted, and, as a result, the application may violate the rule set by the UR CHECK FREQ parameter. For this reason, add logic to your application to commit based on time elapsed since last commit, and not solely based on the amount of SQL processing performed. In addition, take frequent commit points in a long running unit of work that is read-only to reduce lock contention and to provide opportunities for utilities, such as online REORG, to access the data.

Retry an application after deadlock or timeout: Include logic in a batch program so that it retries an operation after a deadlock or timeout. Such a method could help you recover from the situation without assistance from operations personnel. Field SQLERRD(3) in the SQLCA returns a reason code that indicates whether a deadlock or timeout occurred. Alternatively, you can use the GET DIAGNOSTICS statement to check the reason code.

Close cursors: If you define a cursor using the WITH HOLD option, the locks it needs can be held past a commit point. Use the CLOSE CURSOR statement as soon as possible in your program to cause those locks to be released and the resources they hold to be freed at the first commit point that follows the CLOSE CURSOR statement. Whether page or row locks are held for WITH HOLD cursors is controlled by the RELEASE LOCKS parameter on installation panel DSNTIP4. Closing cursors is particularly important in a distributed environment.

Free locators: If you have executed the HOLD LOCATOR statement, the LOB locator holds locks on LOBs past commit points. Use the FREE LOCATOR statement to release these locks.

Bind plans with ACQUIRE(USE): ACQUIRE(USE), which indicates that DB2 will acquire table and table space locks when the objects are first used and not when the plan is allocated, is the best choice for concurrency. Packages are always bound with ACQUIRE(USE), by default. ACQUIRE(ALLOCATE) can provide better protection against timeouts. Consider ACQUIRE(ALLOCATE) for applications that need gross locks instead of intent locks or that run with other applications that may request gross locks instead of intent locks. Acquiring the locks at plan allocation also prevents any one transaction in the application from incurring the cost of acquiring the table and table space locks. If you need ACQUIRE(ALLOCATE), you might want to bind all DBRMs directly to the plan.

For information about intent and gross locks, see “The mode of a lock” on page 404.

Bind with ISOLATION(CS) and CURRENTDATA(NO) typically: ISOLATION(CS) lets DB2 release acquired row and page locks as soon as possible. CURRENTDATA(NO) lets DB2 avoid acquiring row and page locks as often as possible. After that, in order of decreasing preference for concurrency, use these bind options:

1. ISOLATION(CS) with CURRENTDATA(YES), when data returned to the application must not be changed before your next FETCH operation.
2. ISOLATION(RS), when data returned to the application must not be changed before your application commits or rolls back. However, you do not care if other application processes insert additional rows.

3. ISOLATION(RR), when data evaluated as the result of a query must not be changed before your application commits or rolls back. New rows cannot be inserted into the answer set.

For more information about the ISOLATION option, see “The ISOLATION option” on page 412.

For updatable static scrollable cursors, ISOLATION(CS) provides the additional advantage of letting DB2 use *optimistic concurrency control* to further reduce the amount of time that locks are held. With optimistic concurrency control, DB2 releases the row or page locks on the base table after it materializes the result table in a temporary global table. DB2 also releases the row lock after each FETCH, taking a new lock on a row only for a positioned update or delete to ensure data integrity. For more information about optimistic concurrency control, see “Advantages and disadvantages of the isolation values” on page 412.

For updatable dynamic scrollable cursors and ISOLATION(CS), DB2 holds row or page locks on the base table (DB2 does not use a temporary global table). The most recently fetched row or page from the base table remains locked to maintain data integrity for a positioned update or delete.

Use ISOLATION(UR) cautiously: UR isolation acquires almost no locks on rows or pages. It is fast and causes little contention, but it reads uncommitted data. Do not use it unless you are sure that your applications and end users can accept the logical inconsistencies that can occur.

For information on how to make an agent part of a global transaction for RRSAF applications, see Chapter 31, “Programming for the Resource Recovery Services attachment facility,” on page 893.

Use sequence objects to generate unique, sequential numbers: Using an identity column is one way to generate unique sequential numbers. However, as a column of a table, an identity column is associated with and tied to the table, and a table can have only one identity column. Your applications might need to use one sequence of unique numbers for many tables or several sequences for each table. As a user-defined object, sequences provide a way for applications to have DB2 generate unique numeric key values and to coordinate the keys across multiple rows and tables.

The use of sequences can avoid the lock contention problems that can result when applications implement their own sequences, such as in a one-row table that contains a sequence number that each transaction must increment. With DB2 sequences, many users can access and increment the sequence concurrently without waiting. DB2 does not wait for a transaction that has incremented a sequence to commit before allowing another transaction to increment the sequence again.

Examine multi-row operations: In an application, multi-row inserts, positioned updates, and positioned deletes have the potential of expanding the unit of work. This can affect the concurrency of other users accessing the data. Minimize contention by adjusting the size of the host-variable-array, committing between inserts, updates, and preventing lock escalation.

Use global transactions: The Resource Recovery Services attachment facility (RRSAF) relies on an z/OS component called Resource Recovery Services (RRS). RRS provides system-wide services for coordinating two-phase commit operations

across z/OS products. For RRSAF applications and IMS transactions that run under RRS, you can group together a number of DB2 agents into a single global transaction. A global transaction allows multiple DB2 agents to participate in a single global transaction and thus share the same locks and access the same data. When two agents that are in a global transaction access the same DB2 object within a unit of work, those agents will not deadlock with each other. The following restrictions apply:

- There is no Parallel Sysplex support for global transactions.
- Because each of the "branches" of a global transaction are sharing locks, uncommitted updates issued by one branch of the transaction are visible to other branches of the transaction.
- Claim/drain processing is not supported across the branches of a global transaction, which means that attempts to issue CREATE, DROP, ALTER, GRANT, or REVOKE may deadlock or timeout if they are requested from different branches of the same global transaction.
- LOCK TABLE may deadlock or timeout across the branches of a global transaction.

Aspects of transaction locks

Transaction locks have the following four basic aspects:

- "The size of a lock"
- "The duration of a lock" on page 404
- "The mode of a lock" on page 404
- "The object of a lock" on page 407

Knowing the aspects helps you understand why a process suspends or times out or why two processes deadlock.

The size of a lock

Definition

The *size* (sometimes *scope* or *level*) of a lock on data in a table describes the amount of data controlled. The possible sizes of locks are table space, table, partition, page, and row. This section contains information about locking for non-LOB data. See "LOB locks" on page 425 for information on locking for LOBs.

Hierarchy of lock sizes

The same piece of data can be controlled by locks of different sizes. A table space lock (the largest size) controls the most data, all the data in an entire table space. A page or row lock controls only the data in a single page or row.

As Figure 142 on page 403 suggests, row locks and page locks occupy an equal place in the hierarchy of lock sizes.

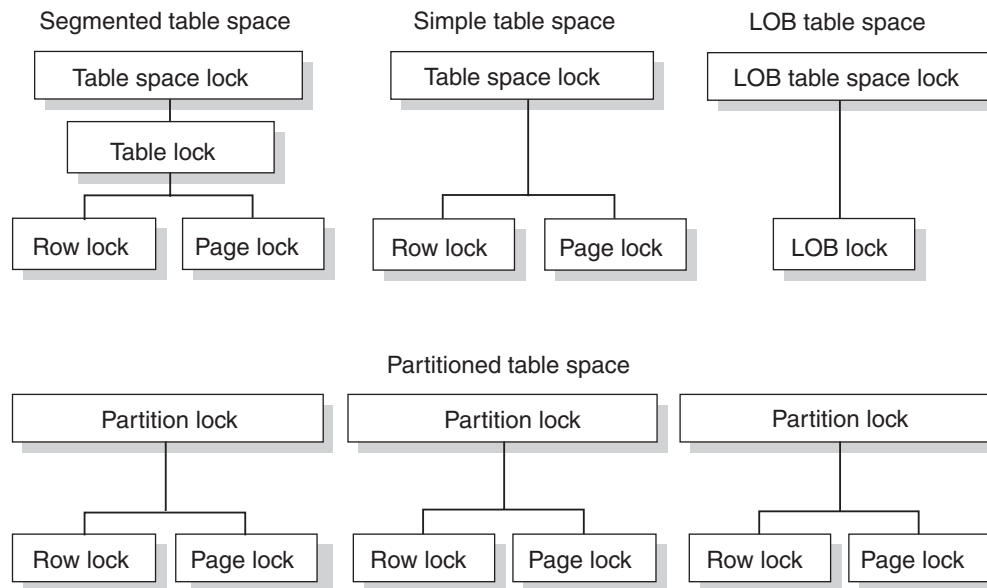


Figure 142. Sizes of objects locked

General effects of size

Locking larger or smaller amounts of data allows you to trade performance for concurrency. Using page or row locks instead of table or table space locks has the following effects:

- Concurrency usually improves, meaning better response times and higher throughput rates for many users.
- Processing time and use of storage increases. That is especially evident in batch processes that scan or update a large number of rows.

Using only table or table space locks has the following effects:

- Processing time and storage usage is reduced.
- Concurrency can be reduced, meaning longer response times for some users but better throughput for one user.

Effects of table spaces of different types

- In a **partitioned table space**, locks are obtained at the partition level. Individual partitions are locked as they are accessed. This locking behavior enables greater concurrency because gross locks (S, U, or X) can be obtained on individual partitions instead of on the entire partitioned table space. (Partition locks are always acquired, even if the table space was defined in a version of DB2 prior to Version 8 with the LOCKPART NO clause. For LOCKPART NO table spaces, DB2 no longer locks the entire table space with one lock when any partition of the table space is accessed.)

Restrictions: If any of the following conditions are true, DB2 must lock *all* partitions:

- The plan is bound with ACQUIRE(ALLOCATE).
- The table space is defined with LOCKSIZE TABLESPACE.
- The LOCK TABLE statement is used without the PART option.

- A **simple table space** can contain more than one table. A lock on the table space locks all the data in every table. A single page of the table space can contain rows from every table. A lock on a page locks every row in the page, no matter what tables the data belongs to. Thus, a lock needed to access data from one

table can make data from other tables temporarily unavailable. That effect can be partly undone by using row locks instead of page locks.

- In a **segmented table space**, rows from different tables are contained in different pages. Locking a page does not lock data from more than one table. Also, DB2 can acquire a table lock, which locks only the data from one specific table. Because a single row, of course, contains data from only one table, the effect of a row lock is the same as for a simple or partitioned table space: it locks one row of data from one table.
- In a **LOB table space**, pages are not locked. Because there is no concept of a row in a LOB table space, rows are not locked. Instead, LOBs are locked. See “LOB locks” on page 425 for more information.

The duration of a lock

Definition: The *duration* of a lock is the length of time the lock is held. It varies according to when the lock is acquired and when it is released.

Effects

For maximum concurrency, locks on a small amount of data held for a short duration are better than locks on a large amount of data held for a long duration. However, acquiring a lock requires processor time, and holding a lock requires storage; thus, acquiring and holding one table space lock is more economical than acquiring and holding many page locks. Consider that trade-off to meet your performance and concurrency objectives.

Duration of partition, table, and table space locks: Partition, table, and table space locks can be acquired when a plan is first allocated, or you can delay acquiring them until the resource they lock is first used. They can be released at the next commit point or be held until the program terminates.

On the other hand, LOB table space locks are always acquired when needed and released at a commit or held until the program terminates. See “LOB locks” on page 425 for information about locking LOBs and LOB table spaces.

Duration of page and row locks: If a page or row is locked, DB2 acquires the lock only when it is needed. When the lock is released depends on many factors, but it is rarely held beyond the next commit point.

For information about controlling the duration of locks, see “Bind options” on page 408 for information about the ACQUIRE and RELEASE, ISOLATION, and CURRENTDATA bind options.

The mode of a lock

Definition: The *mode* (sometimes *state*) of a lock tells what access to the locked object is permitted to the lock owner and to any concurrent processes.

The possible modes for page and row locks and the modes for partition, table, and table space locks are listed in “Modes of page and row locks” on page 405 and “Modes of table, partition, and table space locks” on page 405. See “LOB locks” on page 425 for more information about modes for LOB locks and locks on LOB table spaces.

When a page or row is locked, the table, partition, or table space containing it is also locked. In that case, the table, partition, or table space lock has one of the *intent* modes: IS, IX, or SIX. The modes S, U, and X of table, partition, and table

space locks are sometimes called *gross* modes. In the context of reading, SIX is a gross mode lock because you don't get page or row locks; in this sense, it is like an S lock.

Example: An SQL statement locates John Smith in a table of customer data and changes his address. The statement locks the entire table space in mode IX and the specific row that it changes in mode X.

Modes of page and row locks

Modes and their effects are listed in the order of increasing control over resources.

S (SHARE) The lock owner and any concurrent processes can read, but not change, the locked page or row. Concurrent processes can acquire S or U locks on the page or row or might read data without acquiring a page or row lock.

U (UPDATE) The lock owner can read, but not change, the locked page or row. Concurrent processes can acquire S locks or might read data without acquiring a page or row lock, but no concurrent process can acquire a U lock.

U locks reduce the chance of deadlocks when the lock owner is reading a page or row to determine whether to change it, because the owner can start with the U lock and then promote the lock to an X lock to change the page or row.

X (EXCLUSIVE)

The lock owner can read or change the locked page or row. A concurrent process cannot acquire S, U, or X locks on the page or row. However, a concurrent process, such as those bound with the CURRENTDATA(NO) or ISO(UR) options or running with YES specified for the EVALUNC subsystem parameter, can read the data without acquiring a page or row lock.

Modes of table, partition, and table space locks

Modes and their effects are listed in the order of increasing control over resources.

IS (INTENT SHARE) The lock owner can read data in the table, partition, or table space, but not change it. Concurrent processes can both read and change the data. The lock owner might acquire a page or row lock on any data it reads.

IX (INTENT EXCLUSIVE) The lock owner and concurrent processes can read and change data in the table, partition, or table space. The lock owner might acquire a page or row lock on any data it reads; it must acquire one on any data it changes.

S (SHARE) The lock owner and any concurrent processes can read, but not change, data in the table, partition, or table space. The lock owner does not need page or row locks on data it reads.

U (UPDATE) The lock owner can read, but not change, the locked data; however, the owner can promote the lock to an X lock and then can change the data. Processes concurrent with the U lock can acquire S

locks and read the data, but no concurrent process can acquire a U lock. The lock owner does not need page or row locks.

U locks reduce the chance of deadlocks when the lock owner is reading data to determine whether to change it. U locks are acquired on a table space when locksize is TABLESPACE and the statement is a SELECT with a FOR UPDATE clause. Similarly, U locks are acquired on a table when lock size is TABLE and the statement is a SELECT with a FOR UPDATE clause.

SIX (SHARE with INTENT EXCLUSIVE)

The lock owner can read and change data in the table, partition, or table space. Concurrent processes can read data in the table, partition, or table space, but not change it. Only when the lock owner changes data does it acquire page or row locks.

X (EXCLUSIVE)

The lock owner can read or change data in the table, partition, or table space. A concurrent process can access the data if the process runs with UR isolation or if data in a partitioned table space is running with CS isolation and CURRENTDATA((NO). The lock owner does not need page or row locks.

Lock mode compatibility

The major effect of the lock mode is to determine whether one lock is compatible with another.

Definition: Locks of some modes do not shut out all other users. Assume that application process A holds a lock on a table space that process B also wants to access. DB2 requests, on behalf of B, a lock of some particular mode. If the mode of A's lock permits B's request, the two locks (or modes) are said to be *compatible*.

Effects of incompatibility: If the two locks are not compatible, B cannot proceed. It must wait until A releases its lock. (And, in fact, it must wait until all existing incompatible locks are released.)

Compatible lock modes: Compatibility for page and row locks is easy to define. Table 45 shows whether page locks of any two modes, or row locks of any two modes, are compatible (Yes) or not (No). No question of compatibility of a page lock with a row lock can arise, because a table space cannot use both page and row locks.

Table 45. Compatibility of page lock and row lock modes

Lock Mode	S	U	X
S	Yes	Yes	No
U	Yes	No	No
X	No	No	No

Compatibility for table space locks is slightly more complex. Table 46 on page 407 shows whether or not table space locks of any two modes are compatible.

Table 46. Compatibility of table and table space (or partition) lock modes

Lock Mode	IS	IX	S	U	SIX	X
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No	No
S	Yes	No	Yes	Yes	No	No
U	Yes	No	Yes	No	No	No
SIX	Yes	No	No	No	No	No
X	No	No	No	No	No	No

The object of a lock

Definition and examples

The *object* of a lock is the resource being locked.

You might have to consider locks on any of the following objects:

- **User data in target tables.** A *target table* is a table that is accessed specifically in an SQL statement, either by name or through a view. Locks on those tables are the most common concern, and the ones over which you have most control.
- **User data in related tables.** Operations subject to referential constraints can require locks on related tables. For example, if you delete from a parent table, DB2 might delete rows from the dependent table as well. In that case, DB2 locks data in the dependent table as well as in the parent table.

Similarly, operations on rows that contain LOB values might require locks on the LOB table space and possibly on LOB values within that table space. See “LOB locks” on page 425 for more information.

If your application uses triggers, any triggered SQL statements can cause additional locks to be acquired.

- **DB2 internal objects.** Most of these you are never aware of, but you might notice the following locks on internal objects:
 - Portions of the **DB2 catalog**
 - The **skeleton cursor table** (SKCT) representing an application plan
 - The **skeleton package table** (SKPT) representing a package
 - The **database descriptor** (DBD) representing a DB2 database

For information about any of those, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Indexes and data-only locking

No index page locks are acquired during processing. Instead, DB2 uses a technique called *data-only locking* to serialize changes. Index page latches are acquired to serialize changes within a page and guarantee that the page is physically consistent. Acquiring page latches ensures that transactions accessing the same index page concurrently do not see the page in a partially changed state.

The underlying data page or row locks are acquired to serialize the reading and updating of index entries to ensure the data is logically consistent, meaning that the data is committed and not subject to rollback or abort. The data locks can be held for a long duration such as until commit. However, the page latches are only held for a short duration while the transaction is accessing the page. Because the index pages are not locked, hot spot insert scenarios (which involve several transactions trying to insert different entries into the same index page at the same time) do not cause contention problems in the index.

A query that uses index-only access might lock the data page or row, and that lock can contend with other processes that lock the data. However, using lock avoidance techniques can reduce the contention. See “Lock avoidance” on page 418 for more information about lock avoidance.

Options for tuning locks

The following sections describe the options that affect transaction locks for applications:

- “Bind options”
- “Isolation overriding with SQL statements” on page 421
- “The LOCK TABLE statement” on page 422

Bind options

These options determine when an application process acquires and releases its locks and to what extent it isolates its actions from possible effects of other processes acting concurrently.

These options of bind operations are relevant to transaction locks:

- “The ACQUIRE and RELEASE options”
- “The ISOLATION option” on page 412
- “The CURRENTDATA option” on page 417

The ACQUIRE and RELEASE options

Effects: The ACQUIRE and RELEASE options of bind determine when DB2 locks an object (table, partition, or table space) your application uses and when it releases the lock. (The ACQUIRE and RELEASE options do not affect page, row, or LOB locks.) The options apply to static SQL statements, which are bound before your program executes. If your program executes dynamic SQL statements, the objects they lock are locked when first accessed and released at the next commit point though some locks acquired for dynamic SQL may be held past commit points. See 409.

ACQUIRE(ALLOCATE)

Acquires the lock when the object is allocated. This option is not allowed for BIND or REBIND PACKAGE.

ACQUIRE(USE)

Acquires the lock when the object is first accessed.

RELEASE(DEALLOCATE)

Releases the lock when the object is deallocated (the application ends). The value has no effect on dynamic SQL statements, which always use RELEASE(COMMIT), unless you are using dynamic statement caching. For information about the RELEASE option with dynamic statement caching, see 409. The value also has no effect on packages that are executed on a DB2 server through a DRDA connection with the client system.

RELEASE(COMMIT)

Releases the lock at the next commit point, unless there are held cursors or held locators. If the application accesses the object again, it must acquire the lock again.

Example: An application selects employee names and telephone numbers from a table, according to different criteria. Employees can update their own telephone

numbers. They can perform several searches in succession. The application is bound with the options ACQUIRE(USE) and RELEASE(DEALLOCATE), for these reasons:

- The alternative to ACQUIRE(USE), ACQUIRE(ALLOCATE), gets a lock of mode IX on the table space as soon as the application starts, because that is needed if an update occurs. But most uses of the application do not update the table and so need only the less restrictive IS lock. ACQUIRE(USE) gets the IS lock when the table is first accessed, and DB2 promotes the lock to mode IX if that is needed later.
- Most uses of this application do not update and do not commit. For those uses, there is little difference between RELEASE(COMMIT) and RELEASE(DEALLOCATE). But administrators might update several phone numbers in one session with the application, and the application commits after each update. In that case, RELEASE(COMMIT) releases a lock that DB2 must acquire again immediately. RELEASE(DEALLOCATE) holds the lock until the application ends, avoiding the processing needed to release and acquire the lock several times.

Partition locks: Partition locks follow the same rules as table space locks, and **all** partitions are held for the same duration. Thus, if one package is using RELEASE(COMMIT) and another is using RELEASE(DEALLOCATE), all partitions use RELEASE(DEALLOCATE).

The RELEASE option and dynamic statement caching: Generally, the RELEASE option has no effect on dynamic SQL statements with one exception. When you use the bind options RELEASE(DEALLOCATE) and KEEP DYNAMIC(YES), and your subsystem is installed with YES for field CACHE DYNAMIC SQL on installation panel DSNTIP4, DB2 retains prepared SELECT, INSERT, UPDATE, and DELETE statements in memory past commit points. For this reason, DB2 can honor the RELEASE(DEALLOCATE) option for these dynamic statements. The locks are held until deallocation, or until the commit after the prepared statement is freed from memory, in the following situations:

- The application issues a PREPARE statement with the same statement identifier.
- The statement is removed from memory because it has not been used.
- An object that the statement is dependent on is dropped or altered, or a privilege needed by the statement is revoked.
- RUNSTATS is run against an object that the statement is dependent on.

If a lock is to be held past commit and it is an S, SIX, or X lock on a table space or a table in a segmented table space, DB2 sometimes demotes that lock to an intent lock (IX or IS) at commit. DB2 demotes a gross lock if it was acquired for one of the following reasons:

- DB2 acquired the gross lock because of lock escalation.
- The application issued a LOCK TABLE.
- The application issued a mass delete (DELETE FROM ... without a WHERE clause).

For partitioned table spaces, lock demotion occurs for each partition for which there is a lock.

Defaults: The defaults differ for different types of bind operations, as shown in Table 47 on page 410.

Table 47. Default ACQUIRE and RELEASE values for different bind options

Operation	Default values
BIND PLAN	ACQUIRE(USE) and RELEASE(COMMIT).
BIND PACKAGE	There is no option for ACQUIRE; ACQUIRE(USE) is always used. At the local server the default for RELEASE is the value used by the plan that includes the package in its package list. At a remote server the default is COMMIT.
REBIND PLAN or PACKAGE	The existing values for the plan or package that is being rebound.

Recommendation: Choose a combination of values for ACQUIRE and RELEASE based on the characteristics of the particular application.

The RELEASE option and DDL operations for remote requesters: When you perform DDL operations on behalf of remote requesters and RELEASE(DEALLOCATE) is in effect, be aware of the following condition. When a package that is bound with RELEASE(DEALLOCATE) accesses data at a server, it might prevent other remote requesters from performing CREATE, ALTER, DROP, GRANT, or REVOKE operations at the server.

To allow those operations to complete, you can use the command STOP DDF MODE(SUSPEND). The command suspends server threads and terminates their locks so that DDL operations from remote requesters can complete. When these operations complete, you can use the command START DDF to resume the suspended server threads. However, even after the command STOP DDF MODE(SUSPEND) completes successfully, database resources might be held if DB2 is performing any activity other than inbound DB2 processing. You might have to use the command CANCEL THREAD to terminate other processing and thereby free the database resources.

Advantages and disadvantages of the combinations

Different combinations of bind options have advantages and disadvantages for certain situations.

ACQUIRE(ALLOCATE) / RELEASE(DEALLOCATE): In some cases, this combination can avoid deadlocks by locking all needed resources as soon as the program starts to run. This combination is most useful for a long-running application that runs for hours and accesses various tables, because it prevents an untimely deadlock from wasting that processing.

- All tables or table spaces used in DBRMs bound directly to the plan are locked when the plan is allocated.
- All tables or table spaces are unlocked only when the plan terminates.
- The locks used are the most restrictive needed to execute all SQL statements in the plan regardless of whether the statements are actually executed.
- Restrictive states are not checked until the page set is accessed. Locking when the plan is allocated insures that the job is compatible with other SQL jobs. Waiting until the first access to check restrictive states provides greater availability; however, it is possible that an SQL transaction could:
 - Hold a lock on a table space or partition that is stopped
 - Acquire a lock on a table space or partition that is started for DB2 utility access only (ACCESS(UT))

- Acquire an exclusive lock (IX, X) on a table space or partition that is started for read access only (ACCESS(RO)), thus prohibiting access by readers

Disadvantages: This combination reduces concurrency. It can lock resources in high demand for longer than needed. Also, the option ACQUIRE(ALLOCATE) turns off selective partition locking; if you are accessing a partitioned table space, all partitions are locked.

Restriction: This combination is not allowed for BIND PACKAGE. Use this combination if processing efficiency is more important than concurrency. It is a good choice for batch jobs that would release table and table space locks only to reacquire them almost immediately. It might even improve concurrency, by allowing batch jobs to finish sooner. Generally, do not use this combination if your application contains many SQL statements that are often not executed.

ACQUIRE(USE) / RELEASE(DEALLOCATE): This combination results in the most efficient use of processing time in most cases.

- A table, partition, or table space used by the plan or package is locked only if it is needed while running.
- All tables or table spaces are unlocked only when the plan terminates.
- The least restrictive lock needed to execute each SQL statement is used, with the exception that if a more restrictive lock remains from a previous statement, that lock is used without change.

Disadvantages: This combination can increase the frequency of deadlocks. Because all locks are acquired in a sequence that is predictable only in an actual run, more concurrent access delays might occur.

ACQUIRE(USE) / RELEASE(COMMIT): This combination is the default combination and provides the greatest concurrency, but it requires more processing time if the application commits frequently.

- A table or table space is locked only when needed. That locking is important if the process contains many SQL statements that are rarely used or statements that are intended to access data only in certain circumstances.
- All tables and table spaces are unlocked when:

TSO, Batch, and CAF

An SQL COMMIT or ROLLBACK statement is issued, or your application process terminates

IMS

A CHKP or SYNC call (for single-mode transactions), a GU call to the I/O PCB, or a ROLL or ROLB call is completed

CICS

A SYNCPOINT command is issued.

Exception: If the cursor is defined WITH HOLD, table or table space locks necessary to maintain cursor position are held past the commit point. (See “The effect of WITH HOLD for a cursor” on page 420 for more information.)

- The least restrictive lock needed to execute each SQL statement is used except when a more restrictive lock remains from a previous statement. In that case, that lock is used without change.

Disadvantages: This combination can increase the frequency of deadlocks. Because all locks are acquired in a sequence that is predictable only in an actual run, more concurrent access delays might occur.

ACQUIRE(ALLOCATE) / RELEASE(COMMIT): This combination is not allowed; it results in an error message from BIND.

The ISOLATION option

Effects: The ISOLATION option Specifies the degree to which operations are isolated from the possible effects of other operations acting concurrently. Based on this information, DB2 releases S and U locks on rows or pages as soon as possible. You can use the following ISOLATION options:

Default: The default differs for different types of bind operations, as shown in.

Table 48. The default ISOLATION values for different bind operations

Operation	Default value
BIND PLAN	ISOLATION(RR)
BIND PACKAGE	The value used by the plan that includes the package in its package list
REBIND PLAN or PACKAGE	The existing value for the plan or package being rebound

For more detailed examples, see *DB2 Application Programming and SQL Guide*.

Recommendation: Choose an ISOLATION value based on the characteristics of the particular application.

Advantages and disadvantages of the isolation values

The various isolation levels offer less or more concurrency at the cost of more or less protection from other application processes. The values you choose should be based primarily on the needs of the application. This section presents the isolation levels in order of recommendation, from the most recommended (CS) to the least recommended (RR). For ISOLATION (CS), CURRENTDATA(NO) is preferred over CURRENTDATA(YES). For more information on the CURRENTDATA option, see “The CURRENTDATA option” on page 417.

Regardless of the isolation level, uncommitted claims on DB2 objects can inhibit the execution of DB2 utilities or commands.

ISOLATION (CS)

Allows maximum concurrency with data integrity. However, after the process leaves a row or page, another process can change the data. With CURRENTDATA(NO), the process doesn't have to leave a row or page to allow another process to change the data. If the first process returns to read the same row or page, the data is not necessarily the same. Consider these consequences of that possibility:

- For table spaces created with LOCKSIZE ROW, PAGE, or ANY, a change can occur even while executing a single SQL statement, if the statement reads the same row more than once. In the following example:

```
SELECT * FROM T1
WHERE COL1 = (SELECT MAX(COL1) FROM T1);
```

data read by the inner SELECT can be changed by another transaction before it is read by the outer SELECT. Therefore, the information returned by this query might be from a row that is no longer the one with the maximum value for COL1.

- In another case, if your process reads a row and returns later to update it, that row might no longer exist or might not exist in the state that it did when your application process originally read it. That is, another application might have deleted or updated the row. **If your application is doing non-cursor operations on a row under the cursor, make sure the application can tolerate “not found” conditions.**

Similarly, assume another application updates a row after you read it. If your process returns later to update it based on the value you originally read, you are, in effect, erasing the update made by the other process. **If you use isolation (CS) with update, your process might need to lock out concurrent updates.** One method is to declare a cursor with the FOR UPDATE clause.

General-use Programming Interface

For packages and plans that contain updatable static scrollable cursors, ISOLATION(CS) lets DB2 use *optimistic concurrency control*. DB2 can use optimistic concurrency control to shorten the amount of time that locks are held in the following situations:

- Between consecutive fetch operations
- Between fetch operations and subsequent positioned update or delete operations

DB2 cannot use optimistic concurrency control for dynamic scrollable cursors. With dynamic scrollable cursors, the most recently fetched row or page from the base table remains locked to maintain position for a positioned update or delete.

Figure 143 and Figure 144 on page 414 show processing of positioned update and delete operations with static scrollable cursors without optimistic concurrency control and with optimistic concurrency control.

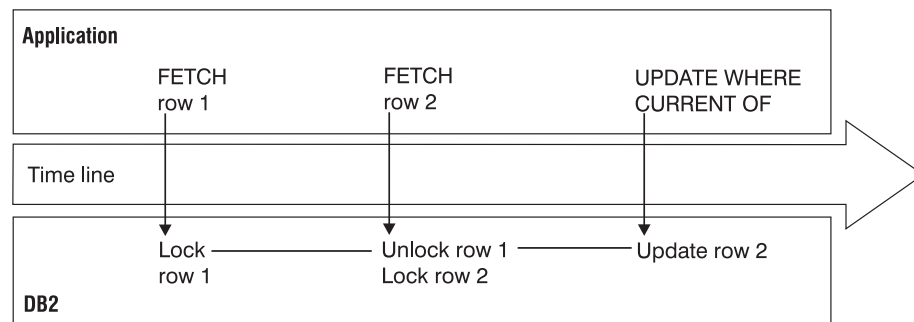


Figure 143. Positioned updates and deletes without optimistic concurrency control

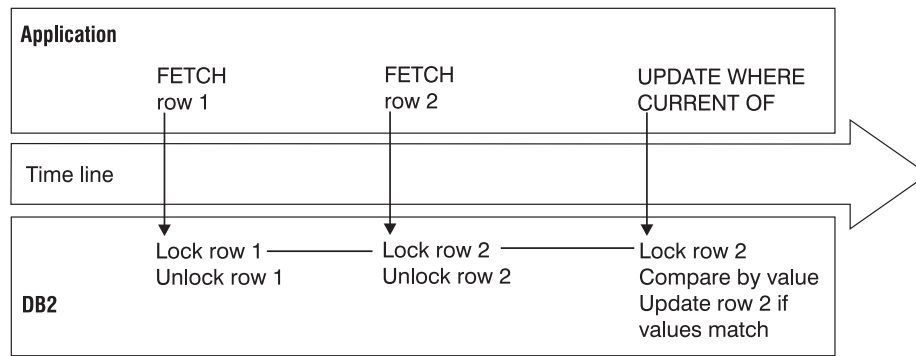


Figure 144. Positioned updates and deletes with optimistic concurrency control

Optimistic concurrency control consists of the following steps:

1. When the application requests a fetch operation to position the cursor on a row, DB2 locks that row, executes the FETCH, and releases the lock.
2. When the application requests a positioned update or delete operation on the row, DB2 performs the following steps:
 - a. Locks the row.
 - b. Reevaluates the predicate to ensure that the row still qualifies for the result table.
 - c. For columns that are in the result table, compares current values in the row to the values of the row when step 1 was executed. Performs the positioned update or delete operation only if the values match.

End of General-use Programming Interface

ISOLATION (UR)

Allows the application to read while acquiring few locks, at the risk of reading uncommitted data. UR isolation applies only to read-only operations: SELECT, SELECT INTO, or FETCH from a read-only result table.

Reading uncommitted data introduces an element of uncertainty.

Example: An application tracks the movement of work from station to station along an assembly line. As items move from one station to another, the application subtracts from the count of items at the first station and adds to the count of items at the second. Assume you want to query the count of items at all the stations, while the application is running concurrently.

What can happen if your query reads data that the application has changed but has not committed?

If the application subtracts an amount from one record before adding it to another, *the query could miss the amount entirely.*

If the application adds first and then subtracts, *the query could add the amount twice.*

When an application uses ISO(UR) and runs concurrently with applications
 # that update variable-length records such that the update creates a
 # double-overflow record, the ISO(UR) application might miss rows that are
 # being updated.

If those situations can occur and are unacceptable, do not use UR isolation.

Restrictions: You cannot use UR isolation for the following types of statements:

- INSERT, UPDATE, and DELETE
- Any cursor defined with a FOR UPDATE clause

If you bind with ISOLATION(UR) and the statement does not specify WITH RR or WITH RS, DB2 uses CS isolation for these types of statements.

When can you use uncommitted read (UR)? You can probably use UR isolation in cases like the following ones:

- **When errors cannot occur.**

Example: A reference table, like a table of descriptions of parts by part number. It is rarely updated, and reading an uncommitted update is probably no more damaging than reading the table 5 seconds earlier. Go ahead and read it with ISOLATION(UR).

Example: The employee table of Spiffy Computer, our hypothetical user. For security reasons, updates can be made to the table only by members of a single department. And that department is also the only one that can query the entire table. It is easy to restrict queries to times when no updates are being made and then run with UR isolation.

- **When an error is acceptable.**

Example: Spiffy wants to do some statistical analysis on employee data. A typical question is, "What is the average salary by sex within education level?" Because reading an occasional uncommitted record cannot affect the averages much, UR isolation can be used.

- **When the data already contains inconsistent information.**

Example: Spiffy gets sales leads from various sources. The data is often inconsistent or wrong, and end users of the data are accustomed to dealing with that. Inconsistent access to a table of data on sales leads does not add to the problem.

Do not use uncommitted read (UR) in the following cases:

When the computations must balance

When the answer must be accurate

When you are not sure it can do no damage

ISOLATION (RS)

Allows the application to read the same pages or rows more than once without allowing qualifying rows to be updated or deleted by another process. It offers possibly greater concurrency than repeatable read, because although other applications cannot change rows that are returned to the original application, they can insert new rows or update rows that did not satisfy the original application's search condition. Only those rows or pages that satisfy the stage 1 predicate (and all rows or pages evaluated during stage 2 processing) are locked until the application commits. Figure 145 on page 416 illustrates this. In the example, the rows held by locks L2 and L4 satisfy the predicate.

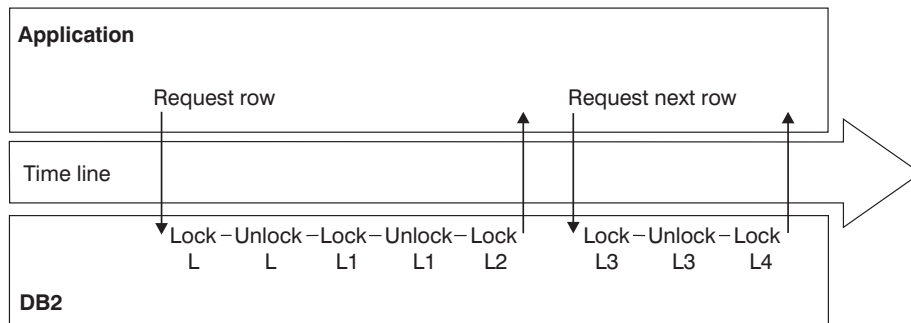


Figure 145. How an application using RS isolation acquires locks when no lock avoidance techniques are used. Locks L2 and L4 are held until the application commits. The other locks aren't held.

Applications using read stability can leave rows or pages locked for long periods, especially in a distributed environment.

If you do use read stability, plan for frequent commit points.

ISOLATION (RR)

Allows the application to read the same pages or rows more than once without allowing any UPDATE, INSERT, or DELETE by another process. All accessed rows or pages are locked, even if they do not satisfy the predicate.

Figure 146 shows that all locks are held until the application commits. In the following example, the rows held by locks L2 and L4 satisfy the predicate.

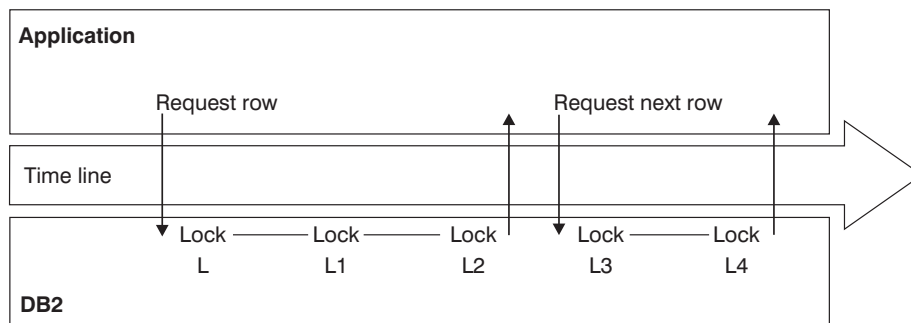


Figure 146. How an application using RR isolation acquires locks. All locks are held until the application commits.

Applications that use repeatable read can leave rows or pages locked for longer periods, especially in a distributed environment, and they can claim more logical partitions than similar applications using cursor stability.

They are also subject to being drained more often by utility operations.

Because so many locks can be taken, lock escalation might take place. Frequent commits release the locks and can help avoid lock escalation.

With repeatable read, lock promotion occurs for table space scan to prevent the insertion of rows that might qualify for the predicate. (If access is via index, DB2 locks the key range. If access is via table space scans, DB2 locks the table, partition, or table space.)

Restrictions on concurrent access: An application using UR isolation cannot run concurrently with a utility that drains all claim classes. Also, the application must acquire the following locks:

- A special *mass delete lock* acquired in S mode on the target table or table space. A “mass delete” is a DELETE statement without a WHERE clause; that operation must acquire the lock in X mode and thus cannot run concurrently.
- An IX lock on any table space used in the work file database. That lock prevents dropping the table space while the application is running.
- If LOB values are read, LOB locks and a lock on the LOB table space. If the LOB lock is not available because it is held by another application in an incompatible lock state, the UR reader skips the LOB and moves on to the next LOB that satisfies the query.

The CURRENTDATA option

The CURRENTDATA option has different effects, depending on if access is local or remote:

- For **local** access, the option tells whether the data upon which your cursor is positioned must remain identical to (or “current with”) the data in the local base table. For cursors positioned on data in a work file, the CURRENTDATA option has no effect. This effect only applies to read-only or *ambiguous* cursors in plans or packages bound with CS isolation.

A cursor is “ambiguous” if DB2 cannot tell whether it is used for update or read-only purposes. If the cursor appears to be used only for read-only, but dynamic SQL could modify data through the cursor, then the cursor is ambiguous. If you use CURRENTDATA to indicate an ambiguous cursor is read-only when it is actually targeted by dynamic SQL for modification, you’ll get an error. See “Problems with ambiguous cursors” on page 419 for more information about ambiguous cursors.

- For a request to a **remote** system, CURRENTDATA has an effect for ambiguous cursors using isolation levels RR, RS, or CS. For ambiguous cursors, it turns block fetching on or off. (Read-only cursors and UR isolation always use block fetch.) Turning on block fetch offers best performance, but it means the cursor is not current with the base table at the remote site.

Local access: Locally, **CURRENTDATA(YES)** means that the data upon which the cursor is positioned cannot change while the cursor is positioned on it. If the cursor is positioned on data in a local base table or index, then the data returned with the cursor is current with the contents of that table or index. If the cursor is positioned on data in a work file, the data returned with the cursor is current only with the contents of the work file; it is not necessarily current with the contents of the underlying table or index.

Figure 147 on page 418 shows locking with CURRENTDATA(YES).

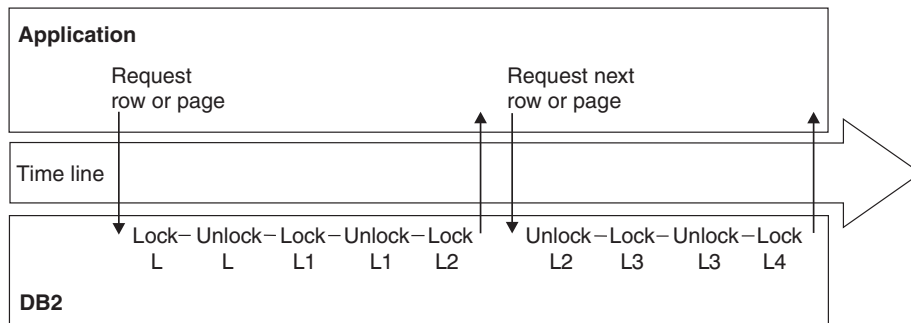


Figure 147. How an application using CS isolation with `CURRENTDATA(YES)` acquires locks. This figure shows access to the base table. The L2 and L4 locks are released after DB2 moves to the next row or page. When the application commits, the last lock is released.

As with work files, if a cursor uses query parallelism, data is not necessarily current with the contents of the table or index, regardless of whether a work file is used. Therefore, for work file access or for parallelism on read-only queries, the `CURRENTDATA` option has no effect.

If you are using parallelism but want to maintain currency with the data, you have the following options:

- Disable parallelism (Use `SET DEGREE = '1'` or bind with `DEGREE(1)`).
- Use isolation RR or RS (parallelism can still be used).
- Use the `LOCK TABLE` statement (parallelism can still be used).

For local access, `CURRENTDATA(NO)` is similar to `CURRENTDATA(YES)` except for the case where a cursor is accessing a base table rather than a result table in a work file. In those cases, although `CURRENTDATA(YES)` can guarantee that the cursor and the base table are current, `CURRENTDATA(NO)` makes no such guarantee.

Remote access: For access to a remote table or index, `CURRENTDATA(YES)` turns off block fetching for ambiguous cursors. The data returned with the cursor is current with the contents of the remote table or index for ambiguous cursors. See “Using block fetch in distributed applications” on page 458 for more information about the effect of `CURRENTDATA` on block fetch.

Lock avoidance: With `CURRENTDATA(NO)`, you have much greater opportunity for avoiding locks. DB2 can test to see if a row or page has committed data on it. If it has, DB2 does not have to obtain a lock on the data at all. Unlocked data is returned to the application, and the data can be changed while the cursor is positioned on the row. (For `SELECT` statements in which no cursor is used, such as those that return a single row, a lock is not held on the row unless you specify `WITH RS` or `WITH RR` on the statement.)

To take the best advantage of this method of avoiding locks, make sure all applications that are accessing data concurrently issue `COMMITs` frequently.

Figure 148 on page 419 shows how DB2 can avoid taking locks and Table 54 on page 419 summarizes the factors that influence lock avoidance.

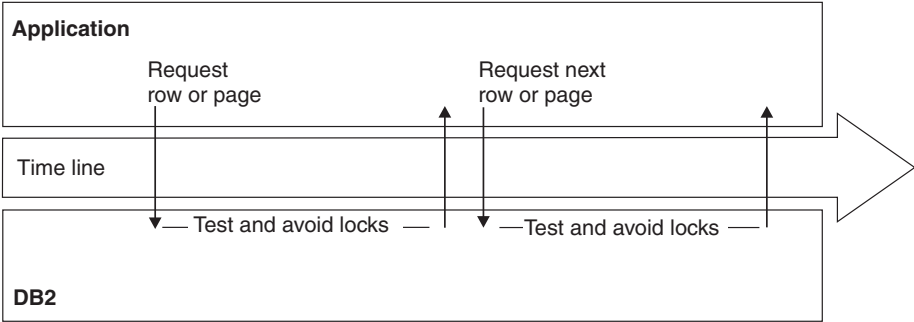


Figure 148. Best case of avoiding locks using CS isolation with *CURRENTDATA(NO)*. This figure shows access to the base table. If DB2 must take a lock, then locks are released when DB2 moves to the next row or page, or when the application commits (the same as *CURRENTDATA(YES)*).

Table 54. Lock avoidance factors. “Returned data” means data that satisfies the predicate. “Rejected data” is that which does not satisfy the predicate.

Isolation	CURRENTDATA	Cursor type	Avoid locks on returned data?	Avoid locks on rejected data?
UR	N/A	Read-only	N/A	N/A
# CS	YES	Read-only	No	Yes ¹
		Updatable		
		Ambiguous		
	NO	Read-only	Yes	
		Updatable	No	
		Ambiguous	Yes	
# RS	N/A	Read-only	No	Yes ^{1, 2}
		Updatable		
		Ambiguous		
# RR	N/A	Read-only	No	No
		Updatable		
		Ambiguous		

Note:

1. Locks are avoided when the row is disqualified after stage 1 processing
2. When using ISO(RS) and multi-row fetch, DB2 releases locks that were acquired on Stage 1 qualified rows, but which subsequently failed to qualify for stage 2 predicates at the next fetch of the cursor.

Problems with ambiguous cursors: As shown in Table 54, ambiguous cursors can sometimes prevent DB2 from using lock avoidance techniques. However, misuse of an ambiguous cursor can cause your program to receive a -510 SQLCODE:

- The plan or package is bound with *CURRENTDATA(NO)*
- An OPEN CURSOR statement is performed *before* a dynamic DELETE WHERE CURRENT OF statement against that cursor is prepared
- One of the following conditions is true for the open cursor:
 - Lock avoidance is successfully used on that statement.
 - Query parallelism is used.

- The cursor is distributed, and block fetching is used.

In all cases, it is a good programming technique to eliminate the ambiguity by declaring the cursor with either the FOR FETCH ONLY or the FOR UPDATE clause.

When plan and package options differ

A plan bound with one set of options can include packages in its package list that were bound with different sets of options. In general, statements in a DBRM bound as a package use the options that the package was bound with, and statements in DBRMs bound to a plan use the options that the plan was bound with.

For example, the plan value for CURRENTDATA has no effect on the packages executing under that plan. If you do not specify a CURRENTDATA option explicitly when you bind a package, the default is CURRENTDATA(YES).

The rules are slightly different for the bind options RELEASE and ISOLATION. The values of those two options are set when the lock on the resource is acquired and usually stay in effect until the lock is released. But a conflict can occur if a statement that is bound with one pair of values requests a lock on a resource that is already locked by a statement that is bound with a different pair of values. DB2 resolves the conflict by resetting each option with the available value that causes the lock to be held for the greatest duration.

If the conflict is between RELEASE(COMMIT) and RELEASE(DEALLOCATE), then the value used is RELEASE(DEALLOCATE).

Table 55 shows how conflicts between isolation levels are resolved. The first column is the existing isolation level, and the remaining columns show what happens when another isolation level is requested by a new application process.

Table 55. Resolving isolation conflicts

	UR	CS	RS	RR
UR	n/a	CS	RS	RR
CS	CS	n/a	RS	RR
RS	RS	RS	n/a	RR
RR	RR	RR	RR	n/a

The effect of WITH HOLD for a cursor

For a cursor defined as WITH HOLD, the cursor position is maintained past a commit point. Hence, locks and claims needed to maintain that position are not released immediately, even if they were acquired with ISOLATION(CS) or RELEASE(COMMIT).

For locks and claims that are needed for cursor position, the following exceptions exist for special cases:

Page and row locks: If your installation specifies NO on the RELEASE LOCKS field of installation panel DSNTIP4, as described in *DB2 Administration Guide*, a page or row lock is held past the commit point. This page or row lock is not necessary for cursor position, but the NO option is provided for compatibility that might rely on this lock. However, an X or U lock is demoted to an S lock at that time. (Because changes have been committed, exclusive control is no longer

needed.) After the commit point, the lock is released at the next commit point, provided that no cursor is still positioned on that page or row.

#

If your installation specifies YES on the RELEASE LOCKS field on installation panel DSN TIP4, data page or row locks are not held past commit.

Table, table space, and DBD locks: All necessary locks are held past the commit point. After that, they are released according to the RELEASE option under which they were acquired: for COMMIT, at the next commit point after the cursor is closed; for DEALLOCATE, when the application is deallocated.

Claims: All claims, for any claim class, are held past the commit point. They are released at the next commit point after all held cursors have moved off the object or have been closed.

Isolation overriding with SQL statements

Function of the WITH clause: You can override the isolation level with which a plan or package is bound by the WITH clause on certain SQL statements.

Example: This statement:

```
SELECT MAX(BONUS), MIN(BONUS), AVG(BONUS)
      INTO :MAX, :MIN, :AVG
      FROM DSN8810.EMP
      WITH UR;
```

finds the maximum, minimum, and average bonus in the sample employee table. The statement is executed with uncommitted read isolation, regardless of the value of ISOLATION with which the plan or package containing the statement is bound.

Rules for the WITH clause: The WITH clause:

- Can be used on these statements:
 - Select-statement
 - SELECT INTO
 - Searched delete
 - INSERT from fullselect
 - Searched update
- Cannot be used on subqueries.
- Can specify the isolation levels that specifically apply to its statement. (For example, because WITH UR applies only to read-only operations, you cannot use it on an INSERT statement.)
- Overrides the isolation level for the plan or package only for the statement in which it appears.

|
|
|
|
|

USE AND KEEP ... LOCKS options of the WITH clause: If you use the WITH RR or WITH RS clause, you can use the USE AND KEEP EXCLUSIVE LOCKS, USE AND KEEP UPDATE LOCKS, USE AND KEEP SHARE LOCKS options in SELECT and SELECT INTO statements.

|
|
|

Example: To use these options, specify them as shown in the following example:

```
SELECT ...
      WITH RS USE KEEP UPDATE LOCKS;
```

|
|
|

By using one of these options, you tell DB2 to acquire and hold a specific mode of lock on all the qualified pages or rows. Table 56 on page 422 shows which mode of lock is held on rows or pages when you specify the SELECT using the WITH RS or

WITH RR isolation clause.

Table 56. Which mode of lock is held on rows or pages when you specify the SELECT using the WITH RS or WITH RR isolation clause

Option Value	Lock Mode
USE AND KEEP EXCLUSIVE LOCKS	X
USE AND KEEP UPDATE LOCKS	U
USE AND KEEP SHARE LOCKS	S

With read stability (RS) isolation, a row or page that is rejected during stage 2 processing might still have a lock held on it, even though it is not returned to the application.

With repeatable read (RR) isolation, DB2 acquires locks on all pages or rows that fall within the range of the selection expression.

All locks are held until the application commits. Although this option can reduce concurrency, it can prevent some types of deadlocks and can better serialize access to data.

The LOCK TABLE statement

For information about using LOCK TABLE on an auxiliary table, see “The LOCK TABLE statement for LOBs” on page 428.

The purpose of LOCK TABLE

Use the LOCK TABLE statement to override DB2’s rules for choosing initial lock attributes. Two examples are:

```
LOCK TABLE table-name IN SHARE MODE;  
LOCK TABLE table-name PART n IN EXCLUSIVE MODE;
```

Executing the statement requests a lock immediately, unless a suitable lock exists already. The bind option RELEASE determines when locks acquired by LOCK TABLE or LOCK TABLE with the PART option are released.

You can use LOCK TABLE on any table, including auxiliary tables of LOB table spaces. See “The LOCK TABLE statement for LOBs” on page 428 for information about locking auxiliary tables.

LOCK TABLE has no effect on locks acquired at a remote server.

The effect of LOCK TABLE

Table 57 shows the modes of locks acquired in segmented and nonsegmented table spaces for the SHARE and EXCLUSIVE modes of LOCK TABLE. Auxiliary tables of LOB table spaces are considered nonsegmented table spaces and have the same locking behavior.

Table 57. Modes of locks acquired by LOCK TABLE. LOCK TABLE on partitions behave the same as nonsegmented table spaces.

LOCK TABLE IN	Nonsegmented Table Space	Segmented Table Space	
		Table	Table Space
EXCLUSIVE MODE	X	X	IX
SHARE MODE	S or SIX	S or SIX	IS

Table 57. Modes of locks acquired by LOCK TABLE (continued). LOCK TABLE on partitions behave the same as nonsegmented table spaces.

LOCK TABLE IN	Nonsegmented Table Space	Segmented Table Space	
		Table	Table Space
Note: The SIX lock is acquired if the process already holds an IX lock. SHARE MODE has no effect if the process already has a lock of mode SIX, U, or X.			

Recommendations for using LOCK TABLE

Use LOCK TABLE to prevent other application processes from changing any row in a table or partition that your process is accessing. For example, suppose that you access several tables. You can tolerate concurrent updates on all the tables except one; for that one, you need RR or RS isolation. There are several ways to handle the situation:

- Bind the application plan with RR or RS isolation. But that affects all the tables you access and might reduce concurrency.
- Design the application to use packages and access the exceptional table in only a few packages. Bind those packages with RR or RS isolation and the plan with CS isolation. Only the tables accessed within those packages are accessed with RR or RS isolation.
- Add the clause WITH RR or WITH RS to statements that must be executed with RR or RS isolation. Statements that do not use WITH are executed as specified by the bind option ISOLATION.
- Bind the application plan with CS isolation *and* execute LOCK TABLE for the exceptional table. (If there are other tables in the same table space, see the caution that follows.) The LOCK TABLE statement locks out changes by any other process, giving the exceptional table a degree of isolation even more thorough than repeatable read. All tables in other table spaces are shared for concurrent update.

Caution when using LOCK TABLE with simple table spaces: The statement locks all tables in a simple table space, even though you name only one table. No other process can update the table space for the duration of the lock. If the lock is in exclusive mode, no other process can read the table space, unless that process is running with UR isolation.

Additional examples of LOCK TABLE: You might want to lock a table or partition that is normally shared for any of the following reasons:

Taking a “snapshot”

If you want to access an entire table throughout a unit of work as it was at a particular moment, you must lock out concurrent changes. If other processes can access the table, use LOCK TABLE IN SHARE MODE. (RR isolation is not enough; it locks out changes only from rows or pages you have already accessed.)

Avoiding overhead

If you want to update a large part of a table, it can be more efficient to prevent concurrent access than to lock each page as it is updated and unlock it when it is committed. Use LOCK TABLE IN EXCLUSIVE MODE.

Preventing timeouts

Your application has a high priority and must not risk timeouts from contention with other application processes. Depending on

whether your application updates or not, use either LOCK IN EXCLUSIVE MODE or LOCK TABLE IN SHARE MODE.

Access paths

The access path used can affect the mode, size, and even the object of a lock. For example, an UPDATE statement using a table space scan might need an X lock on the entire table space. If rows to be updated are located through an index, the same statement might need only an IX lock on the table space and X locks on individual pages or rows.

If you use the EXPLAIN statement to investigate the access path chosen for an SQL statement, then check the lock mode in column TSLOCKMODE of the resulting PLAN_TABLE. If the table resides in a nonsegmented table space, or is defined with LOCKSIZE TABLESPACE, the mode shown is that of the table space lock. Otherwise, the mode is that of the table lock.

Important points about DB2 locks:

- You usually do not have to lock data explicitly in your program.
- DB2 ensures that your program does not retrieve uncommitted data unless you specifically allow that.
- Any page or row where your program updates, inserts, or deletes stays locked at least until the end of a unit of work, regardless of the isolation level. No other process can access the object in any way until then, unless you specifically allow that access to that process.
- Commit often for concurrency. Determine points in your program where changed data is consistent. At those points, issue:

TSO, Batch, and CAF An SQL COMMIT statement

IMS A CHKP or SYNC call, or (for single-mode transactions) a GU call to the I/O PCB

CICS A SYNCPOINT command.

- Bind with ACQUIRE(USE) to improve concurrency.
- Set ISOLATION (usually RR, RS, or CS) when you bind the plan or package.
 - With RR (repeatable read), all accessed pages or rows are locked until the next commit point. (See “Recommendations for database design” on page 398 for information about cursor position locks for cursors defined WITH HOLD.)
 - With RS (read stability), all qualifying pages or rows are locked until the next commit point. (See “Recommendations for application design” on page 399 for information about cursor position locks for cursors defined WITH HOLD.)
 - With CS (cursor stability), only the pages or rows currently accessed can be locked, and those locks might be avoided. (You can access one page or row for each open cursor.)

- You can also set isolation for specific SQL statements, using WITH.
- A deadlock can occur if two processes each hold a resource that the other needs. One process is chosen as “victim”, its unit of work is rolled back, and an SQL error code is issued.
- You can lock an entire nonsegmented table space, or an entire table in a segmented table space, by the LOCK TABLE statement:
 - To let other users retrieve, but not update, delete, or insert, issue:
LOCK TABLE *table-name* IN SHARE MODE
 - To prevent other users from accessing rows in any way, except by using UR isolation, issue:
LOCK TABLE *table-name* IN EXCLUSIVE MODE

LOB locks

The locking activity for LOBs is described separately from transaction locks because the purpose of LOB locks is different than that of regular transaction locks.

A lock that is taken on a LOB value in a LOB table space is called a *LOB lock*.

In this section, the following topics are described:

- “Relationship between transaction locks and LOB locks”
- “Hierarchy of LOB locks” on page 426
- “LOB and LOB table space lock modes” on page 426
- “LOB lock and LOB table space lock duration” on page 427
- “Instances when LOB table space locks are not taken” on page 428
- “The LOCK TABLE statement for LOBs” on page 428

Relationship between transaction locks and LOB locks

As described in “Introduction to LOBs” on page 297, LOB column values are stored in a different table space, a LOB table space, from the values in the base table. An application that reads or updates a row in a table that contains LOB columns obtains its normal transaction locks on the base table. The locks on the base table also control concurrency for the LOB table space. When locks are not acquired on the base table, such as for ISO(UR), DB2 maintains data consistency by using locks on the LOB table space. Even when locks are acquired on the base table, DB2 still obtains locks on the LOB table space. When a conditional LOB lock cannot be acquired for some SQL statements with UR isolation, DB2 might return no rows and issue an SQL return code +100.

DB2 also obtains locks on the LOB table space and the LOB values stored in that LOB table space, but those locks have the following primary purposes:

- To determine whether space from a deleted LOB can be reused by an inserted or updated LOB
Storage for a deleted LOB is not reused until no more readers (including held locators) are on the LOB and the delete operation has been committed.
- To prevent deallocating space for a LOB that is currently being read
A LOB can be deleted from one application’s point-of-view while a reader from another application is reading the LOB. The reader continues reading the LOB because all readers, including those readers that are using uncommitted read isolation, acquire S-locks on LOBs to prevent the storage for the LOB they are reading from being deallocated. That lock is held until commit. A held LOB locator or a held cursor cause the LOB lock and LOB table space lock to be held past commit.

In summary, the main purpose of LOB locks is for managing the space used by LOBs and to ensure that LOB readers do not read partially updated LOBs. Applications need to free held locators so that the space can be reused.

Table 58 shows the relationship between the action that is occurring on the LOB value and the associated LOB table space and LOB locks that are acquired.

Table 58. Locks that are acquired for operations on LOBs. This table does not account for gross locks that can be taken because of LOCKSIZE TABLESPACE, the LOCK TABLE statement, or lock escalation.

Action on LOB value	LOB table space lock	LOB lock	Comment
Read (including UR)	IS	S	Prevents storage from being reused while the LOB is being read or while locators are referencing the LOB
Insert	IX	X	Prevents other processes from seeing a partial LOB
Delete	IS	S	To hold space in case the delete is rolled back. (The X is on the base table row or page.) Storage is not reusable until the delete is committed and no other readers of the LOB exist.
Update	IS->IX	Two LOB locks: an S-lock for the delete and an X-lock for the insert.	Operation is a delete followed by an insert.
Update the LOB to null or zero-length	IS	S	No insert, just a delete.
Update a null or zero-length LOB to a value	IX	X	No delete, just an insert.

ISOLATION(UR) or ISOLATION(CS): When an application is reading rows using uncommitted read or lock avoidance, no page or row locks are taken on the base table. Therefore, these readers must take an S LOB lock to ensure that they are not reading a partial LOB or a LOB value that is inconsistent with the base row.

Hierarchy of LOB locks

Just as page locks (or row locks) and table space locks have a hierarchical relationship, LOB locks and locks on LOB table spaces have a hierarchical relationship. (See Figure 142 on page 403 for a picture of the hierarchical relationship.) If the LOB table space is locked with a gross lock, then LOB locks are not acquired. In a data sharing environment, the lock on the LOB table space is used to determine whether the lock on the LOB must be propagated beyond the local IRLM.

LOB and LOB table space lock modes

This section describes the modes of LOB locks and LOB table space locks.

Modes of LOB locks

The following LOB lock modes are possible:

S (SHARE)

The lock owner and any concurrent processes can read, update, or delete the locked LOB. Concurrent processes can acquire an S lock on the LOB. The purpose of the S lock is to reserve the space used by the LOB.

X (EXCLUSIVE)

The lock owner can read or change the locked LOB. Concurrent processes cannot access the LOB.

Modes of LOB table space locks

The following locks modes are possible on the LOB table space:

IS (INTENT SHARE)

The lock owner can update LOBs to null or zero-length, or read or delete LOBs in the LOB table space. Concurrent processes can both read and change LOBs in the same table space. The lock owner acquires a LOB lock on any data that it reads or deletes.

IX (INTENT EXCLUSIVE)

The lock owner and concurrent processes can read and change data in the LOB table space. The lock owner acquires a LOB lock on any data it accesses.

S (SHARE)

The lock owner and any concurrent processes can read and delete LOBs in the LOB table space. The lock owner does not need LOB locks.

SIX (SHARE with INTENT EXCLUSIVE)

The lock owner can read and change data in the LOB table space. If the lock owner is inserting (INSERT or UPDATE), the lock owner obtains a LOB lock. Concurrent processes can read or delete data in the LOB table space (or update to a null or zero-length LOB).

X (EXCLUSIVE)

The lock owner can read or change LOBs in the LOB table space. The lock owner does not need LOB locks. Concurrent processes cannot access the data.

LOB lock and LOB table space lock duration

This section describes the duration of LOB locks and LOB table space locks.

The duration of LOB locks

Locks on LOBs are taken when they are needed and are usually released at commit. However, if that LOB value is assigned to a LOB locator, the S lock remains until the application commits.

If the application uses HOLD LOCATOR, the LOB lock is not freed until the first commit operation after a FREE LOCATOR statement is issued, or until the thread is deallocated.

If a cursor is defined WITH HOLD, LOB locks are held through commit operations.

Because LOB locks are held until commit and because locks are put on each LOB column in both a source table and a target table, it is possible that a statement such as an INSERT with a fullselect that involves LOB columns can accumulate many

more locks than a similar statement that does not involve LOB columns. To prevent system problems caused by too many locks, you can:

- Ensure that you have lock escalation enabled for the LOB table spaces that are involved in the INSERT. In other words, make sure that LOCKMAX is non-zero for those LOB table spaces.
- Alter the LOB table space to change the LOCKSIZE to TABLESPACE before executing the INSERT with fullselect.
- Increase the LOCKMAX value on the table spaces involved and ensure that the user lock limit is sufficient.
- Use LOCK TABLE statements to lock the LOB table spaces. (Locking the auxiliary table that is contained in the LOB table space locks the LOB table space.)

The duration of LOB table space locks

Locks on LOB table spaces are acquired when they are needed; that is, the ACQUIRE option of BIND has no effect on when the table space lock on the LOB table space is taken. When the table space lock is released is determined by a combination of factors:

- The RELEASE option of bind
- Whether the SQL statement is static or dynamic
- Whether there are held cursors or held locators

When the release option is COMMIT, the lock is released at the next commit point, unless there are held cursors or held locators. If the release option is DEALLOCATE, the lock is released when the object is deallocated (the application ends). The BIND option has no effect on dynamic SQL statements, which always use RELEASE(COMMIT), unless you use dynamic statement caching.

Instances when LOB table space locks are not taken

A lock might not be acquired on a LOB table space at all. For example, if a row is deleted from a table and the value of the LOB column is null, the LOB table space associated with that LOB column is not locked. DB2 does not access the LOB table space if the application:

- Selects a LOB that is null or zero length
- Deletes a row where the LOB is null or zero length
- Inserts a null or zero length LOB
- Updates a null or zero-length LOB to null or zero-length

The LOCK TABLE statement for LOBs

“The LOCK TABLE statement” on page 422 describes how and why you might use a LOCK TABLE statement on a table. The reasons for using LOCK TABLE on an auxiliary table are somewhat different than that for regular tables.

- You can use LOCK TABLE to control the number of locks acquired on the auxiliary table.
- You can use LOCK TABLE IN SHARE MODE to prevent other applications from inserting LOBs.

With auxiliary tables, LOCK TABLE IN SHARE MODE does not prevent any changes to the auxiliary table. The statement does prevent LOBs from being inserted into the auxiliary table, but it does not prevent deletes. Updates are generally restricted also, except where the LOB is updated to a null value or a zero-length string.

- You can use `LOCK TABLE IN EXCLUSIVE MODE` to prevent other applications from accessing LOBs.
With auxiliary tables, `LOCK TABLE IN EXCLUSIVE MODE` also prevents access from uncommitted readers.
- Either statement eliminates the need for lower-level LOB locks.

Chapter 19. Planning for recovery

During recovery, when a DB2 database is restoring to its most recent consistent state, you must back out any uncommitted changes to data that occurred before the program abend or system failure. You must do this without interfering with other system activities.

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)`.

A *unit of work* is a logically distinct procedure containing 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 `DSN8810.EMP` exchange offices. You need to exchange their office phone numbers in the `PHONENO` column. You would 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.

When a unit of work completes, all locks that were implicitly acquired by that unit of work are released, allowing a new unit of work to begin.

The amount of processing time that is used by a unit of work in your program affects the length of time DB2 prevents other users from accessing that locked data. When several programs try to use the same data concurrently, each program's unit of work should be as short as possible to minimize the interference between the programs.

This chapter describes the way a unit of work functions in various environments. It contains the following sections:

- "Unit of work in TSO batch and online" on page 432
- "Unit of work in CICS" on page 432
- "Unit of work in IMS online programs" on page 433
- "Unit of work in DL/I and IMS batch programs" on page 438
- "Using savepoints to undo selected changes within a unit of work" on page 439

For more information about unit of work, see Chapter 1 of *DB2 SQL Reference* or Part 4 (Volume 1) of *DB2 Administration Guide*.

Unit of work in TSO batch and online

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 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.
- The program terminates and returns to the DSN command processor, which returns to the TSO Terminal Monitor Program (TMP).

A *commit point* occurs when you issue a COMMIT statement or your program terminates normally. You should issue a COMMIT statement only when you are sure the data is in a consistent state. For example, a bank transaction might transfer funds from account A to account B. The transaction first subtracts the amount of the transfer from account A, and then adds the amount to account B. Both events, taken together, are a unit of work. When both events complete (and not before), the data in the two accounts is consistent. The program can then issue a COMMIT statement. A ROLLBACK statement causes any data changes that were made since the last commit point to be backed out.

Before you can connect to another DBMS, you must issue a COMMIT statement. If the system fails at this point, DB2 cannot know that 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.

| You can provide an abend exit routine in your program. 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.

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

| 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.

Unit of work in CICS

In CICS, 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*. Generally, a unit of work is a *sequence* of actions that must complete before any of the *individual* actions in the sequence can complete. For example, decrementing an inventory file and incrementing a reorder file by the same quantity can constitute a unit of work: *both* steps must complete before either step is complete. (If one action occurs and not the other, the database loses its integrity, or consistency.)

A unit of work is marked as complete by a *commit* or *synchronization (sync)* point, which is defined:

- Implicitly at the end of a transaction, signalled 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 can occur when the batch DL/I program is sharing a database with CICS applications through the database sharing facility.

Consider the inventory example, in which the quantity of items sold is subtracted from the inventory file and then added to the 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 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.

If the system fails, DB2 backs out uncommitted changes to data. Changed data returns to its original condition without interfering with other system activities. Sometimes, DB2 data does not return to a consistent state immediately. DB2 does not process *indoubt data* (data that is neither uncommitted nor committed) until the CICS attachment facility is also restarted. To ensure that DB2 and CICS are synchronized, restart both DB2 and the CICS attachment facility.

Unit of work in IMS online programs

In IMS, a *unit of work* starts:

- 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:

- 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.

A *commit point* can occur in a program as the result of any one of the following four 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*. The SYNC call is a Fast Path system service call to request commit-point processing. You can use a SYNC call only in a nonmessage-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. IMS considers a new message the start of a new unit of work in the program. Commit points occur given the following conditions:
 - If you specify *single-mode*, a commit point in DB2 occurs each time the program issues a call to retrieve a new message. Specifying single-mode can simplify recovery; you can restart the program from the most recent call for a new message if the program abends. When IMS restarts the program, the program starts by processing the next message.
 - If you specify *multiple-mode*, a commit point occurs when the program issues a checkpoint call or when it terminates normally. Those 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.

If you do not define the transaction as single- or multiple-mode on the TRANSACT statement of the APPLCTN macro for the program, retrieving a new message does not signal a commit point. For more information about the APPLCTN macro, see *IMS Install Volume 2: System Definition and Tailoring*.

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.

At the time of a commit point:

- IMS and DB2 can release locks that the program has held since the last commit point. That makes the data available to other application programs and users. (However, when you define a cursor as WITH HOLD in a BMP program, DB2 holds those locks until the cursor closes or the program ends.)
- DB2 closes any open cursors that the program has been using. Your program must issue CLOSE CURSOR statements **before** a checkpoint call or a GU to the message queue, **not after**.
- IMS and DB2 make the program's changes to the database permanent.

If the program abends before reaching the commit point:

- 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, 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, people at terminals and other application programs receive information from the terminating application program.

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

If the system fails, DB2 backs out uncommitted changes to data. Changed data returns to its original state without interfering with other system activities. Sometimes DB2 data does not return to a consistent state immediately. DB2 does not process data in an indoubt state until you restart IMS. To ensure that DB2 and IMS are synchronized, you must restart both DB2 and IMS.

Planning ahead for program recovery: Checkpoint and restart

Both IMS and DB2 handle recovery in an IMS application program that accesses DB2 data. IMS coordinates the process and DB2 participates by handling recovery for DB2 data.

Two calls that are available to IMS programs to simplify program recovery are the symbolic checkpoint call and the restart call.

What symbolic checkpoint does

Symbolic checkpoint calls indicate to IMS that the program has reached a sync point. Such calls also establish places in the program from which you can restart the program.

A CHKP call causes IMS to:

- Inform DB2 that the changes 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 containing 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, which resets special registers as follows:
 - CURRENT PACKAGESET to blanks
 - CURRENT SERVER to blanks
 - CURRENT SQLID to blanks
 - CURRENT DEGREE to 1

Your program must restore these special registers if their values are needed after the checkpoint.

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.

What restart does

The restart call (XRST), which you must use with symbolic checkpoints, provides a method for restarting a program after an abend. It 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.

When are checkpoints important?

Issuing checkpoint calls releases locked resources. 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:

- 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

Checkpoints in MPPs and transaction-oriented BMPs

Single-mode programs: 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.

Multiple-mode programs: 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 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.

The following factors affect the use of checkpoint calls in multiple-mode programs:

- 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
- 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.

Checkpoints in batch-oriented BMPs

Issuing checkpoints in a batch-oriented BMP is important for several reasons:

- To commit changes to the database
- To establish places from which the program can be restarted
- To release locked DB2 and IMS data that IMS has enqueued for the program

Checkpoints also close all open cursors, which means that you must reopen the cursors you want and re-establish positioning.

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

- 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 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.
- If IMS is using program isolation enqueueing, the space 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. If a BMP enqueues too many segments, the amount of storage needed for the enqueued segments can exceed the amount of available storage. If that happens, IMS terminates the program abnormally with an abend code of U0775. You then need to increase the program's checkpoint frequency before rerunning the program. The amount of storage available is specified during IMS system definition. For more information, see *IMS Install Volume 2: System Definition and Tailoring*.

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 do this is to put such information in a data area that is included in the DL/I CHKP call. Undesirable performance implications can be associated with re-establishing position within a DB2 database as a result of the commit processing that takes place because of a DL/I CHKP call. 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.

Another limitation of processing DB2 databases in a BMP program is that you can restart the program only from the latest checkpoint and not from any checkpoint, as in IMS.

Specifying checkpoint frequency

You must specify checkpoint frequency in your program in a way that makes changing it easy in case the frequency you initially specify is not right. Some ways to do this are to use a counter in your program to keep a record of:

- Elapsed time; issue a checkpoint call after a certain time interval.
- The number of root segments your program accesses; issue a checkpoint call after a certain number of root segments.
- The number of updates your program performs; issue a checkpoint call after a certain number of updates.

Unit of work in DL/I and IMS batch programs

This section describes how to coordinate commit and rollback operations for DL/I batch, and how to restart and recover data in IMS batch.

Commit and rollback coordination

DB2 coordinates commit and rollback for DL/I batch, with the following considerations:

- DB2 and DL/I changes are committed as the result of IMS CHKP calls. However, you lose the application program database positioning in DL/I. In addition, the program database positioning in DB2 can be affected as follows:
 - 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.
- DB2 automatically backs out changes whenever the application program abends. To back out DL/I changes, you must use the DL/I batch backout utility.
- You cannot use SQL statements COMMIT and ROLLBACK in the DB2 DL/I batch support environment, because IMS coordinates the unit of work. Issuing COMMIT causes SQLCODE -925 (SQLSTATE '2D521'); issuing ROLLBACK causes SQLCODE -926 (SQLSTATE '2D521').
- 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.
- You can use IMS rollback calls, ROLL and ROLB, to back out DB2 and DL/I changes to the last commit point. When you issue a ROLL call, DL/I terminates your program with an abend. When you issue a ROLB call, DL/I returns control to your program after the call.

How ROLL and ROLB affect DL/I changes in a batch environment depends on the IMS system log and the back out options that are specified, as the following summary indicates:

- A ROLL call with tape logging (with any BKO value), or disk logging and BKO=NO specified. DL/I does not back out updates, and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
- A ROLB call with tape logging (with any BKO value), or disk logging and BKO=NO specified. 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.

- A ROLL call with disk logging and BKO=YES specified. DL/I backs out updates, and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
- A ROLB call with disk logging and BKO=YES specified. DL/I backs out databases, and control is passed back to the application program. DB2 backs out updates to the previous checkpoint.

Using ROLL

Issuing a ROLL call causes IMS to terminate the program with a user abend code U0778. This terminates the program without a storage dump.

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

Using ROLB

The advantage of using ROLB is that IMS returns control to the program after executing ROLB, allowing the program to continue processing. The options for ROLB are:

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

In batch programs

If your IMS system log is on direct access storage, and if the run option BKO is Y to specify dynamic back out, you can use the ROLB call in a batch program. The ROLB call backs out the database updates made since the last commit point and returns control to your program. 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. You must, however, have an I/O PCB for your program. Specify CMPAT=YES on the CMPAT keyword in the PSBGEN statement for your program's PSB. For more information about using the CMPAT keyword, see *IMS Utilities Reference: System*.

Restart and recovery in IMS batch

In an online IMS system, recovery and restart are part of the IMS system. For a batch region, your location's operational procedures control recovery and restart. For more information, see *IMS Application Programming: Design Guide*.

Using savepoints to undo selected changes within a unit of work

Savepoints let you undo selected changes within a transaction. Your application can set any number of savepoints using SQL SAVEPOINT statements, and then use SQL ROLLBACK TO SAVEPOINT statements to indicate which changes within the unit of work to undo. When the application no longer uses a savepoint, it can delete that savepoint using the SQL RELEASE SAVEPOINT statement.

You can write a ROLLBACK TO SAVEPOINT statement with or without a savepoint name. If you do not specify a savepoint name, DB2 rolls back work to the most recently created savepoint.

Example: 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;
```


When savepoints are active, you cannot access remote sites using three-part names or aliases for three-part names. You can, however, use DRDA access with explicit CONNECT statements when savepoints are active. 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.

Example: Setting savepoints during distributed processing: Suppose that an application performs these 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.

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.

Example: Setting a savepoint multiple times: Suppose that the following actions take place 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 ROLLBACK TO SAVEPOINT S.

When DB2 executes ROLLBACK to SAVEPOINT S, 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.

If you do not want a savepoint to have different values within a unit of work, you can use the UNIQUE option in the SAVEPOINT statement. If an application executes a SAVEPOINT statement for a savepoint that was previously defined as unique, an SQL error occurs.

Savepoints are automatically released at the end of a unit of work. However, if you no longer need a savepoint before the end of a transaction, you should execute the SQL RELEASE SAVEPOINT statement. Releasing savepoints is essential if you need to use three-part names to access remote locations.

Restrictions on using savepoints: You cannot use savepoints in global transactions, triggers, or user-defined functions, or in stored procedures, user-defined functions, or triggers that are nested within triggers or user-defined functions.

For more information about the SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT statements, see Chapter 5 of *DB2 SQL Reference*.

Chapter 20. Planning to access distributed data

Distributed data is data that resides on some 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*.

This chapter assumes that you are requesting services from a remote DBMS. That DBMS is a server in that situation, 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 remote in the physical sense or it can be another subsystem of the same operating system that your local DBMS runs under. This chapter assumes that your local DBMS is an instance of DB2 UDB for z/OS. A remote server might be an instance of DB2 UDB for z/OS also, or it might be an instance of one of many other products.

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. For more information about location names or the communications database, see Part 3 of *DB2 Installation Guide*.)

Authorization to connect to a remote server and to use resources there must be granted at the server to the appropriate authorization ID. When the server is DB2 UDB for z/OS, see Part 3 (Volume 1) of *DB2 Administration Guide* for information about authorization. For other servers, see the documentation for the appropriate product.

For distributed applications that contain SQL statements that run at the requester, you must include a DBRM that is either bound directly to a plan or to a package that is included in the plan's package list at the requester.

Additionally, you must include a package at the server for any SQL statements that run at the server. See Appendix C of *DB2 SQL Reference* for information about which SQL statements are processed at the requester.

This chapter contains the following sections:

- "Planning for DRDA and DB2 private protocol access"
- "Coding methods for distributed data" on page 448
- "Coordinating updates to two or more data sources" on page 452
- "Maximizing performance for distributed data" on page 454
- "Working with distributed data" on page 465

Planning for DRDA and DB2 private protocol access

This section considers two methods for accessing distributed data: DRDA access and DB2 private protocol access. In almost all cases, you should use DRDA access instead of DB2 private protocol access.

Advantages of DRDA access

DRDA access has the following advantages over DB2 private protocol access:

- **Integration:** DRDA access is available to **all** DBMSs that implement Distributed Relational Database Architecture™ (DRDA). Those include supported releases of DB2 UDB for z/OS, other members of the DB2 UDB family of IBM products, and many products of other companies.

DB2 private protocol access is available only to supported releases of DB2 UDB for z/OS.

- **SQL compatibility:** DRDA access allows **any** statement that the server can execute.

DB2 private protocol access supports only data manipulation statements:

INSERT, UPDATE, DELETE, SELECT, OPEN, FETCH, and CLOSE. In addition, you cannot use any syntax of an SQL statement that was introduced after DB2 Version 7. You cannot invoke user-defined functions and stored procedures or use LOBs or distinct types in applications that use DB2 private protocol access.

- **Reduced network load:** DRDA access uses a more compact format for sending data over the network, which improves the performance on slow network links.
- **Reduced bind processing:** A DBRM for statements executed by DRDA access is bound to a package at the server only once. Those statements can include PREPARE and EXECUTE, so your application can accept dynamic statements that are to be executed at the server. Binding the package is an extra step in program preparation.

Queries that are sent by DB2 private protocol access are bound at the server whenever they are first executed in a unit of work. Repeated binds can reduce the performance of a query that is executed often.

- **Stored procedures:** You can use stored procedures with DRDA access. While a stored procedure is running, it requires no message traffic over the network; this reduces the biggest obstacle to high performance for distributed data.
- **Scrollable cursors:** You can use scrollable cursors if you use DRDA access.
- **Savepoints:** You can set savepoints, only if you use DRDA access with explicit CONNECT statements. If you set a savepoint and then execute an SQL statement with a three-part name, an SQL error occurs.

The site at which a savepoint is recognized depends on whether the CONNECT statement is executed before or after the savepoint is set. For example, if an application executes the statement SET SAVEPOINT C1 at the local site before it executes a CONNECT TO S1 statement, savepoint C1 is known only at the local site. If the application executes CONNECT to S1 before SET SAVEPOINT C1, the savepoint is known only at site S1.

For more information about savepoints, see “Using savepoints to undo selected changes within a unit of work” on page 439.

Moving from DB2 private protocol access to DRDA access

Recommendation: If you do not already use DRDA access, move from DB2 private protocol access to DRDA access whenever possible. An application that uses DB2 private protocol access cannot include SQL statements that were added to DB2 after Version 7. Because DB2 supports three-part names, you can move to DRDA access without modifying your applications. For any application that uses DB2 private protocol access, follow these steps to make the application use DRDA access:

1. Determine which locations the application accesses.

For static SQL applications, search for all SQL statements that include three-part names and aliases for three-part names. For three-part names, the high-level qualifier is the location name. For potential aliases, query the catalog table SYSTABLES to determine whether the object is an alias, and if so, the location name of the table that the alias represents. For example:

```
SELECT NAME, CREATOR, LOCATION, TBcreator, TBNAME
FROM SYSIBM.SYSTABLES
WHERE NAME='name'
AND TYPE='A';
```

where *name* is the potential alias.

For dynamic SQL applications, bind packages at all remote locations that users might access with three-part names.

2. Bind the application into a package at every location that is named in the application. Also bind a package locally.

For an application that uses explicit CONNECT statements to connect to a second site and then accesses a third site using a three-part name, bind a package at the second site with DBPROTOCOL(DRDA), and bind another package at the third site.

3. Bind all remote packages into a plan with the local package or DBRM. Bind this plan with the option DBPROTOCOL(DRDA).
4. Ensure that aliases resolve correctly.

For DB2 private protocol access, DB2 resolves aliases at the requester site. For DRDA access, however, DB2 resolves aliases at the site where the package executes. Therefore, you might need to define aliases for three-part names at remote locations.

For example, suppose you use DRDA access to run a program that contains this statement:

```
SELECT * FROM MYALIAS;
```

MYALIAS is an alias for LOC2.MYID.MYTABLE. DB2 resolves MYALIAS at the local site to determine that this statement needs to run at LOC2 but does not send the resolved name to LOC2. When the statement executes at LOC2, DB2 resolves MYALIAS using the catalog at LOC2. If the catalog at LOC2 does not contain the alias MYID.MYTABLE for MYALIAS, the SELECT statement does not execute successfully.

This situation can become more complicated if you use three-part names to access DB2 objects from remote sites. For example, suppose you are connected explicitly to LOC2, and you use DRDA access to execute the following statement:

```
SELECT * FROM YRALIAS;
```

YRALIAS is an alias for LOC3.MYID.MYTABLE. When this SELECT statement executes at LOC3, both LOC2 and LOC3 must have an alias YRALIAS that resolves to MYID.MYTABLE at location LOC3.

5. If you use the resource limit facility at the remote locations that are specified in three-part names to control the amount of time that distributed dynamic SQL statements run, modify the resource limit specification tables at those locations.

For DB2 private protocol access, you specify plan names to govern SQL statements that originate at a remote location. For DRDA access, you specify package names for this purpose. Therefore, you must add rows to your resource limit specification tables at the remote locations for the packages you

bound for DRDA access with three-part names. You should also delete the rows that specify plan names for DB2 private protocol access.

For more information about the resource limit facility, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Bind processes for DRDA and DB2 private protocol access

The following examples explore different bind processes for DRDA and DB2 private protocol access.

Example: Suppose that you need to access data at a remote server CHICAGO, by using the following query:

```
SELECT * FROM CHICAGO.DSN8810.EMP
WHERE EMPNO = '0001000';
```

This statement can be executed with DRDA access or DB2 private protocol access. The method of access depends on whether you bind your DBRMs into packages and on the value of the DATABASE PROTOCOL field in installation panel DSN TIP5 or the value of bind option DBPROTOCOL. Bind option DBPROTOCOL overrides the installation setting.

If you bind the DBRM that contains the statement into a plan at the local DB2 and specify the bind option DBPROTOCOL(PRIVATE), you access the server by using DB2 private protocol access.

If you bind the DBRM that contains the statement by using one of the following processes, you access the server using DRDA access:

Local-bind DRDA access process:

1. Bind the DBRM into a package at the local DB2 using the bind option DBPROTOCOL(DRDA).
2. Bind the DBRM into a package at the remote location (CHICAGO).
3. Bind the packages into a plan using bind option DBPROTOCOL(DRDA).

Remote-bind DRDA access process:

1. Bind the DBRM into a package at the remote location.
2. Bind the remote package and the DBRM into a plan at the local site, using the bind option DBPROTOCOL(DRDA).

In some cases you cannot use private protocol to access distributed data. The following examples require DRDA access.

Example: Suppose that you need to access data at a remote server CHICAGO, by using the following CONNECT and SELECT statements:

```
EXEC SQL
CONNECT TO CHICAGO;
EXEC SQL SELECT * FROM DSN8810.EMP
WHERE EMPNO = '0001000';
```

This example requires DRDA access and the correct binding procedure to work from a remote server. Before you can execute the query at location CHICAGO, you must bind the application as a remote package at the CHICAGO server. Before you can run the application, you must also bind a local package and a local plan with a package list that includes the local and remote package.

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);
```

This example requires DRDA access because private protocol does not support stored procedures. 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.

Precompiler and bind options for DRDA access

For the most part, 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. For the general instructions, see Chapter 21, “Preparing an application program to run,” on page 471. This section describes the differences. For further information on improving the performance of distributed data access with bind options, see “Using bind options to improve performance for distributed applications” on page 456.

Precompiler options for DRDA access

The following precompiler options are relevant to preparing a package to be run using 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 UDB 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 UDB for z/OS only. The precompiler then rejects any statement that does not obey the rules of DB2 UDB for z/OS.

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. See “Using block fetch in distributed applications” on page 458 for more information.

DBPROTOCOL

Use **DBPROTOCOL(PRIVATE)** if you want DB2 to use DB2 private protocol access for accessing remote data that is specified with three-part names.

Use **DBPROTOCOL(DRDA)** if you want DB2 to use DRDA access to access remote data that is specified with three-part names. You must bind a package at all locations whose names are specified in three-part names.

The package value for the **DBPROTOCOL** option overrides the plan option. For example, if you specify **DBPROTOCOL(DRDA)** for a remote package and **DBPROTOCOL(PRIVATE)** for the plan, DB2 uses DRDA access when it accesses data at that location using a three-part name. If you do not specify any value for **DBPROTOCOL**, DB2 uses the value of **DATABASE PROTOCOL** on installation panel **DSNTIP5**.

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. See “Using block fetch in distributed applications” on page 458 for more information.

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.

DBPROTOCOL

Use **DBPROTOCOL(PRIVATE)** if you want DB2 to use DB2 private protocol access for accessing remote data that is specified with three-part names.

Use **DBPROTOCOL(DRDA)** if you want DB2 to use DRDA access to access remote data that is specified with three-part names. You must bind a package at all locations whose names are specified in three-part names.

These values override the value of **DATABASE PROTOCOL** on installation panel **DSNTIP5**. Therefore, if the setting of **DATABASE PROTOCOL** at the requester site specifies the type of remote access you want to use for three-part names, you do not need to specify the **DBPROTOCOL** bind option.

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 UDB 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.

Checking BIND PACKAGE options

You can request only the options of **BIND PACKAGE** that are supported by the server by specifying those options at the requester and using the requester’s syntax for **BIND PACKAGE**. 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 21, “Preparing an application program to run,” on page 471.
- For the syntax of DB2 **BIND** and **REBIND** subcommands, see Part 3 of *DB2 Command Reference*.

- For a list of DB2 bind options in generic terms, including options you cannot request from DB2 but can use if you request from a non-DB2 server, see Appendix I, “Program preparation options for remote packages,” on page 1123.

Coding methods for distributed data

You can use two primary coding methods to access distributed data:

- Three-part table names, as described in “Using three-part table names to access distributed data.”
- Explicit connect statements, as described in “Using explicit CONNECT statements to access distributed data” on page 450.

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 DSN8810.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 DSN8810.PROJ for each location.

Using three-part table names to access distributed data

You can use three-part table names to access data at a remote location through DRDA access or DB2 private protocol access. When you use three-part table names, the way you code your application is the same, regardless of the access method you choose. You determine the access method when you bind the SQL statements into a package or plan. If you use DRDA access, you must bind the DBRMs for the SQL statements to be executed at the server to packages that reside at that server.

See “Binding packages at a remote location” on page 496 for information about binding DBRMs at the server.

Recommendation: If you plan to port a distributed application that runs on a
 # z/OS client to another client, avoid the use of three-part names. If a client
 # application includes three-part names, and the z/OS client or Linux, UNIX, and
 # Windows client directly accesses a DB2 for Linux, UNIX, and Windows server, an
 # error occurs.

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.

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. (See Chapter 24, “Coding dynamic SQL in application programs,” on page 593 for the technique.) 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 three-part names:

```

Read input values
Do for all locations
    Read location name
    Set up statement to prepare
    Prepare statement
    Execute statement
End loop
Commit

```

After the application obtains a location name, for example 'SAN_JOSE', it next creates the following character string:

```
INSERT INTO SAN_JOSE.DSN8810.PROJ VALUES (?, ?, ?, ?, ?, ?, ?, ?)
```

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 in "Project table (DSN8810.PROJ)" on page 1000.

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.)

Recommendation: You might find it convenient to use aliases when creating character strings that become prepared statements, instead of using full three-part names like SAN_JOSE.DSN8810.PROJ. For information about aliases, see Chapter 5 of *DB2 SQL Reference*.

Three-part names and multiple servers

If you use a three-part name, or an alias that resolves to one, in a statement that is executed at a remote server by DRDA access, and if the location name is not that of the server, then the method by which the remote server accesses data at the named location depends on the value of DBPROTOCOL. If the package at the first remote server is bound with DBPROTOCOL(PRIVATE), DB2 uses DB2 private protocol access to access the second remote server. If the package at the first remote server is bound with DBPROTOCOL(DRDA), DB2 uses DRDA access to access the second remote server. The following steps are recommended so that access to the second remote server is by DRDA access:

1. Rebind the package at the first remote server with DBPROTOCOL(DRDA).
2. Bind the package that contains the three-part name at the second server.

Accessing declared temporary tables by using three-part names

You can access a remote declared temporary table by using a three-part name only if you use DRDA access. 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.

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 */
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)*/
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)*/

```

Using explicit CONNECT statements to access distributed data

With this method, 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.

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 DSN8810.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 in "Project table (DSN8810.PROJ)" on page 1000. LOCATION_NAME is a character-string variable of length 16.

Using a location alias name for multiple sites

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: SVL_EMPLOYEE and SJ_EMPLOYEE. To access each site, insert a row for each site into SYSIBM.LOCATIONS. 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 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.

Releasing connections

When you connect to remote locations explicitly, you must also break 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.

Example: 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;
- All DB2 private protocol connections. If the first phase of your application program uses DB2 private protocol access and the second phase uses DRDA access, open DB2 private protocol connections from the first phase could cause a

CONNECT operation to fail in the second phase. To prevent that error, execute the following statement before the commit operation that separates the two phases:

```
EXEC SQL RELEASE ALL PRIVATE;
```

PRIVATE refers to DB2 private protocol connections, which exist only between instances of DB2 UDB for z/OS.

Coordinating updates to two or more data sources

Definition: 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.

Updates to two or more DBMSs can be coordinated automatically when both systems implement a method called *two-phase commit*.

Recommendation: When you need to coordinate updates, work with systems that implement two-phase commit.

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.

Examples: The examples described in “Using three-part table names to access distributed data” on page 448 and “Using explicit CONNECT statements to access distributed data” on page 450 assume that all systems involved implement two-phase commit. Both examples suggest updating several systems in a loop and ending the unit of work by committing only when the loop is complete. In both cases, updates are coordinated across the entire set of systems.

For more information about the two-phase commit process, see Part 4 (Volume 1) of *DB2 Administration Guide*.

Working without two-phase commit

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

- 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.

Update restrictions on servers that do not support two-phase commit

You cannot really have coordinated updates with a DBMS that does not implement two-phase commit. In the description that follows, the DBMS is called a *restricted system*. DB2 prevents you from updating both a restricted system and any other system in the same unit of work. In this context, update includes the statements INSERT, DELETE, UPDATE, CREATE, ALTER, DROP, GRANT, REVOKE, RENAME, COMMENT, and LABEL.

To achieve the effect of coordinated updates with a restricted system, you must first update one system and commit that work, and then update the second system and commit its work. 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. Your program must perform that task.

CICS and IMS

You cannot update data at servers that do not support two-phase commit.

TSO and batch

You can update data if one of the following conditions is true:

- No other connections exist.
- All existing connections are to servers that are restricted to read-only operations.

If neither condition is met, you are restricted to read-only operations.

If the first connection in a logical unit of work is to a server that supports two-phase commit, and no connections exist or only read-only connections exist, 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.

Recommendation: Rely on DB2 to prevent updates to two systems in the same unit of work if either of them is a restricted system.

Forcing update restrictions by using CONNECT (Type 1)

You can restrict your program completely to the rules for restricted systems, by using the type 1 rules for CONNECT. To put these rules into effect for a package, use the precompiler option CONNECT(1). Be careful not to use packages that are precompiled with CONNECT(1) and packages that are precompiled with CONNECT(2) 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. An attempt to execute a later CONNECT statement that is precompiled with the other type returns an error.

For more information about CONNECT (Type 1) and about managing connections to other systems, see Chapter 1 of *DB2 SQL Reference*.

Maximizing performance for distributed data

When your application access distributed data, you should pay particularly close attention to performance. This section covers the following topics about maximizing performance for distributed data:

- “Coding efficient queries”
- “Maximizing LOB performance in a distributed environment”
- “Using bind options to improve performance for distributed applications” on page 456
- “Using block fetch in distributed applications” on page 458
- “Limiting the number of DRDA network transmissions” on page 461
- “Limiting the number of rows returned to DRDA clients” on page 464

Coding efficient queries

A query that is sent to a remote subsystem almost always takes longer to execute than the same query that accesses tables of the same size on the local subsystem. The principle causes are:

- Overhead processing, including startup, negotiating session limits, and, for DB2 private protocol access, the bind required at the remote location
- The time required to send messages across the network

To gain the greatest efficiency when accessing remote subsystems, write queries that send few messages over the network:

- Reduce the number of columns and rows in the result table that is returned to your application. Keep your SELECT lists as short as possible. Use WHERE, GROUP BY, and HAVING clauses to eliminate unwanted data at the remote server.
- Use FOR FETCH ONLY or FOR READ ONLY. For example, retrieving thousands of rows as a continuous stream is reasonable. Sending a separate message for each one can be significantly slower.
- When possible, do not bind application plans and packages with ISOLATION(RR), even though that is the default. If your application does not need to refer again to rows it has read once, another isolation level might reduce lock contention and message overhead during COMMIT processing.
- Minimize the use of parameter markers. When your program package causes the use of DRDA access, DB2 can streamline the processing of dynamic queries that do not have parameter markers.

When a DB2 requester encounters a PREPARE statement for such a query, it anticipates that the application is going to open a cursor. DB2 therefore sends a single message to the server that contains a combined request for PREPARE, DESCRIBE, and OPEN. A server that receives this message sequence will return a reply message sequence that includes the output from the PREPARE, DESCRIBE, and OPEN operations. As a result, the number of network messages sent and received for these operations is reduced from two to one. DB2 combines messages for these queries regardless of whether the bind option DEFER(PREPARE) is specified.

Maximizing LOB performance in a distributed environment

If you use DRDA access, you can access LOB columns in a remote table. Because LOB values are usually quite large, you should use the following techniques for data retrieval that minimize the number of bytes that are transferred between the client and server:

- **Use LOB locators instead of LOB host variables:** If you need to store only a portion of a LOB value at the client, or if your client program manipulates the LOB data but does not need a copy of it, LOB locators are a good choice. When a client program retrieves a LOB column from a server into a locator, DB2 transfers only the 4-byte locator value to the client, not the entire LOB value. For information about how to use LOB locators in an application, see “Using LOB locators to save storage” on page 305.
- **Use stored procedure result sets:** When you return LOB data to a client program from a stored procedure, use result sets, rather than passing the LOB data to the client in parameters. Using result sets to return data causes less LOB materialization and less movement of data among address spaces. For information about how to write a stored procedure to return result sets, see “Writing a stored procedure to return result sets to a DRDA client” on page 650. For information about how to write a client program to receive result sets, see “Writing a DB2 UDB for z/OS client program or SQL procedure to receive result sets” on page 708.
- **Set the CURRENT RULES special register to DB2:** When a DB2 UDB for z/OS server receives an OPEN request for a cursor, the server uses the value in the CURRENT RULES special register to determine the type of host variables the associated statement uses to retrieve LOB values. If you specify a value of DB2 for CURRENT RULES before you perform a CONNECT, and the first FETCH for the cursor uses a LOB locator to retrieve LOB column values, DB2 lets you use only LOB locators for all subsequent FETCH statements for that column until you close the cursor. If the first FETCH uses a host variable, DB2 lets you use only host variables for all subsequent FETCH statements for that column until you close the cursor. However, if you set the value of CURRENT RULES to STD, DB2 lets you use the same open cursor to fetch a LOB column into either a LOB locator or a host variable.

Although a value of STD for CURRENT RULES gives you more programming flexibility when you retrieve LOB data, you get better performance if you use a value of DB2 for CURRENT RULES. With the STD option, the server must send and receive network messages for each FETCH to indicate whether the data that is being transferred is a LOB locator or a LOB value. With the DB2 option, the server knows the size of the LOB data after the first FETCH, so an extra message about LOB data size is unnecessary. The server can send multiple blocks of data to the requester at one time, which reduces the total time for data transfer.

Example: Suppose that an end user wants to browse through a large set of employee records and look at pictures of only a few of those employees. At the server, you set the CURRENT RULES special register to DB2. In the application, you declare and open a cursor to select employee records. The application then fetches all picture data into 4-byte LOB locators. Because DB2 knows that 4 bytes of LOB data is returned for each FETCH, DB2 can fill the network buffers with locators for many pictures. When a user wants to see a picture for a particular person, the application can retrieve the picture from the server by assigning the value that is referenced by the LOB locator to a LOB host variable:

```
SQL TYPE IS BLOB my_blob[1M];
SQL TYPE IS BLOB AS LOCATOR my_loc;
:
:
FETCH C1 INTO :my_loc;    /* Fetch BLOB into LOB locator */
:
:
SET :my_blob = :my_loc; /* Assign BLOB to host variable */
```


Using bind options to improve performance for distributed applications

Your choice of these bind options can affect the performance of your distributed applications:

- DEFER(PREPARE) or NODEFER(PREPARE)
- PKLIST
- REOPT(ALWAYS), REOPT(ONCE), or REOPT(NONE)
- CURRENTDATA(YES) or CURRENTDATA(NO)
- KEEP_DYNAMIC(YES) or KEEP_DYNAMIC(NO)
- DBPROTOCOL(PRIVATE) or DBPROTOCOL(DRDA)

DEFER(PREPARE)

To improve performance for both static and dynamic SQL used in DB2 private protocol access, and for dynamic SQL in DRDA access, consider specifying the option DEFER(PREPARE) when you bind or rebind your plans or packages. Remember that statically bound SQL statements in DB2 private protocol access are processed dynamically. When a dynamic SQL statement accesses remote data, the PREPARE and EXECUTE statements can be transmitted over the network together and processed at the remote location. Responses to both statements can be sent together back to the local subsystem, thus reducing traffic on the network. DB2 does not prepare the dynamic SQL statement until the statement executes. (The exception to this is dynamic SELECT, which combines PREPARE and DESCRIBE, regardless of whether the DEFER(PREPARE) option is in effect.)

All PREPARE messages for dynamic SQL statements that refer to a remote object will be deferred until one of these events occurs:

- The statement executes
- The application requests a description of the results of the statement

In general, when you defer PREPARE, DB2 returns SQLCODE 0 from PREPARE statements. You must therefore code your application to handle any SQL codes that might have been returned from the PREPARE statement after the associated EXECUTE or DESCRIBE statement.

When you use predictive governing, the SQL code returned to the requester if the server exceeds a predictive governing warning threshold depends on the level of DRDA at the requester. See “Writing an application to handle predictive governing” on page 602 for more information.

For DB2 private protocol access, when a static SQL statement refers to a remote object, the transparent PREPARE statement and the EXECUTE statements are automatically combined and transmitted across the network together. The PREPARE statement is deferred only if you specify the bind option DEFER(PREPARE).

PREPARE statements that contain INTO clauses are not deferred.

PKLIST

The order in which you specify package collections in a package list can affect the performance of your application program. When a local instance of DB2 attempts to execute an SQL statement at a remote server, the local DB2 subsystem must determine which package collection the SQL statement is in. DB2 must send a message to the server, requesting that the server check each collection ID for the SQL statement, until the statement is found or no more collection IDs are in the package list. You can reduce the amount of network traffic, and thereby improve

performance, by reducing the number of package collections that each server must search. The following examples show ways to reduce the collections to search:

- Reduce the number of packages per collection that must be searched. The following example specifies only one package in each collection:
`PKLIST(S1.COLLA.PGM1, S1.COLLB.PGM2)`
- Reduce the number of package collections at each location that must be searched. The following example specifies only one package collection at each location:
`PKLIST(S1.COLLA.*, S2.COLLB.*)`
- Reduce the number of collections that are used for each application. The following example specifies only one collection to search:
`PKLIST(*.COLLA.*)`

You can also specify the package collection that is associated with an SQL statement in your application program. Execute the SQL statement `SET CURRENT PACKAGESET` before you execute an SQL statement to tell DB2 which package collection to search for the statement.

When you use `DEFER(PREPARE)` with DRDA access, the package containing the statements whose preparation you want to defer must be the first qualifying entry in the package search sequence that DB2 uses. (See “Identifying packages at run time” on page 498 for more information.) For example, assume that the package list for a plan contains two entries:

```
PKLIST(LOCB.COLLA.*, LOCB.COLLB.*)
```

If the intended package is in collection `COLLB`, ensure that DB2 searches that collection first. You can do this by executing the SQL statement:

```
SET CURRENT PACKAGESET = 'COLLB';
```

Alternatively, you can list `COLLB` first in the `PKLIST` parameter of `BIND PLAN`:

```
PKLIST(LOCB.COLLB.*, LOCB.COLLA.*)
```

For `NODEFER(PREPARE)`, the collections in the package list can be in any order, but if the package is not found in the first qualifying `PKLIST` entry, the result is significant network overhead for searching through the list.

REOPT(ALWAYS)

When you specify `REOPT(ALWAYS)`, DB2 determines access paths at both bind time and run time for statements that contain one or more of the following variables:

- Host variables
- Parameter markers
- Special registers

At run time, DB2 uses the values in those variables to determine the access paths.

If you specify the bind option `REOPT(ALWAYS)` or `REOPT(ONCE)`, DB2 sets the bind option `DEFER(PREPARE)` automatically. However, when you specify `REOPT(ONCE)`, DB2 determines the access path for a statement only once (at the first run time).

Because of performance costs when DB2 reoptimizes the access path at run time, you should use one of the following bind options:

- `REOPT(ALWAYS)` — use this option only on packages or plans that contain statements that perform poorly because of a bad access path.

- REOPT(ONCE) — use this option when the following conditions are true:
 - You are using the dynamic statement cache.
 - You have plans or packages that contain dynamic SQL statements that perform poorly because of access path selection.
 - Your dynamic SQL statements are executed many times with possibly different input variables.
- REOPT(NONE) — use this option when you bind a plan or package that contains statements that use DB2 private protocol access.

If you specify REOPT(ALWAYS) when you bind a plan that contains statements that use DB2 private protocol access to access remote data, DB2 prepares those statements twice. See “How bind options REOPT(ALWAYS) and REOPT(ONCE) affect dynamic SQL” on page 625 for more information about REOPT(ALWAYS).

CURRENTDATA(NO)

Use this bind option to force block fetch for ambiguous queries. See “Using block fetch in distributed applications” for more information about block fetch.

KEEPDYNAMIC(YES)

Use this bind option to improve performance for queries that use cursors defined WITH HOLD. With KEEPDYNAMIC(YES), DB2 automatically closes the cursor when no more data exists for retrieval. The client does not need to send a network message to tell DB2 to close the cursor. For more information about KEEPDYNAMIC(YES), see “Keeping prepared statements after commit points” on page 599.

DBPROTOCOL(DRDA)

If the value of installation default DATABASE PROTOCOL is not DRDA, use this bind option to cause DB2 to use DRDA access to execute SQL statements with three-part names. Statements that use DRDA access perform better at execution time because:

- Binding occurs when the package is bound, not during program execution.
- DB2 does not destroy static statement information at commit time, as it does with DB2 private protocol access. This means that with DRDA access, if a commit occurs between two executions of a statement, DB2 does not need to prepare the statement twice.

Using block fetch in distributed applications

DB2 uses two different methods to reduce the number of messages that are sent across the network when fetching data with a cursor:

- *Limited block fetch* optimizes data transfer by guaranteeing the transfer of a minimum amount of data in response to each request from the requesting system.
- *Continuous block fetch* sends a single request from the requester to the server. The server fills a buffer with data it retrieves and transmits it back to the requester. Processing at the requester is asynchronous with the server; the server continues to send blocks of data to the requester with minimal or no further prompting.

For more information about block fetch, see Part 5 (Volume 2) of *DB2 Administration Guide*.

To use either type of block fetch, DB2 must determine that the cursor is not used for updating or deleting. Indicate that the cursor does not modify data by adding

FOR FETCH ONLY or FOR READ ONLY to the query in the DECLARE CURSOR statement. If you do not use FOR FETCH ONLY or FOR READ ONLY, DB2 still uses block fetch for the query if any of the following conditions are true:

- The cursor is a non-scrollable cursor, and the result table of the cursor is read-only. (See Chapter 5 of *DB2 SQL Reference* for a description of read-only tables.)
- The cursor is a scrollable cursor that is declared as INSENSITIVE, and the result table of the cursor is read-only.
- The cursor is a scrollable cursor that is declared as SENSITIVE, the result table of the cursor is read-only, and the value of bind option CURRENTDATA is NO.
- The result table of the cursor is not read-only, but the cursor is ambiguous, and the value of bind option CURRENTDATA is NO. A cursor is ambiguous when any of the following conditions are true:
 - It is not defined with the clauses FOR FETCH ONLY, FOR READ ONLY, or FOR UPDATE.
 - It is not defined on a read-only result table.
 - It is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement.
 - It is in a plan or package that contains the SQL statements PREPARE or EXECUTE IMMEDIATE.

DB2 does not use continuous block fetch if:

- The cursor is referred to in the statement DELETE WHERE CURRENT OF elsewhere in the program.
- The cursor statement appears that it can be updated at the requesting system. (DB2 does not check whether the cursor references a view at the server that cannot be updated.)

When DB2 uses block fetch for non-scrollable cursors

Table 59 summarizes the conditions under which a DB2 server uses block fetch for a non-scrollable cursor.

Table 59. Effect of CURRENTDATA and isolation level on block fetch for a non-scrollable cursor

Isolation	CURRENTDATA	Cursor type	Block fetch
CS, RR, or RS	YES	Read-only	Yes
		Updatable	No
		Ambiguous	No
	No	Read-only	Yes
		Updatable	No
		Ambiguous	Yes
UR	Yes	Read-only	Yes
	No	Read-only	Yes

When DB2 uses block fetch for scrollable cursors

Table 60 summarizes the conditions under which a DB2 server uses block fetch for a scrollable cursor when the cursor is not used to retrieve result sets.

Table 60. Effect of CURRENTDATA and isolation level on block fetch for a scrollable cursor that is not used for a stored procedure result set

Isolation	Cursor sensitivity	CURRENTDATA	Cursor type	Block fetch
CS, RR, or RS	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	No
			Updatable	No
			Ambiguous	No
		No	Read-only	Yes
			Updatable	No
			Ambiguous	Yes
UR	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes

Table 61 summarizes the conditions under which a DB2 server uses block fetch for a scrollable cursor when the cursor is used to retrieve result sets.

Table 61. Effect of CURRENTDATA and isolation level on block fetch for a scrollable cursor that is used for a stored procedure result set

Isolation	Cursor sensitivity	CURRENTDATA	Cursor type	Block fetch
CS, RR, or RS	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	No
		No	Read-only	Yes
UR	INSENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes
	SENSITIVE	Yes	Read-only	Yes
		No	Read-only	Yes

When a DB2 UDB for z/OS requester uses a scrollable cursor to retrieve data from a DB2 UDB for z/OS server, the following conditions are true:

- The requester never requests more than 64 rows in a query block, even if more rows fit in the query block. In addition, the requester never requests extra query blocks. This is true even if the setting of field EXTRA BLOCKS REQ in the DISTRIBUTED DATA FACILITY PANEL 2 installation panel on the requester allows extra query blocks to be requested. If you want to limit the number of rows that the server returns to fewer than 64, you can specify the FETCH FIRST *n* ROWS ONLY clause when you declare the cursor.
- The requester discards rows of the result table if the application does not use those rows. For example, if the application fetches row *n* and then fetches row *n*+2, the requester discards row *n*+1. The application gets better performance for a blocked scrollable cursor if it mostly scrolls forward, fetches most of the rows in a query block, and avoids frequent switching between FETCH ABSOLUTE statements with negative and positive values.

- If the scrollable cursor does not use block fetch, the server returns one row for each FETCH statement.

Limiting the number of DRDA network transmissions

You can use the clause OPTIMIZE FOR n ROWS in your SELECT statements to limit the number of data rows that the server returns on each DRDA network transmission. You can also use OPTIMIZE FOR n ROWS with query result sets from stored procedures. OPTIMIZE FOR n ROWS has no effect on scrollable cursors.

The number of rows that DB2 transmits on each network transmission depends on the following factors:

- If n rows of the SQL result set fit within a single DRDA query block, a DB2 server can send n rows to any DRDA client. In this case, DB2 sends n rows in each network transmission, until the entire query result set is exhausted.
- If n rows of the SQL result set exceed a single DRDA query block, the number of rows that are contained in each network transmission depends on the client's DRDA software level and configuration:
 - If the client does not support extra query blocks, the DB2 server automatically reduces the value of n to match the number of rows that fit within a DRDA query block.
 - If the client supports extra query blocks, the DRDA client can choose to accept multiple DRDA query blocks in a single data transmission. DRDA allows the client to establish an upper limit on the number of DRDA query blocks in each network transmission.

The number of rows that a DB2 server sends is the smaller of n rows and the number of rows that fit within the lesser of these two limitations:

- The value of EXTRA BLOCKS SRV in install panel DSNTIP5 at the DB2 server

This is the maximum number of extra DRDA query blocks that the DB2 server returns to a client in a single network transmission.

- The client's extra query block limit, which is obtained from the DDM MAXBLKEXT parameter that is received from the client

When DB2 acts as a DRDA client, the DDM MAXBLKEXT parameter is set to the value that is specified on the EXTRA BLOCKS REQ install option of the DSNTIP5 install panel.

The OPTIMIZE FOR n ROWS clause is useful in two cases:

- If n is less than the number of rows that fit in the DRDA query block, OPTIMIZE FOR n ROWS can improve performance by preventing the DB2 server from fetching rows that might never be used by the DRDA client application.
- If n is greater than the number of rows that fit in a DRDA query block, OPTIMIZE FOR n ROWS lets the DRDA client request multiple blocks of query data on each network transmission. This use of OPTIMIZE FOR n ROWS can significantly improve elapsed time for large query download operations.

Specifying a large value for n in OPTIMIZE FOR n ROWS can increase the number of DRDA query blocks that a DB2 server returns in each network transmission. This function can significantly improve performance for applications that use DRDA access to download large amounts of data. However, this same function can

degrade performance if you do not use it properly. The following examples demonstrate the performance problems that can occur when you do not use `OPTIMIZE FOR n ROWS` judiciously.

In Figure 149,, the DRDA client opens a cursor and fetches rows from the cursor. At some point before all rows in the query result set are returned, the application issues an SQL INSERT. DB2 uses normal DRDA blocking, which has two advantages over the blocking that is used for `OPTIMIZE FOR n ROWS`:

- If the application issues an SQL statement other than `FETCH` (the example shows an `INSERT` statement), the DRDA client can transmit the SQL statement immediately, because the DRDA connection is not in use after the `SQL OPEN`.
- If the SQL application closes the cursor before fetching all the rows in the query result set, the server fetches only the number of rows that fit in one query block, which is 100 rows of the result set. Basically, the DRDA query block size places an upper limit on the number of rows that are fetched unnecessarily.

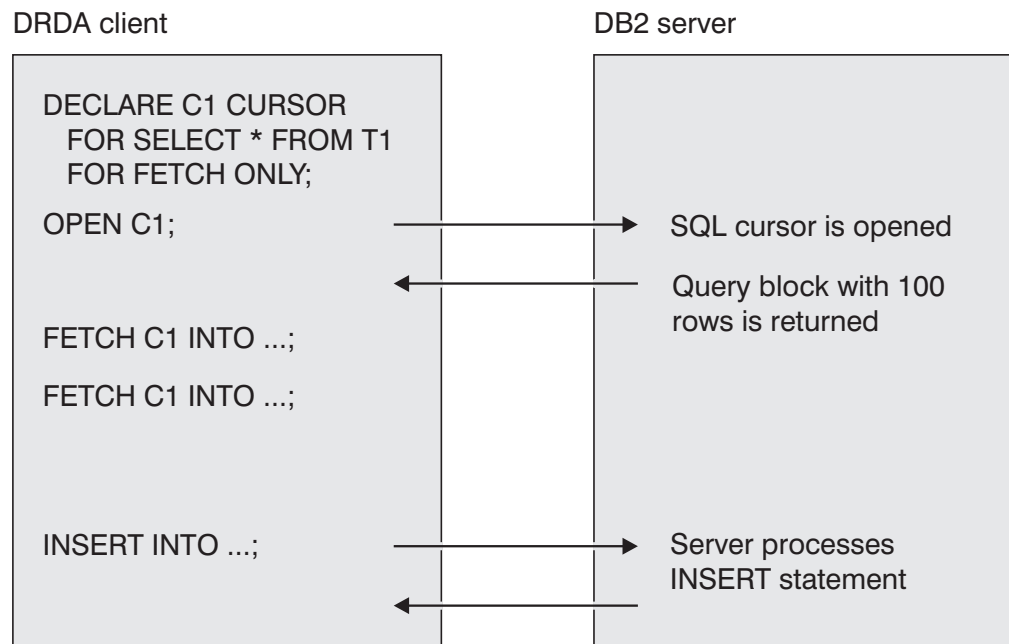


Figure 149. Message flows without `OPTIMIZE FOR n ROWS`

In Figure 150 on page 463,, the DRDA client opens a cursor and fetches rows from the cursor using `OPTIMIZE FOR n ROWS`. Both the DRDA client and the DB2 server are configured to support multiple DRDA query blocks. At some time before the end of the query result set, the application issues an SQL INSERT. Because `OPTIMIZE FOR n ROWS` is being used, the DRDA connection is not available when the SQL INSERT is issued because the connection is still being used to receive the DRDA query blocks for 1000 rows of data. This causes two performance problems:

- Application elapsed time can increase if the DRDA client waits for a large query result set to be transmitted, before the DRDA connection can be used for other SQL statements. Figure 150 on page 463 shows how an SQL INSERT statement can be delayed because of a large query result set.
- If the application closes the cursor before fetching all the rows in the SQL result set, the server might fetch a large number of rows unnecessarily.

Limiting the number of rows returned to DRDA clients

You can use the `FETCH FIRST  $n$  ROWS ONLY` clause of a `SELECT` statement to limit the number of rows that are returned to a client program. `FETCH FIRST  $n$  ROWS ONLY` improves performance of DRDA client applications when the client needs no more than n rows from a potentially large result table.

If you specify `FETCH FIRST  $n$  ROWS ONLY` and do not specify `OPTIMIZE FOR  $n$  ROWS`, the access path for the statement will use the value that is specified for `FETCH FIRST  $n$  ROWS ONLY` for optimization. However, DRDA will not consider the value when it determines network blocking.

When you specify both the `FETCH FIRST  $n$  ROWS ONLY` clause and the `OPTIMIZE FOR  $m$  ROWS` clause in a statement, DB2 uses the value that you specify for `OPTIMIZE FOR  $m$  ROWS`, even if that value is larger than the value that you specify for the `FETCH FIRST  $n$  ROWS ONLY` clause.

Fast implicit close and `FETCH FIRST $n$ ROWS ONLY`

Fast implicit close means that during a distributed query, the DB2 server automatically closes the cursor after it prefetches the n th row if you specify `FETCH FIRST  $n$  ROWS ONLY`, or when there are no more rows to return. Fast implicit close can improve performance because it saves an additional network transmission between the client and the server.

DB2 uses fast implicit close when the following conditions are true:

- The query uses limited block fetch.
- The query retrieves no LOBs.
- The cursor is not a scrollable cursor.
- Either of the following conditions is true:
 - The cursor is declared `WITH HOLD`, and the package or plan that contains the cursor is bound with the `KEEPDYNAMIC(YES)` option.
 - The cursor is not defined `WITH HOLD`.

When you use `FETCH FIRST  $n$  ROWS ONLY`, and DB2 does a fast implicit close, the DB2 server closes the cursor after it prefetches the n th row, or when there are no more rows to return.

Example of `FETCH FIRST $n$ ROWS ONLY`

In a DRDA environment, the following SQL statement causes DB2 to prefetch 16 rows of the result table even though n has a value of 1.

```
SELECT * FROM EMP
  OPTIMIZE FOR 1 ROW ONLY;
```

For `OPTIMIZE FOR  $n$  ROWS`, when n is 1, 2, or 3, DB2 uses the value 16 (instead of n) for network blocking and prefetches 16 rows. As a result, network usage is more efficient even though DB2 uses the small value of n for query optimization.

Suppose that you need only one row of the result table. To avoid 15 unnecessary prefetches, add the `FETCH FIRST 1 ROW ONLY` clause:

```
SELECT * FROM EMP
  OPTIMIZE FOR 1 ROW ONLY
  FETCH FIRST 1 ROW ONLY;
```

Limiting the number of rows with the rowset parameter

The rowset parameter can be used in ODBC and JDBC applications on some platforms to limit the number of rows that are returned from a fetch operation. If a DRDA requester sends the rowset parameter to a DB2 UDB for z/OS server, the server does the following things:

- Returns no more than the number of rows in the rowset parameter
- Returns extra query blocks if the value of field EXTRA BLOCKS SRV in the DISTRIBUTED DATA FACILITY PANEL 2 installation panel on the server allows extra query blocks to be returned
- Processes the FETCH FIRST *n* ROWS ONLY clause, if it is specified
- Does not process the OPTIMIZE FOR *n* ROWS clause

Working with distributed data

The following sections discuss various issues related to working with data in a distributed environment:

- “SQL limitations at dissimilar servers”
- “Executing long SQL statements in a distributed environment” on page 466
- “Retrieving data from ASCII or Unicode tables” on page 466
- “Accessing data with a scrollable cursor when the requester is down-level” on page 467
- “Accessing data with a rowset-positioned cursor when the requester is down-level” on page 467
- “Maintaining data currency by using cursors” on page 467
- “Copying a table from a remote location” on page 467
- “Transmitting mixed data” on page 467

SQL limitations at dissimilar servers

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. *DB2 SQL Reference* tells what DB2 UDB for z/OS supports; similar documentation is usually available for other products. The following examples suggest what to expect from dissimilar servers:

- They support SELECT, INSERT, UPDATE, DELETE, DECLARE CURSOR, and FETCH, but details vary.

Example: In an UPDATE statement for DB2 UDB for z/OS, you cannot use DEFAULT on the right side of the assignment clause; for DB2 UDB for LUW, you can use DEFAULT. Any UPDATE statement that uses DEFAULT as the value to be assigned cannot run across all platforms.

- Data definition statements vary more widely.

Example: DB2 UDB for z/OS supports ROWID columns; DB2 UDB for LUW 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 UDB 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. For a list of those statements, see Appendix H, “Characteristics of SQL statements in DB2 UDB for z/OS,” on page 1113.

- 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.

Executing long SQL statements in a distributed environment

Remote access allows large SQL statements to flow between the requester and the server. With this support, a distributed application can send prepared SQL statements greater than 32 KB to the server. If the statements are greater than 32 KB, the server must support these long statements. If you are using DB2 private protocol access, long SQL statements are not supported.

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.

Retrieving data from 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 UDB 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 UDB 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.

See Part 2 of *DB2 Command Reference* for more information about the ENCODING bind options.

- For static or dynamic SQL

An application program can specify overriding CCSIDs for individual host variables in DECLARE VARIABLE statements. See “Changing the coded character set ID of host variables” on page 85 for information about how to specify the CCSID for a host variable.

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. See “Changing the CCSID for retrieved data” on page 619 for information about how to specify an overriding CCSID in an SQLDA.

Accessing data with a scrollable cursor when the requester is down-level

If a DB2 UDB 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 UDB 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 UDB 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.

Accessing data with a rowset-positioned cursor when the requester is down-level

If a DB2 UDB 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 UDB 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.

Maintaining data currency by using cursors

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.

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.

Transmitting mixed 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

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;
```

Part 5. Developing your application

Chapter 21. Preparing an application program to run	471
Steps in program preparation	472
Step 1: Process SQL statements	473
Using the DB2 precompiler.	474
# Using the DB2 coprocessor for C programs	478
# Using the DB2 coprocessor for C++ programs	479
# Using the DB2 coprocessor for COBOL programs	480
# DB2 coprocessor for PL/I programs	481
# Differences between the DB2 precompiler and DB2 coprocessor	482
Options for SQL statement processing	483
Translating command-level statements in a CICS program	493
Step 2: Compile (or assemble) and link-edit the application	494
Step 3: Bind the application	495
Binding a DBRM to a package.	496
Binding an application plan	498
Identifying packages at run time	498
Using BIND and REBIND options for packages and plans	502
Using packages with dynamic plan selection	507
Step 4: Run the application.	508
DSN command processor	508
Running a program in TSO foreground.	509
Running a batch DB2 application in TSO	510
Calling applications in a command procedure (CLIST).	511
Running a DB2 REXX application	512
Using JCL procedures to prepare applications	512
Available JCL procedures	512
Including code from SYSLIB data sets	513
Starting the precompiler dynamically	514
Precompiler option list format.	514
DDNAME list format.	514
Page number format	515
An alternative method for preparing a CICS program	516
Using JCL to prepare a program with object-oriented extensions	517
Using ISPF and DB2 Interactive	518
DB2I help	518
DB2I Primary Option Menu	518
DB2 Program Preparation panel	521
DB2I Defaults Panel 1	526
DB2I Defaults Panel 2	527
Precompile panel	528
Bind/Rebind/Free selection panel	531
Bind Package panel	533
Bind Plan panel	536
Rebind Package panel	539
Rebind Trigger Package panel	540
Rebind Plan panel.	542
Free Package panel	544
Free Plan panel.	545
The Defaults for Bind or Rebind Package or Plan panels	546
System Connection Types panel	550
Panels for entering lists of values.	552
Program Preparation: Compile, Link, and Run panel	553
The Run panel	555
Chapter 22. Testing an application program	557
Establishing a test environment	557

Designing a test data structure	557
Analyzing application data needs	557
Obtaining authorization	559
Creating a comprehensive test structure	559
Filling the tables with test data	559
Testing SQL statements using SPUFI.	560
Debugging your program	560
Debugging programs in TSO	560
Language test facilities	561
The TSO TEST command	561
Debugging programs in IMS	561
Debugging programs in CICS	562
Debugging aids for CICS	562
CICS execution diagnostic facility	563
Locating the problem.	566
Analyzing error and warning messages from the precompiler	567
SYSTEM output from the precompiler	567
SYSPRINT output from the precompiler	568
Chapter 23. Processing DL/I batch applications	573
Planning to use DL/I batch applications	573
Features and functions of DB2 DL/I batch support.	573
Requirements for using DB2 in a DL/I batch job	574
Authorization	574
Program design considerations	574
Address spaces	574
Commits	575
SQL statements and IMS calls	575
Checkpoint calls	575
Application program synchronization	575
Checkpoint and XRST considerations	575
Synchronization call abends	576
Input and output data sets for DL/I batch jobs	576
DB2 DL/I batch input	576
DB2 DL/I batch output	578
Preparation guidelines for DL/I batch programs	578
Precompiling	578
Binding	578
Link-editing	579
Loading and running	579
Submitting a DL/I batch application using DSNMTV01	579
Submitting a DL/I batch application without using DSNMTV01	580
Restart and recovery	580
JCL example of a batch backout	581
JCL example of restarting a DL/I batch job	581
Finding the DL/I batch checkpoint ID	582

Chapter 21. Preparing an application program to run

DB2 applications require different methods of program preparation depending on the type of the application:

- Applications that contain embedded static or dynamic SQL statements
- Applications that contain ODBC calls
- Applications in interpreted languages, such as REXX
- Java applications, which can contain JDBC calls or embedded SQL statements

Before you can run DB2 applications of the first type, you must precompile, compile, link-edit, and bind them. This chapter details the steps needed to prepare and run this type of application program; see “Steps in program preparation” on page 472.

You can control the steps in program preparation by using the following methods:

- “Using JCL procedures to prepare applications” on page 512
- “Using ISPF and DB2 Interactive” on page 518

For information about applications with ODBC calls that pass dynamic SQL statements as arguments, see *DB2 ODBC Guide and Reference*.

For information about running REXX programs, which you do not prepare for execution, see “Running a DB2 REXX application” on page 512.

For information about preparing and running Java programs, see *DB2 Application Programming Guide and Reference for Java*.

Productivity hint: To avoid rework, first test your SQL statements using SPUIFI, and then compile your program *without* SQL statements and resolve all compiler errors. Then proceed with the preparation and the DB2 precompile and bind steps.

SQL statement processing: Because most compilers do not recognize SQL statements, you can use the DB2 precompiler before you compile the program to prevent compiler errors. The other alternative is to use a host language SQL coprocessor. (An SQL coprocessor performs DB2 precompiler functions at compile time.) 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). Bind this DBRM to a package or plan using the BIND subcommand. (For information about packages and plans, see Chapter 17, “Planning for DB2 program preparation,” on page 381.) When you complete these steps, you can run your DB2 application.

CCSID conversion of source programs: If the SQL statements of your source program are not in Unicode UTF-8, the DB2 Version 8 precompiler converts them to UTF-8 for parsing. The precompiler uses the source CCSID(*n*) value, as described in Table 64 on page 484, to convert from that CCSID to CCSID 1208 (UTF-8). 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. For information about indirect conversions, see *z/OS Support for Unicode*.

Dependencies and binding for DB2 Version 8: Table 62 on page 472 summarizes the relationship between characteristics of SQL processing in source programs and the

ability to bind in various DB2 releases and modes. This relationship is affected by the SQL processing option NEWFUN and the new-function mode of DB2 Version 8:

- If you use the option NEWFUN(NO), the SQL statements in the DBRM use EBCDIC. As a result, the DBRM is not a DB2 Version 8 object. DB2 Version 7 and earlier releases can bind the DBRM. NEWFUN(NO) causes the compiler to reject any DB2 Version 8 functions.
- If you use the option NEWFUN(YES) to process the SQL statements in your program, the SQL statements in the DBRM use Unicode UTF-8. As a result, the DBRM is a DB2 Version 8 object even if the application program does not use any DB2 Version 8 functions. Therefore, the DBRM is Version 8 dependent. DB2 Version 8 can bind the DBRM; DB2 Version 7 and earlier releases cannot bind the DBRM. Version 8 can bind the DBRM even before Version 8 new-function mode if the DBRM does not use any DB2 Version 8 functions.
- If the application program uses DB2 Version 8 functions, Version 8 can bind the DBRM only in new-function mode for Version 8 (or later). If the program does not use any DB2 Version 8 functions, Version 8 can bind the DBRM even before Version 8 new-function mode.

Table 62. Dependencies and binding for DB2 Version 8

Value of NEWFUN	NO ¹	YES ²	YES ³
Is the DBRM a DB2 Version 8 object and dependent on DB2 Version 8?	No	Yes	Yes
Can DB2 Version 7, or an earlier release, bind the DBRM?	Yes	No	No
Can DB2 Version 8 bind before DB2 Version 8 new-function mode?	Yes	Yes	No
Can DB2 Version 8 bind while in DB2 Version 8 new-function mode?	Yes	Yes	Yes

Notes:

1. The DBRM is created with NEWFUN(NO), which prevents the use of DB2 Version 8 functions.
2. The DBRM is created with NEWFUN(YES), although the program does not use any DB2 Version 8 functions.
3. The DBRM is created with NEWFUN(YES), and the program uses DB2 Version 8 functions.

For more information about the NEWFUN option, see Table 64 on page 484. For information about DB2 Version 8 new-function mode, see *DB2 Installation Guide*. For information about the DSNHPC7 precompiler, see the Installation guide.

Steps in program preparation

The following sections provide details on preparing and running a DB2 application:

- “Step 1: Process SQL statements” on page 473
- “Step 2: Compile (or assemble) and link-edit the application” on page 494
- “Step 3: Bind the application” on page 495
- “Step 4: Run the application” on page 508.

As described in Chapter 17, “Planning for DB2 program preparation,” on page 381, binding a package is not necessary in all cases. In these instructions, it is assumed 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 493
- “Calling applications in a command procedure (CLIST)” on page 511

Step 1: Process SQL statements

One step in preparing an SQL application to run is processing SQL statements in the program. The SQL statements must be replaced with calls to DB2 language interface modules, and a DBRM must be created.

For assembler or Fortran applications, use the DB2 precompiler to prepare the SQL statements. See “Using the DB2 precompiler” on page 474.

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 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:

- For C/C++, invoke the DB2 coprocessor by specifying the SQL compiler option. For C, you need z/OS C/C++ Version 1 Release 5 with APAR PK14587. For C++, you need z/OS C/C++ Version 1 Release 6 with APAR PK31682. You can also use later versions of z/OS C/C++ with the appropriate APARs applied. For a list of those required C/C++ APARs, see *DB2 Program Directory*. For more information about how to use the C/C++ DB2 coprocessor, see *z/OS C/C++ Programming Guide* for Version 1 Release 5 or later.
- For COBOL, invoke the DB2 coprocessor by specifying the SQL compiler option. You need Enterprise COBOL Version 3 Release 2 or later. If your application requires DB2 Version 8 new functions, you also need COBOL APAR PQ83744 for Enterprise COBOL Version 3 Release 2. For more information about using the COBOL DB2 coprocessor, see “Using the DB2 coprocessor for COBOL programs” on page 480 and *Enterprise COBOL for z/OS Programming Guide* for Version 3 Release 2 or later.
- For PL/I, invoke the DB2 coprocessor by specifying the PL/I preprocessor option PP(SQL('option,...')). You need Enterprise PL/I Version 3 Release 2 or later. If your application requires DB2 Version 8 new functions, you also need PL/I APAR PQ84513 for Enterprise PL/I Version 3 Release 2. For more information about using the PL/I DB2 coprocessor, see “DB2 coprocessor for PL/I programs” on page 481 and *IBM Enterprise PL/I for z/OS Programming Guide* for Version 3 Release 2 or later.

For information about the differences between the DB2 precompiler and the DB2 coprocessor, see “Differences between the DB2 precompiler and DB2 coprocessor” on page 482.

In this section, references to the *SQL statement processor* apply to either the DB2 precompiler or the DB2 coprocessor. References to the *DB2 precompiler* apply specifically to the precompiler that is provided with DB2.

CICS

If the application contains CICS commands, you must translate the program before you compile it. (See “Translating command-level statements in a CICS program” on page 493.)

DB2 version for the DSNHDECP module: When you process SQL statements, the DB2 version in the DSNHDECP data-only load module must match the DB2

version of the DB2 precompiler or DB2 coprocessor. Otherwise, DB2 issues an
error, and SQL statement processing terminates.

Using the DB2 precompiler

- To start the precompile process, use one of the following methods:
- DB2I panels. Use the Precompile panel or the DB2 Program Preparation panels.
 - The DSNH command procedure (a TSO CLIST). For a description of that CLIST, see Part 3 of *DB2 Command Reference*.
 - JCL procedures supplied with DB2. See “Available JCL procedures” on page 512 for more information about this method.

When you precompile your program, DB2 does not need to be active. The precompiler does not validate the names of tables and columns that are used in SQL statements. However, the precompiler checks table and column references against SQL DECLARE TABLE statements in the program. Therefore, you should use DCLGEN to obtain accurate SQL DECLARE TABLE statements.

You might need to precompile and compile program source statements several times before they are error-free and ready to link-edit. During that time, you can get complete diagnostic output from the DB2 precompiler by specifying the SOURCE and XREF precompiler options.

A summary of all of the data sets that the precompiler uses are listed in the
following table.

Table 63. 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. If a partitioned data set is allocated to DBRMLIB, the data set name must include a member name.	Yes

Table 63. DD statements and data sets that the DB2 precompiler uses (continued)

DD statement	Data set description	Required?
STEPLIB	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. 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.	No, but recommended
SYSCIN	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.	Yes
SYSIN	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.	Yes
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

Table 63. DD statements and data sets that the DB2 precompiler uses (continued)

DD statement	Data set description	Required?
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.	Yes
SYSTEM	Terminal output file, which contains diagnostic messages from the DB2 precompiler.	No

Input to the precompiler: The primary input for the precompiler consists of statements in the host programming language and embedded SQL statements.

Important

The size of a source program that DB2 can precompile is limited by the region size and the virtual memory available to the precompiler. The maximum region size and memory available to the DB2 precompiler is usually around 8 MB, but these amounts vary with each system installation.

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.

There are limits on the forms of source statements that can pass through the precompiler. 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. You might want to run the host compiler before the precompiler to find the source statements that are unacceptable to the host compiler. At this point you can ignore the compiler error messages for SQL statements. After the source statements are free of unacceptable compiler errors, you can then perform the normal DB2 program preparation process for that host language.

The following restrictions apply only to the DB2 precompiler:

- 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.

Output from the precompiler: The following sections describe various kinds of output from the precompiler.

Listing output: The output data set, SYSPRINT, used to print output from the precompiler, 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 always display as they appear in the listing. However, DB2 stores statement numbers greater than 32767 as 0 in the DBRM.

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.

Terminal diagnostics: If a terminal output file, SYSTERM, is present, 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 major output from the precompiler is a database request module (DBRM). That data set contains the SQL statements and host variable information 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, TSO commands COPY and DELETE, or other PDS management tools for maintaining these data sets.

In a DBRM, the SQL statements and the list of host variable names use the following character encoding schemes:

- EBCDIC, for the result of a DB2 Version 8 precompilation with NEWFUN NO or a precompilation in an earlier release of DB2
- Unicode UTF-8, for the result of a DB2 Version 8 precompilation with NEWFUN YES

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. In a Version 8 precompilation, the DBRM dependency marker (DBRMPDRM) in the header of a DBRM is marked for Version 8 if the value of NEWFUN is YES; otherwise, it is not marked for Version 8.

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. If you plan to run multiple precompilation jobs and are not using the DFSMSdfp[™] partitioned data set extended (PDSE), you must 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.

Binding on another system: It is not necessary to precompile the program on the same DB2 system on which you bind the DBRM and run the program. In particular, you can bind a DBRM at the current release level 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. Of course, you can run applications on the current release that were previously bound on systems at the previous release level.

Using the DB2 coprocessor for C programs

The DB2 coprocessor performs DB2 precompiler functions at compile time. In addition, the DB2 coprocessor lifts some of the DB2 precompiler's restrictions on SQL programs. 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 C program, instead of in only the top-level source file
- Use nested SQL INCLUDE statements

To use the DB2 coprocessor, you need to do the following things:

- Specify the following options when you compile your program:
 - SQL

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. The list is enclosed in single or double quotation marks. Table 64 on page 484 lists the options that you can specify.

For example, suppose that you want to process SQL statements as you compile a C program. In your program, the apostrophe is the string delimiter in SQL statements, and the SQL statements conform to DB2 rules. This means that you need to specify the APOSTSQL and STDSQL(NO) options. Therefore, you need to include this option in your compile step:

```
SQL("APOSTSQL STDSQL(NO)")
```

- LIMITS(FIXEDBIN(63), FIXEDDEC(31))

These options are required for LOB support.

- SIZE(nnnnnnn)

You might need to increase the SIZE value (nnnnnnn) so that the user region is large enough for the DB2 coprocessor. Do not specify SIZE(MAX).


```
#          Do not specify SIZE(MAX).
|
|
|
|
#          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 compile step.
|
|          - DB2 load library (prefix.SDSNLOAD)
|
#          The DB2 coprocessor produces a DBRM. DBRMs and the DBRM library are
#          described in "Output from the precompiler" on page 476. You need to include
#          a DBRMLIB DD statement that specifies the DBRM library data set.
|
|          - DBRM library
|
#          The DB2 coprocessor produces a DBRM. DBRMs and the DBRM library are
#          described in "Output from the precompiler" on page 476. You need to include
#          a DBRMLIB DD statement that specifies the DBRM library data set.
|
|          - Library for SQL INCLUDE statements
|
|          If your program contains EXEC SQL INCLUDE statements other than EXEC
|          SQL INCLUDE SQLCA and EXEC SQL INCLUDE SQLDA, you need to
|          include the SYSLIB DD statement to indicate the include library and the C
|          header files.
|
|          Note: When you use both EXEC SQL INCLUDE and #include statements in a
|          C++ program, the member names that you use for the statements must be
|          unique.
```

Using the DB2 coprocessor for COBOL programs

The DB2 coprocessor performs DB2 precompiler functions at compile time. In addition, the DB2 coprocessor lifts some of the DB2 precompiler's restrictions on SQL programs. 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 COBOL program, instead of in only the top-level source file
- Use nested SQL INCLUDE statements
- Use COBOL REPLACE statements to replace text strings in SQL statements

To use the DB2 coprocessor, you need to do the following things:

- Specify the following options when you compile your program:
 - SQL

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. The list is enclosed in single or double quotes. Table 64 on page 484 lists the options that you can specify.

For example, suppose that you want to process SQL statements as you compile a COBOL program. In your program, the apostrophe is the string delimiter in SQL statements, and the SQL statements conform to DB2 rules. This means that you need to specify the APOSTSQL and STDSQL(NO) options. Therefore, you need to include this option in your compile step:

```
SQL("APOSTSQL STDSQL(NO)")
```

- LIB

You need to specify the LIB option when you specify the SQL option, whether or not you have any COPY, BASIS, or REPLACE statements in your program.

- LIMITS(FIXEDBIN(63), FIXEDDEC(31))

These options are required for LOB support.

- SIZE(*nnnnnnn*)

You might need to increase the SIZE value (*nnnnnn*) so that the user region is
 # large enough for the DB2 coprocessor. Do not specify SIZE(MAX).

- 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 compile step.

- DBRM library

The DB2 coprocessor produces a DBRM. DBRMs and the DBRM library are
 # described in “Output from the precompiler” on page 476. 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 include the name of the data set that contains *member-name* in the SYSLIB concatenation for the compile step.

DB2 coprocessor for PL/I programs

The PL/I compiler that supports the DB2 coprocessor is called the PL/I SQL preprocessor. The PL/I SQL preprocessor performs DB2 precompiler functions at compile time. The DB2 coprocessor lifts some of the DB2 precompiler's restrictions on SQL programs. When you process SQL statements with 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 PL/I program, instead of in only the top-level source file
- Use nested SQL INCLUDE statements

To use the SQL statement preprocessor, you need to do the following things:

- Specify the following options when you compile your program, using the Enterprise PL/I for z/OS and OS/390 Version 3 Release 1 or later:

- PP(SQL(*'option,...'*))

The PP compiler option indicates that you want the compiler to invoke the PLI SQL preprocessor. You can further specify a list of SQL processing options after the SQL suboption PP. The list is enclosed in single or double quotation marks. Separate options in the list by a comma, blank, or both. Table 64 on page 484 lists the options that you can specify.

For example, suppose that you want to process SQL statements as you compile a PL/I program. In your program, the DATE data types require USA format, and the SQL statements conform to DB2 rules. This means that you need to specify the DATE(USA) and STDSQL(NO) options. Therefore, you need to include this option in your compile step:

```
PP(SQL('DATE(USA), STDSQL(NO)'))
```

- LIMITS(FIXEDBIN(63), FIXEDDEC(31))

These options are required for LOB support.

- SIZE(*nnnnnn*)

You might need to increase the SIZE value so that the user region is large enough for the processing of the DB2 coprocessor. Do not specify SIZE(MAX).

- Include DD statements for the following data sets in the JCL for your compile step:

```
#      – DB2 load library (prefix.SDSNLOAD)
#      The PL/I SQL preprocessor calls the DB2 coprocessor APIs to do the SQL
#      statement processing. You therefore need to include the name of the DB2 load
#      library data set in the STEPLIB concatenation for the compile step.
#
#      – DBRM library
#      The DB2 coprocessor produces a DBRM. DBRMs and the DBRM library are
#      described in “Output from the precompiler” on page 476. 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 include the name of the
#      data set that contains member-name in the SYSLIB concatenation for the
#      compile step.
```

Differences between the DB2 precompiler and DB2 coprocessor

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:

DB2 coprocessor:

DB2 uses the source CCSID that is specified by the compiler CODEPAGE option. If this option is not specified and the compiler does not provide a default, DB2 uses the value for the DB2 coprocessor CCSID option. If that option is also not specified, DB2 uses the CCSID in DSNHDECP as the default.

DB2 precompiler:

DB2 uses the value for the DB2 precompiler CCSID option. If that option is not specified, DB2 uses the CCSID in DSNHDECP as the default. Because the DB2 precompiler is processed before the compile step, you must ensure that the CCSID specified during the precompile step is the same as the source CCSID that is specified in the compile step.

- Differences in handling host variable CCSIDs:

- COBOL:

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.

DB2 precompiler:

The DB2 precompiler sets CCSIDs for alphanumeric host variables only when the program includes an explicit `DECLARE :hv VARIABLE` statement.

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'.
#
# '7F0000FFFF000000007F'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.

#
# For more information about these options, see the IBM Enterprise PL/I for
# z/OS Programming Guide.

#
# Options for SQL statement processing
# To control the DB2 precompiler or DB2 coprocessor, you specify options when you
# use it. The options specify how the SQL statement processor interprets or processes
# its input, and how it presents its output.

```

If you are using the DB2 precompiler, you can specify SQL processing options in one of the following ways:

- With DSNH operands
- With the PARM.PC option of the EXEC JCL statement
- In DB2I panels

```
# If you are using the SQL preprocessor, specify the DB2 coprocessor options in the
# following way:
|
# • 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.
```

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 in the APPLICATION PROGRAMMING DEFAULTS installation panels.

```
# Table of SQL processing options: Table 64 shows the options that you can specify
# when you use the DB2 precompiler or the DB2 coprocessor. The table also includes
# abbreviations for those options. Not all options apply to all host languages. For
# information about which options are ignored for a particular host language, see
# Table 64.
```

Table 64 uses a vertical bar (|) to separate mutually exclusive options, and brackets ([]) to indicate that you can sometimes omit the enclosed option.

Table 64. SQL processing options

Option keyword	Meaning
# APOST ¹	Indicates that the DB2 precompiler is to use the apostrophe (') as the string delimiter in host language statements that it generates.
#	
#	This option is not available in all languages; see Table 66 on page 492.
#	
#	APOST and QUOTE are mutually exclusive options. The default is in the field STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If STRING DELIMITER is the apostrophe ('), APOST is the default.
# APOSTSQL	Recognizes the apostrophe (') as the string delimiter and the quotation mark (") as the SQL escape character <i>within SQL statements</i> .
#	
#	APOSTSQL and QUOTESQL are mutually exclusive options. The default is in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If SQL STRING DELIMITER is the apostrophe ('), APOSTSQL is the default.
#	
ATTACH(TSO CAF RRSAF)	Specifies the attachment facility that the application uses to access DB2. TSO, CAF, and RRSAF applications that load the attachment facility can use this option to specify the correct attachment facility, instead of coding a dummy DSNHLI entry point.
	This option is not available for Fortran applications.
	The default is ATTACH(TSO).

Table 64. SQL processing options (continued)

Option keyword	Meaning
CCSID(<i>n</i>) # #	Specifies the numeric value <i>n</i> of the CCSID in which the source program is written. The number <i>n</i> must be 65535 or in the range 1 through 65533, and must be an EBCDIC CCSID.
 	The default setting is the EBCDIC system CCSID as specified on the panel DSN TIPF during installation.
# #	The DB2 coprocessor uses the following process to determine the CCSID of the source statements:
# # #	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:
#	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 that 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 is used.
#	d. If the CCSID suboption of the SQL option is specified and contains an invalid CCSID, compilation terminates.
# #	If you specify CCSID(1026) or CCSID(1155), the DB2 coprocessor does not support the code point 'FC'X for the double quotation mark (").
COMMA	Recognizes the comma (,) as the decimal point indicator in decimal or floating point literals in the following cases: • 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. COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on Application Programming Defaults Panel 1 when DB2 is installed.
CONNECT(2 1) CT™(2 1)	Determines whether to apply type 1 or type 2 CONNECT statement rules. CONNECT(2) Default: Apply rules for the CONNECT (Type 2) statement. CONNECT(1) Apply rules for the CONNECT (Type 1) statement If you do not specify the CONNECT option when you precompile a program, the rules of the CONNECT (Type 2) statement apply. See "Precompiler options for DRDA access" on page 445 for more information about this option, and Chapter 5 of <i>DB2 SQL Reference</i> for a comparison of CONNECT (Type 1) and CONNECT (Type 2).

Table 64. SQL processing options (continued)

Option keyword	Meaning
DATE(ISO USA EUR JIS LOCAL)	Specifies that date output should always return in a particular format, regardless of the format that is specified as the location default. For a description of these formats, see Chapter 2 of <i>DB2 SQL Reference</i>. The default is specified in the field DATE FORMAT on Application Programming Defaults Panel 2 when DB2 is installed. 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. You cannot use the LOCAL option unless you have a date exit routine.
DEC(15 31) D(15.s 31.s)	Specifies the maximum precision for decimal arithmetic operations. See “Using 15-digit and 31-digit precision for decimal numbers” on page 16. The default is in the field DECIMAL ARITHMETIC on Application Programming Defaults Panel 1 when DB2 is installed. 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.
FLAG(I W E S) ¹	Suppresses diagnostic messages below the specified severity level (Informational, Warning, Error, and Severe error for severity codes 0, 4, 8, and 12 respectively). The default setting is FLAG(I).
FLOAT(S390 IEEE)	Determines whether the contents of floating-point host variables in assembler, C, C++, or PL/I programs are in IEEE floating-point format or System/390 floating-point format. DB2 ignores this option if the value of HOST is anything other than ASM, C, CPP, or PLI. The default setting is FLOAT(S390).
GRAPHIC	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. GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is chosen under MIXED DATA on Application Programming Defaults Panel 1 when DB2 is installed.

Table 64. SQL processing options (continued)

Option keyword	Meaning
HOST ¹ (ASM C[(FOLD)] CPP[(FOLD)] IBMCOB PLI FORTRAN)	Defines the host language containing the SQL statements. Use IBMCOB for Enterprise COBOL for z/OS and OS/390. If you specify COBOL or COB2, a warning message is issued and the precompiler uses IBMCOB. For C, specify: • 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 For C++, specify: • 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 If you omit the HOST option, the DB2 precompiler issues a level-4 diagnostic message and uses the default value for this option. The default is in the field LANGUAGE DEFAULT on Application Programming Defaults Panel 1 when DB2 is installed. This option also sets the language-dependent defaults; see Table 66 on page 492.
LEVEL[(aaaa)] L	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. For more information, see “Setting the program level” on page 502. 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.
LINECOUNT ¹ (<i>n</i>) LC	Defines the number of lines per page to be <i>n</i> for the DB2 precompiler listing. This includes header lines inserted by the DB2 precompiler. The default setting is LINECOUNT(60).
MARGINS ¹ (<i>m,n[,c]</i>) MAR	Specifies what part of each source record contains host language or SQL statements; and, for assembler, 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 for assembler where 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. Default values depend on the HOST option you specify; see Table 66 on page 492. 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.

Table 64. SQL processing options (continued)

Option keyword	Meaning
NEWFUN(YES NO)	Indicates whether to accept the syntax for DB2 Version 8 functions. NEWFUN(YES) causes the precompiler to accept DB2 Version 8 syntax. A successful precompilation produces a DBRM that can be bound only with Version 8 and later releases, even if the DBRM does not use any Version 8 syntax. NEWFUN(NO) causes the precompiler to reject any syntax that DB2 Version 8 introduces. A successful precompilation produces a DBRM that can be bound with any release of DB2, including DB2 Version 8. During migration of DB2 Version 8 from an earlier release, the default is NO. At the end of enabling-new-function mode, the default changes from NO to YES. If Version 8 is a new installation of DB2, the default is YES. For information about enabling-new-function mode during installation, see the <i>DB2 Installation Guide</i> .
NOFOR	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. 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. Whether you use or do not use NOFOR, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns named in the clause and specify the acquisition of update locks. You imply NOFOR when you use the option STDSQL(YES). 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.
NOGRAPHIC	Indicates the use of X'0E' and X'0F' in a string, but not as control characters. GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is chosen under MIXED DATA on Application Programming Defaults Panel 1 when DB2 is installed.
NOOPTIONS ² NOOPTN	Suppresses the DB2 precompiler options listing.
NOPADNTSTR ³	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. PADNTSTR and NOPADNTSTR are mutually exclusive options. The default (PADNTSTR or NOPADNTSTR) is chosen under PAD NUL-TERMINATED on Application Programming Defaults Panel 2 when DB2 is installed.
NOSOURCE ² NOS	Suppresses the DB2 precompiler source listing. This is the default.
NOXREF ² NOX	Suppresses the DB2 precompiler cross-reference listing. This is the default.
ONEPASS ¹ 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; see Table 66 on page 492. ONEPASS and TWOPASS are mutually exclusive options.

Table 64. SQL processing options (continued)

Option keyword	Meaning
OPTIONS ¹ OPTN	Lists DB2 precompiler options. This is the default.
# PADNTSTR ³ # 	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 chosen under PAD NUL-TERMINATED on Application Programming Defaults Panel 2 when DB2 is installed.
PERIOD	Recognizes the period (.) as the decimal point indicator in decimal or floating point literals in the following cases: • 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. COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on Application Programming Defaults Panel 1 when DB2 is installed.
# 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: • CCSID(1026) and HOST(IBMCOB) • CCSID(1155) and HOST(IBMCOB) The default is in the field STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If STRING DELIMITER is the quote (") or DEFAULT, then QUOTE is the default.
#	APOST and QUOTE are mutually exclusive options.
QUOTESQL	Recognizes the quotation mark (") as the string delimiter and the apostrophe (') as the SQL escape character <i>within SQL statements</i> . This option applies only to COBOL. The default is in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 when DB2 is installed. If SQL STRING DELIMITER is the quote (") or DEFAULT, QUOTESQL is the default. APOSTSQL and QUOTESQL are mutually exclusive options.
SOURCE ¹ S	Lists DB2 precompiler source and diagnostics.

Table 64. SQL processing options (continued)

Option keyword	Meaning
SQL(ALL DB2)	Indicates whether the source contains SQL statements other than those recognized by DB2 UDB for z/OS. SQL(ALL) is recommended for application programs whose SQL statements must execute on a server other than DB2 UDB for z/OS using DRDA access. SQL(ALL) indicates that the SQL statements in the program are not necessarily for DB2 UDB 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. SQL(DB2), the default, means to interpret SQL statements and check syntax for use by DB2 UDB for z/OS. SQL(DB2) is recommended when the database server is DB2 UDB for z/OS.
# SQLFLAG ¹ (IBM STD [(ssname [,qualifier])])	This option is no longer supported.
STDSQL(NO YES) ⁴	Indicates to which rules the output statements should conform. 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. The default is in the field STD SQL LANGUAGE on Application Programming Defaults Panel 2 when DB2 is installed. STDSQL(YES) automatically implies the NOFOR option.
TIME(ISO USA EUR JIS LOCAL)	Specifies that time output always return in a particular format, regardless of the format that is specified as the location default. For a description of these formats, see Chapter 2 of <i>DB2 SQL Reference</i>. The default is specified in the field TIME FORMAT on Application Programming Defaults Panel 2 when DB2 is installed. 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. You cannot use the LOCAL option unless you have a time exit routine.
TWOPASS ¹ TW	Processes in two passes, so that declarations need not precede references. Default values depend on the HOST option specified; see Table 66 on page 492. ONEPASS and TWOPASS are mutually exclusive options.

Table 64. SQL processing options (continued)

Option keyword	Meaning
VERSION(<i>aaaa</i> AUTO)	Defines the version identifier of a package, program, and the resulting DBRM. 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 plan or package. If you do not specify a version at precompile time, then 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 used as the version identifier. The timestamp used is based on the System/370™ Store Clock value. For information about using VERSION, see “Identifying a package version” on page 502. # For more information about the rules for version identifiers, see in <i>DB2 SQL Reference</i> . #
XREF ¹	Includes a sorted cross-reference listing of symbols used in SQL statements in the listing output.

Notes:

- # 1. This option is ignored when the DB2 coprocessor precompiles the application.
- # 2. This option is always in effect when the DB2 coprocessor precompiles the application.
- 3. This option applies only for a C or C++ application.
- # 4. You can use STDSQL(86) as in prior releases of DB2. The DB2 processor treats it the same as STDSQL(YES).
- 5. Precompiler options do not affect ODBC behavior.

Defaults for options of the SQL statement processor: Some SQL statement processor options have defaults based on values specified on the Application Programming Defaults panels. Table 65 shows those options and defaults:

Table 65. 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	APOST QUOTE
SQL STRING DELIMITER	quotation mark (")	QUOTESQL	APOSTSQL QUOTESQL
DECIMAL POINT IS	PERIOD	PERIOD	COMMA PERIOD
DATE FORMAT	ISO	DATE(ISO)	DATE(ISO USA EUR JIS LOCAL)
DECIMAL ARITHMETIC	DEC15	DEC(15)	DEC(15 31)
MIXED DATA	NO	NOGRAPHIC	GRAPHIC NOGRAPHIC
LANGUAGE DEFAULT	COBOL	HOST(COBOL)	HOST(ASM C[(FOLD)] CPP[(FOLD)] IBMCOB FORTRAN PLI)
STD SQL LANGUAGE	NO	STDSQL(NO)	STDSQL(YES NO 86)

Table 65. 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
TIME FORMAT	ISO	TIME(ISO)	TIME(IS USA EUR JIS LOCAL)

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 install options:

- STRING DELIMITER
- SQL STRING DELIMITER
- DECIMAL POINT IS
- DECIMAL ARITHMETIC
- MIXED DATA

If the value of USE FOR DYNAMICRULES is YES, then dynamic SQL statements use the application programming defaults. If the value of USE FOR DYNAMICRULES is NO, then dynamic SQL statements in packages or plans with bind, define, and invoke behavior use the SQL statement processor options. See “Using DYNAMICRULES to specify behavior of dynamic SQL statements” on page 502 for an explanation of bind, define, and invoke behavior.

Some SQL statement processor options have default values based on the host language. Some options do not apply to some languages. Table 66 shows the language-dependent options and defaults.

Table 66. 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)

Notes:

1. Forced for this language; no alternative allowed.
2. The default is chosen on Application Programming Defaults Panel 1 when DB2 is installed. 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 you specify override any defaults 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 DSNHDECP parameter DYNRULS is NO, then 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
- GRAPHIC or NOGRAPHIC
- DEC(15) or DEC(31)

Translating command-level statements in a CICS program

CICS

Translating command-level statements: 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; no translator is available for Fortran.) 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 having 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 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 precompiler option NOGRAPHIC 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
```

CICS (continued)

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.

You can append JCL from a job created by the DB2 Program Preparation panels to the CICS translator JCL to prepare an application program. 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. For information on the required resource entries, see Part 2 of *DB2 Installation Guide* and *CICS Transaction Server for z/OS Resource Definition Guide*.

prefix.SDSNSAMP contains examples of the JCL used to prepare and run a CICS program that includes SQL statements. For a list of CICS program names and JCL member names, see Table 192 on page 1018. The set of JCL includes:

- PL/I macro phase
- DB2 precompiling
- CICS Command Language Translation
- Compiling the host language source statements
- Link-editing the compiler output
- Binding the DBRM
- Running the prepared application.

Step 2: Compile (or assemble) and link-edit the 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:

- 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. You must use JCL procedures when you use 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 these requirements:

TSO and batch

Include the DB2 TSO attachment facility language interface module (DSNELI) or DB2 call attachment facility language interface module (DSNALI).

For a program that uses 31-bit addressing, link-edit the program with the AMODE=31 and RMODE=ANY options.

For more details, see the appropriate z/OS publication.

IMS

Include the DB2 IMS (Version 1 Release 3 or later) language interface module (DFSLLI000). Also, the IMS RESLIB must precede the SDSNLOAD library in the link list, JOBLIB, or STEPLIB concatenations.

CICS

Include the DB2 CICS language interface module (DSNCLI).

You can link DSNCLI with your program in either 24 bit or 31 bit addressing mode (AMODE=31). If your application program runs in 31-bit addressing mode, you should link-edit the DSNCLI stub to your application with the attributes AMODE=31 and RMODE=ANY so that your application can run above the 16M line. For more information on compiling and link-editing CICS application programs, see the appropriate CICS manual.

You also need the CICS EXEC interface module 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.

For more information about compiling and link-editing, see “Using JCL procedures to prepare applications” on page 512.

For more information about link-editing attributes, see the appropriate z/OS manuals. For details on DSNH, see Part 3 of *DB2 Command Reference*.

Step 3: Bind the application

You must bind the DBRM produced by the SQL statement processor to a plan or package before your DB2 application can run. A plan can contain DBRMs, a package list specifying packages or collections of packages, or a combination of DBRMs and a package list. The plan must contain at least one package or at least one directly-bound DBRM. Each package 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, whether or not they reference packages that run remotely. However, you must bind the packages that run at remote locations at those remote locations.

From a DB2 requester, you can run a plan by naming 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.

Binding a DBRM to a package

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 DRDA access, you must bind packages on the systems on which they will run. If a local stored procedure uses a cursor to access data through DRDA access, 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 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 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 you use as resolving to a remote location. To bind the DBRM PROGA at the location PARIS, in the collection GROUP1, use:

```
BIND PACKAGE(PARIS.GROUP1)
      MEMBER(PROGA)
```

Then, include the remote package in the package list of a local plan, say PLANB, by using:

```
BIND PLAN (PLANB)
      PKLIST(PARIS.GROUP1.PROGA)
```

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) and you do not specify the option SQLERROR(CONTINUE), then PLAN_TABLE must exist at the location 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 related to the copied package at the local site. If the local site is installed with time or date format LOCAL, and a package is created at a remote site using the COPY option, the COPY option causes DB2 at the remote site to convert values returned from the remote site in ISO format, unless an SQL statement specifies 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.

Turning an existing plan into packages to run remotely: If you have used DB2 before, you might have an existing application that you want to run at a remote location, using DRDA access. To do that, you need to rebind the DBRMs in the current plan as packages at the remote location. You also need a new plan that includes those remote packages in its package list.

Follow these instructions for each remote location:

1. Choose a name for a collection to contain all the packages in the plan, say REMOTE1. (You can use more than one collection if you like, but one is enough.)
2. Assuming that the server is a DB2 system, at the remote location execute:
 - a. GRANT CREATE IN COLLECTION REMOTE1 TO *authorization-name*;
 - b. GRANT BINDADD TO *authorization-name*;where *authorization-name* is the owner of the package.
3. Bind *each* DBRM as a package at the remote location, using the instructions under “Binding packages at a remote location” on page 496. Before run time, the package owner must have all the data access privileges needed at the remote location. If the owner does not yet have those privileges when you are binding, use the VALIDATE(RUN) option. The option lets you create the package, even if the authorization checks fail. DB2 checks the privileges again at run time.
4. Bind a new application plan at your local DB2, using these options:

```
PKLIST (location-name.REMOTE1.*)
CURRENTSERVER (location-name)
```

where *location-name* is the value of LOCATION, in SYSIBM.LOCATIONS at your local DB2, that denotes the remote location at which you intend to run. You do not need to bind any DBRMs directly to that plan: the package list is sufficient.

When you now run the existing application at your local DB2, using the new application plan, these things happen:

- You connect immediately to the remote location named in the CURRENTSERVER option.

- When about to run a package, DB2 searches for it 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 return to your existing application program, which processes them as though they came from your local DB2.

Binding an application plan

Use the BIND PLAN subcommand to bind DBRMs and package lists to a plan. For BIND PLAN syntax and complete descriptions, see Part 3 of *DB2 Command Reference*.

Binding DBRMs directly to a plan: A plan can contain DBRMs bound directly to it. To bind three DBRMs—PROGA, PROGB, and PROGC—directly to plan PLANW, use:

```
BIND PLAN(PLANW)
      MEMBER(PROGA,PROGB,PROGC)
```

You can include as many DBRMs in a plan as you wish. However, if you use a large number of DBRMs in a plan (more than 500, for example), you could have trouble maintaining the plan. To ease maintenance, you can bind each DBRM separately as a package, specifying the same collection for all packages bound, and then bind a plan specifying that collection in the plan's package list. If the design of the application prevents this method, see if your system administrator can increase the size of the EDM pool to be at least 10 times the size of either the largest database descriptor (DBD) or the plan, whichever is greater.

Including packages in a package list: To include packages in the package list of a plan, list them after the PKLIST keyword of BIND PLAN. To include an entire collection of packages in the list, use an asterisk after the collection name. For example,

```
PKLIST(GROUP1.*)
```

To bind DBRMs directly to the plan, and also include packages in the package list, use both MEMBER and PKLIST. The following example includes:

- The DBRMs PROG1 and PROG2
- All the packages in a collection called TEST2
- The packages PROGA and PROGC in the collection GROUP1

```
MEMBER(PROG1,PROG2)
PKLIST(TEST2.*,GROUP1.PROGA,GROUP1.PROGC)
```

You must specify MEMBER, PKLIST, or both options. The plan that results consists of one of the following:

- Programs associated with DBRMs in the MEMBER list only
- Programs associated with packages and collections identified in PKLIST only
- A combination of the specifications on MEMBER and PKLIST

Identifying 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 plan or 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: see “Setting the program level” on page 502.)

#

You also need other identifiers. The consistency token alone uniquely identifies a DBRM bound directly to a plan, but it does not necessarily identify a unique package. When you bind DBRMs directly to a particular plan, you bind each one only once. But you can bind the same DBRM to many packages, at different locations and in different collections, and then you can include all those packages in the package list of the same plan. All those packages will have the same consistency token. As you might expect, there are ways to specify a particular location or a particular collection at run time.

Identifying the location: When your program executes an SQL statement, DB2 uses the value in the CURRENT SERVER special register to determine the location of the necessary package or DBRM. If the current server is your local DB2 subsystem and it does not have a location name, the value 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 has no location name).

Identifying the collection: 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 are blank when your application begins to run and remain blank. In this case, DB2 searches the available collections, using methods described in “Specifying the package list for the PKLIST option of BIND PLAN.”

However, explicitly specifying the intended collection by using the special registers can avoid a potentially costly search through a package list with many qualifying entries. In addition, DB2 uses the values in these special registers for applications that do not run under a plan. How DB2 uses these special registers is described in “Using the special registers” on page 500.

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. When the stored procedure returns control to the calling program, DB2 restores this register to the value that it contained before the call.

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 are blank: 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 names that location explicitly), the order is:
 1. All DBRMs that are bound directly to the plan.
 2. All packages that are already allocated to the plan while the plan is running.
 3. All unallocated packages that are explicitly named 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 named 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. See “Using bind options to improve performance for distributed applications” on page 456 for more information.

If the order of search is not important: In many cases, DB2’s order of search is not important to you and does not affect performance. For an application that runs only at your local DB2, 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.

Using the special registers: 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, then 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, then 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.

Table 67 shows examples of the relationship between the CURRENT PACKAGE PATH special register and the CURRENT PACKAGESET special register.

Table 67. 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.

Identifying a package version: Sometimes, however, you want to have more than one package with the same name available to your plan. The VERSION option makes that possible. Using VERSION identifies your program with a specific version of a package. If you bind the plan with PKLIST (COLLECT.*), then you can do this:

Step number	For Version 1	For Version 2
1	Precompile program 1, using VERSION(1).	Precompile program 2, using VERSION(2).
2	Bind the DBRM with the collection name COLLECT and your chosen package name (say, 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.

You can do that with many versions of the program, without having to rebind the application plan. Neither do you have to rename the plan or change any RUN subcommands that use it.

Setting the program level: To override DB2's construction of the consistency token, use the LEVEL (aaaa) option. DB2 uses the value you choose for aaaa to generate the consistency token. Although we do not recommend this method for general use and the DSNH CLIST or the DB2 Program Preparation panels do not support it, it allows you to do the following:

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.

Using BIND and REBIND options for packages and plans

This section discusses a few of the more complex bind and rebind options. For syntax and complete descriptions of all of the bind and rebind options, see Part 3 of *DB2 Command Reference*.

Using DYNAMICRULES to specify behavior of dynamic SQL statements: 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 to the DYNAMICRULES value, 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 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.

The combination of the DYNAMICRULES value and the run-time environment determine the values for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement *behavior*. The four behaviors are:

- Run behavior
- Bind behavior
- Define behavior
- Invoke behavior

Table 68 shows the combination of DYNAMICRULES value and run-time environment that yield each dynamic SQL behavior.

Table 68. 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.		

Table 69 shows the dynamic SQL attribute values for each type of dynamic SQL behavior.

Table 69. 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 parameter DYNRULS ³	Install panel DSNTIP4	Determined by DSNHDECP parameter DYNRULS ³	Determined by DSNHDECP parameter DYNRULS ³

Table 69. Definitions of dynamic SQL statement behaviors (continued)

Dynamic SQL attribute	Setting for dynamic SQL attributes			
	Bind behavior	Run behavior	Define behavior	Invoke behavior
Can execute GRANT, REVOKE, CREATE, ALTER, DROP, RENAME?	No	Yes	No	No

Notes:

1. If the invoker is the primary authorization ID of the process or the CURRENT SQLID value, secondary authorization IDs will also be 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. See Table 41 on page 342 and Table 79 on page 647 for information about initialization of CURRENT SQLID for user-defined functions and stored procedures.

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 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 483 for more information.

For more information about DYNAMICRULES, see Chapter 2 of *DB2 SQL Reference* and Part 3 of *DB2 Command Reference*.

Determining the optimal authorization cache size: When DB2 determines that you have the EXECUTE privilege on a plan, package collection, stored procedure, or user-defined function, DB2 can cache your authorization ID. When you run the plan, package, stored procedure, or user-defined function, DB2 can check your authorization more quickly.

Determining the authorization cache size for plans: The CACHESIZE option (optional) allows you to specify the size of the cache to acquire for the plan. DB2 uses this cache for caching the authorization IDs of those users that are running a plan. An authorization ID can take up to 128 bytes of storage. DB2 uses the CACHESIZE value to determine the amount of storage to acquire for the authorization cache. DB2 acquires storage from the EDM storage pool. The default CACHESIZE value is 1024 or the size set at installation time.

The size of the cache you specify depends on the number of individual authorization IDs actively using the plan. Required overhead takes 32 bytes, and each authorization ID takes up 8 bytes of storage. The minimum cache size is 256 bytes (enough for 28 entries and overhead information) and the maximum is 4096 bytes (enough for 508 entries and overhead information). You should specify size in multiples of 256 bytes; otherwise, the specified value rounds up to the next highest value that is a multiple of 256.

If you run the plan infrequently, or if authority to run the plan is granted to PUBLIC, you might want to turn off caching for the plan so that DB2 does not use unnecessary storage. To do this, specify a value of 0 for the CACHESIZE option.

Any plan that you run repeatedly is a good candidate for tuning using the CACHESIZE option. Also, if you have a plan that a large number of users run concurrently, you might want to use a larger CACHESIZE.

Determining the authorization cache size for packages: DB2 provides a single package authorization cache for an entire DB2 subsystem. The DB2 installer sets the size of the package authorization cache by entering a size in field PACKAGE AUTH CACHE of DB2 installation panel DSNTIPP. A 32-KB authorization cache is large enough to hold authorization information for about 375 package collections.

See *DB2 Installation Guide* for more information about setting the size of the package authorization cache.

Determining the authorization cache size for stored procedures and user-defined functions: DB2 provides a single routine authorization cache for an entire DB2 subsystem. The routine authorization cache stores a list of authorization IDs that have the EXECUTE privilege on user-defined functions or stored procedures. The DB2 installer sets the size of the routine authorization cache by entering a size in field ROUTINE AUTH CACHE of DB2 installation panel DSNTIPP. A 32-KB authorization cache is large enough to hold authorization information for about 380 stored procedures or user-defined functions.

See *DB2 Installation Guide* for more information about setting the size of the routine authorization cache.

Specifying the SQL rules: 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. When the server is not the local DB2, the initial value of CURRENT RULES is DB2. After binding a plan, you can change the value in CURRENT RULES in an application program 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 using the statement ALTER TABLE 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 and place the table in check pending status, and restore the rules to STD.

```
EXEC SQL
    SET CURRENT RULES = 'DB2';
EXEC SQL
```

```
ALTER TABLE DSN8810.EMP  
  ADD CONSTRAINT C1 CHECK (BONUS <= 1000.0);  
EXEC SQL  
  SET CURRENT RULES = 'STD';
```

See “Using check constraints” on page 259 for information about check constraints.

You can also use CURRENT RULES in host variable assignments, for example:

```
SET :XRULE = CURRENT RULES;
```

You can also use CURRENT RULES as the argument of a search-condition, for example:

```
SELECT * FROM SAMPTBL WHERE COL1 = CURRENT RULES;
```

Using packages with dynamic plan selection

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.

The benefit of using dynamic plan selection and packages together is that you can convert individual programs in an application containing many programs and plans, one at a time, to use a combination of plans and packages. This reduces the number of plans per application, and having fewer plans reduces the effort needed to maintain the dynamic plan exit.

Assume that you develop the following programs and DBRMs:

Program Name	DBRM Name
MAIN	MAIN
PROGA	
	PLANA
PROGB	
	PKGB
PROGC	
	PLANC

You could create packages and plans using the following bind statements:

```
BIND PACKAGE(PKGB) MEMBER(PKGB)
BIND PLAN(MAIN) MEMBER(MAIN,PLANA) PKLIST(*.PKGB.*)
BIND PLAN(PLANC) MEMBER(PLANC)
```

The following scenario illustrates thread association for a task that runs program MAIN:

Sequence of SQL Statements

Events

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 selects plan MAIN.
3. EXEC CICS LINK PROGRAM(PROGA)
4. EXEC SQL SELECT...
DB2 does not call the default dynamic plan exit, because the program does not issue a sync point. The plan is MAIN.

CICS (continued)

Sequence of SQL Statements

Events

5. EXEC CICS LINK PROGRAM(PROGB)
6. EXEC SQL SELECT...
DB2 does not call the default dynamic plan exit, 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 when the next SQL statement executes.
8. EXEC CICS LINK PROGRAM(PROGC)
9. EXEC SQL SELECT...
DB2 calls the default dynamic plan exit and selects PLANC.
10. EXEC SQL SET CURRENT SQLID = 'ABC'
11. EXEC CICS SYNCPOINT
DB2 does not call the dynamic plan exit 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...
SQLCODE -815 occurs because the plan is currently PLANC and the program is PROGB.

Step 4: Run the application

After you complete all the previous steps, you are ready to run your application. At this time, DB2 verifies that the information in the application plan and its associated packages is consistent with the corresponding information in the DB2 system 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.

DSN command processor

The DSN command processor is a TSO command processor that runs in TSO foreground or under TSO in JES-initiated batch. 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.

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 two task levels below that of 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 Chapter 30, “Programming for the call attachment facility,” on page 861 and Chapter 31, “Programming for the Resource Recovery Services attachment facility,” on page 893.

DSN return code processing: At the end of a DSN session, register 15 contains the highest value placed there by any DSN subcommand used in the session or by any program 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.

Running a program in TSO foreground

Use the DB2I RUN panel to run a program in TSO foreground. As an alternative to the RUN panel, you can issue the DSN command followed by the RUN subcommand of DSN. Before running the program, be sure to allocate any data sets 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) -
              PARM('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 PARMS keyword of the RUN subcommand allows 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 PARMS(*'parameters/run-time-options'*). In those environments, an example of the PARMS keyword might be:

```
PARMS ('D01, D02, D03/')
```

Check your host language publications for the correct form of the PARMS option.

Running a batch DB2 application in TSO

Most application programs written for the batch environment run under the TSO Terminal Monitor Program (TMP) in background mode. Figure 151 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) -
  PARMS ('/D01 D02 D03')
END
/*
```

Figure 151. JCL for running a DB2 application under the TSO terminal monitor program

- 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 the default application programming defaults module, DSNHDECP, 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 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 the application should access.
- The PARMS 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.
- We recommend that you 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 name instead of a specific *ssid* to connect to a member of a data sharing group. For more information, see *DB2 Data Sharing: Planning and Administration*.

For more information about using the TSO TMP in batch mode, see *z/OS TSO/E User's Guide*.

Calling applications in a command procedure (CLIST)

As an alternative to the previously described foreground or batch calls to an application, you can also run a TSO or batch application using a command procedure (CLIST).

The following CLIST calls a DB2 application program named MYPROG. The DB2 subsystem name or group attachment name should replace *ssid*.

```
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, be sure you can respond to the program's interactive requests for data and that you can recognize the expected results. Then, enter the transaction code associated with the program. Users of the transaction code must be authorized to run the program.

To run a non-message-driven program

Submit the job control statements needed to run the 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; see Part 3 (Volume 1) of *DB2 Administration Guide* for more information.

Also, be sure to define to CICS the transaction code 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.

Running a DB2 REXX application

You run DB2 REXX procedures under TSO. You do not precompile, compile, link-edit or bind DB2 REXX procedures before you run them.

In a batch environment, you might use statements like these to invoke procedure 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.

Using JCL procedures to prepare applications

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. See “Using ISPF and DB2 Interactive” on page 518.
- 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 install time.

This section describes how to use JCL procedures to prepare a program. For information about using the DSNH CLIST, the TSO DSN command processor, or JCL procedures added to your SYS1.PROCLIB, see Part 3 of *DB2 Command Reference*.

Available JCL procedures

You can precompile and prepare an application program using a DB2-supplied procedure. DB2 has a unique 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 70. Procedures for precompiling programs

Language	Procedure	Invocation included in...
High-level assembler	DSNHASM	DSNTEJ2A
C	DSNHC	DSNTEJ2D
C++	DSNHCPP DSNHCPP2 ²	DSNTEJ2E N/A
Enterprise COBOL for z/OS	DSNHICOB	DSNTEJ2C ¹
Fortran	DSNHFOR	DSNTEJ2F
PL/I	DSNHPLI	DSNTEJ2P
SQL	DSNHSQL	DSNTEJ63

Notes:

1. You must customize these programs to invoke the procedures listed in this table. For information about how to do that, see Part 2 of *DB2 Installation Guide*.
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.

Including code from SYSLIB data sets

To include the proper interface code when you submit the JCL procedures, use one of the following sets of statements shown in your JCL. Alternatively, if you are using the call attachment facility, follow the instructions given in “Accessing the CAF language interface” on page 867.

TSO, batch, and CAF

```
//LKED.SYSIN DD *  
  INCLUDE SYSLIB(member)  
/*
```

member must be DSNELI, except for FORTRAN, in which case *member* must be DSNHFT.

IMS

```
//LKED.SYSIN DD *  
  INCLUDE SYSLIB(DFSLLI000)  
  ENTRY (specification)  
/*
```

DFSLLI000 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

Your program's name, for assembler language applications.

CICS

```
//LKED.SYSIN DD *  
  INCLUDE SYSLIB(DSNCLT)  
/*
```

For more information on required CICS modules, see "Step 2: Compile (or assemble) and link-edit the application" on page 494.

Starting the precompiler dynamically

You can call the precompiler from an assembler program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL. The following information supplements the description of these macro instructions given in *z/OS MVS Programming: Assembler Services Reference, Volumes 1 and 2*.

To call the precompiler, specify DSNHPC as the entry point name. You can pass three address options to the precompiler; the following sections describe their formats. The options are addresses of:

- A precompiler option list
- A list of alternate ddnames for the data sets that the precompiler uses
- A page number to use for the first page of the compiler listing on SYSPRINT.

Precompiler option list 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.

DDNAME list format

The ddname 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.

Table 71 gives the following sequence of entries:

Table 71. DDNAME list entries

Entry	Standard ddname	Usage
1	Not applicable	

Table 71. DDNAME list entries (continued)

Entry	Standard ddname	Usage
2	Not applicable	
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

A 6-byte field beginning on a 2-byte boundary contains the page number. The first two 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 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.

An alternative method for preparing a CICS program

CICS

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 do the following:

- 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(I),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) //*
```

CICS (continued)

```
//*****
//***   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) //SYSTSPRT DD SYSOUT=*
(2) //SYSUDUMP DD SYSOUT=*
(2) //SYSTSIN 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
(6) //LKED.SYSIN DD *
(7) INCLUDE CICSLOAD(DSNCLI)
    NAME TESTC01(R)
//*****
```

The procedure accounts for these steps:

Step 1. Precompile the program.

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 you need to consider.

Step 4. The output of the DB2 precompiler becomes the input to the CICS command language translator.

Step 5. 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 6. Name the CICS load library that contains the module DSNCLI.

Step 7. 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 one of the last two techniques mentioned previously. 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. Programs using the call attachment facility can use either of the last two techniques mentioned previously. **Be careful to use the correct language interface.**

Using JCL to prepare a program with object-oriented extensions

If your C++ or Enterprise COBOL for z/OS and OS/390 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
# DSN810.SDSNSAMP, shows you one way to do this for C++.
```

Using ISPF and DB2 Interactive

If you develop programs using TSO and ISPF, you can prepare them to run 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.

This topic describes the panels that are associated with DB2 Program Preparation.

Important: If your C++ or IBM COBOL for z/OS 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.

See “Using JCL to prepare a program with object-oriented extensions” on page 517 for more information.

```
# You can use the TSO PROFILE command to control whether message IDs are
# displayed. To view message IDs, type TSO PROFILE MSGID on the ISPF command
# line. To suppress message IDs, type TSO PROFILE NOMSGID.
```

DB2I help

```
| The online help facility enables you to read information about how to use DB2I in
| an online DB2 book from a DB2I panel. It contains detailed information about the
| fields of each of the DB2 Program Preparation panels.
```

For instructions on setting up DB2 online help, see the discussion of setting up DB2 online help in Part 2 of *DB2 Installation Guide*.

If your site makes use of CD-ROM updates, you can make the updated books accessible from DB2I. Select Option 10 on the DB2I Defaults panel and enter the new book data set names. You must have write access to *prefix.SDSNCLST* to perform this function.

```
| To access DB2I HELP, press the PF key that is associated with the HELP function.
| The default PF key for HELP is PF 1; however, your location might have assigned
| a different PF key for HELP.
```

DB2I Primary Option Menu

Figure 152 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 corresponding 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 152. 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 Chapter 5, “Using SPUFI to execute SQL from your workstation,” on page 59.

2 DCLGEN

Lets you generate C, COBOL, or PL/I data declarations of tables. For further information, see Chapter 8, “Generating declarations for your tables using DCLGEN,” on page 131.

3 PROGRAM PREPARATION

Lets you prepare and run an application program to run. For more information, see “DB2 Program Preparation panel” on page 521.

4 PRECOMPILE

Lets you convert embedded SQL statements into statements that your host language can process. For further information, see “Precompile panel” on page 528.

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 531.

6 RUN

Lets you run an application program in a TSO or batch environment. For more information, see “The Run panel” on page 555.

7 DB2 COMMANDS

Lets you issue DB2 commands. For more information about DB2 commands, see Part 3 of *DB2 Command Reference*.

8 UTILITIES

Lets you call DB2 utility programs. For more information, see *DB2 Utility Guide and Reference*.

D DB2I DEFAULTS

Lets you set DB2I defaults. See “DB2I Defaults Panel 1” on page 526.

X EXIT

Lets you exit DB2I.

To prepare a new application, beginning with precompilation and working through each of the subsequent preparation steps, begin by selecting the option that corresponds to the Program Preparation panel.

Table 72 describes each of the panels you will need to use to prepare an application. The DB2I help contains detailed descriptions of each panel.

Table 72. DB2I panels used for program preparation

Panel name	Panel description
"DB2 Program Preparation panel" on page 521	The DB2 Program Preparation panel lets you choose specific program preparation functions to perform. For the functions 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" on page 526	DB2I Defaults Panel 1 lets you change many of the system defaults that are set at DB2 installation time.
"DB2I Defaults Panel 2" on page 527	DB2I Defaults Panel 2 lets you change your default job statement and set additional COBOL options.
"Precompile panel" on page 528	The Precompile panel 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 panel" on page 533	The Bind Package panel 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 panel" on page 536	The Bind Plan panel 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.
"The Defaults for Bind or Rebind Package or Plan panels" on page 546	These panels let you change the defaults for BIND or REBIND PACKAGE or PLAN.
"System Connection Types panel" on page 550	The System Connection Types panel 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.
"Panels for entering lists of values" on page 552	These panels are list panels that lets 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.

Table 72. DB2I panels used for program preparation (continued)

Panel name	Panel description
"Program Preparation: Compile, Link, and Run panel" on page 553	This panel lets you perform the last two steps in the program preparation process (compile and link-edit). It 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.

Table 73 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 73. 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 531	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 539	The Rebind Package panel lets you change options when you rebind a package.
"Rebind Trigger Package panel" on page 540	The Rebind Trigger Package panel lets you change options when you rebind a trigger package.
"Rebind Plan panel" on page 542	The Rebind Plan panel lets you change options when you rebind an application plan.
"Free Package panel" on page 544	The Free Package panel lets you change options when you free a package.
"Free Plan panel" on page 545	The Free Plan panel lets you change options when you free an application plan.
"The Run panel" on page 555	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.

DB2 Program Preparation panel

The Program Preparation panel lets you choose whether to perform specific program preparation functions. 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. See page 528.
- Bind a package. The panel for this function lets you bind your program's DBRM to a package (see page 533), and change your defaults for binding the packages (see page 546).
- Bind a plan. The panel for this function lets you create your program's application plan (see page 536), and change your defaults for binding the plans (see page 546).
- Compile, link, and run. The panel for these functions let you control the compiler or assembler and the linkage editor. See page 553.

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 (see page 526), and perform other precompile and prelink functions.

On the DB2 Program Preparation panel, shown in Figure 153, 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, RRSF)	
5	OTHER DSNH OPTIONS	==>			
			(Optional DSNH keywords)		
Select functions:		Display panel?		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			==>	N (Y/N)
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 153. 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 fill in 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, RRSF) 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 RRSF, 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.

For more information about those options, see DSNH in Part 3 of *DB2 Command Reference*.

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. For more information, see “DB2I Defaults Panel 1” on page 526.

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.

For information about PL/I options, see “Program Preparation: Compile, Link, and Run panel” on page 553.

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. For information about the Precompile panel, see “Precompile panel” on page 528.

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. For information about the panel, see “Bind Package panel” on page 533.

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. For information about the panel, see “Bind Plan panel” on page 536.

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.

For information about the panel, see “Program Preparation: Compile, Link, and Run panel” on page 553.

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 compile step and the link-edit step.

For more information about the prelink utility, see *z/OS Language Environment Programming Guide*.

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 compile step.

For information about the panel, see “Program Preparation: Compile, Link, and Run panel” on page 553.

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.

For information on the panel, see “Program Preparation: Compile, Link, and Run panel” on page 553.

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.

DB2I Defaults Panel 1

DB2I Defaults Panel 1 lets you change many of the system defaults set at DB2 install time. Figure 154 shows the fields that affect the processing of the other DB2I panels.

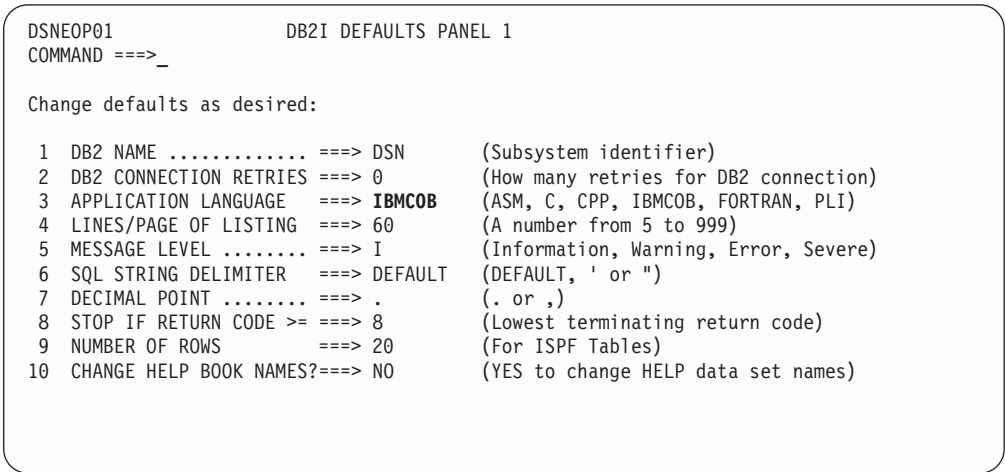


Figure 154. DB2I Defaults Panel 1

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:

ASM

For High Level Assembler/z/OS

C

For C language

CPP

For C++

IBMCOB

For Enterprise COBOL for z/OS and OS/390. 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."

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 CHANGE HELP BOOK NAMES?

Lets you change the name of the BookManager[®] book you reference for online help. The default is NO.

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 154 on page 526 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. Figure 155 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 155. 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 install 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

The next step in the process is to precompile. Figure 152 on page 519, the DB2I Primary Option Menu, shows that you can reach the Precompile panel in two ways: you can either specify 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. Figure 156 on page 529 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 DSNNAME 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 ..... ==>                                (Blank, VERSION, or AUTO)

7 OTHER OPTIONS ..... ==>

```

Figure 156. 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 wish.

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 DSNNAME 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 wish.

If you reached this panel from the DB2I Primary Option Menu, you can either specify a DSNNAME 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 compile 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 DSNNAME 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 example in Figure 156 on page 529, 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 DSNNAME 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 wish.

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:

FOREGROUND 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.

See "Advantages of packages" on page 385 for more information about this option.

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.

For more information about DSNH options, see Part 3 of *DB2 Command Reference*.

Bind/Rebind/Free selection panel

The BIND/REBIND/FREE option on the DB2I PRIMARY OPTIONS MENU (Figure 157 on page 532) lets you 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 157. 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 536.

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 542.

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 545.

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 533.

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 539.

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 540.

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 544.

Bind Package panel

When you request this option, the panel displayed is the first of two Bind Package panels. 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 158 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 ==>

7 -- OPTIONS ==> (Blank, or COPY version-id)
(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 158. 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. For more information, see the BIND PACKAGE command in Part 3 of *DB2 Command Reference*.

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. See “Advantages of packages” on page 385 for more information about this option.

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 “The Defaults for Bind or Rebind Package or Plan panels” on page 546.

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 167 on page 551) 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 implicit qualifier for unqualified tables, views, indexes, and aliases. You can specify a qualifier 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 Bind Plan panels. It specifies options in the bind process of 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. See Figure 159.

DSNEBP02	BIND PLAN	SSID: DSN
COMMAND ==> _		
Enter DBRM data set name(s):		
1 MEMBER	==> SAMPLEPG	
2 PASSWORD	==>	
3 LIBRARY	==> TEMP.DBRM	
4 ADDITIONAL DBRMS?	==> NO	(YES to list more DBRMs)
Enter options as desired:		
5 PLAN NAME	==> SAMPLEPG	(Required to create a plan)
6 CHANGE CURRENT DEFAULTS?	==> NO	(NO or YES)
7 ENABLE/DISABLE CONNECTIONS?	==> NO	(NO or YES)
8 INCLUDE PACKAGE LIST?	==> NO	(NO or YES)
9 OWNER OF PLAN (AUTHID) ...	==>	(Leave blank for your primary ID)
10 QUALIFIER	==>	(For tables, views, and aliases)
11 CACHESIZE	==> 0	(Blank, or value 0-4096)
12 ACTION ON PLAN	==> REPLACE	(REPLACE or ADD)
13 RETAIN EXECUTION AUTHORITY	==> YES	(YES to retain user list)
14 CURRENT SERVER	==>	(Location name)
15 INCLUDE PATH?	==>	(NO or YES)

Figure 159. 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. For more information, see the BIND PLAN command in Part 3 of *DB2 Command Reference*.

1 MEMBER

Lets you specify the DBRMs to include in the plan. You can specify a name from 1 to 8 characters. You must specify MEMBER or INCLUDE PACKAGE LIST, or both. If you do not specify MEMBER, fields 2, 3, and 4 are ignored.

The default member name depends on the input data set.

- If the input data set is partitioned, the default name is the member name of the input data set specified in field 1 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.

If you reached this panel directly from the DB2I Primary Option Menu, you must provide values for the MEMBER and LIBRARY fields.

If you plan to use more than one DBRM, you can include the library name and member name of each DBRM in the MEMBER and LIBRARY fields, separating entries with commas. You can also specify more DBRMs by using the ADDITIONAL DBRMS? field on this panel.

2 PASSWORD

Lets you enter passwords for the libraries you list in the LIBRARY field. You can use this field only if you reached the Bind Plan panel directly from the DB2 Primary Option Menu.

3 LIBRARY

Lets you specify the name of the library or libraries that contain the DBRMs to use for the bind process. You can specify a name up to 44 characters long.

4 ADDITIONAL DBRMS?

Lets you specify more DBRM entries if you need more room. Or, if you reached this panel as part of the program preparation process, you can include more DBRMs by entering YES in this field. A separate panel then displays, where you can enter more DBRM library and member names; see "Panels for entering lists of values" on page 552.

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; for instructions, see "The Defaults for Bind or Rebind Package or Plan panels" on page 546.

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 167 on page 551) 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 INCLUDE PACKAGE LIST?

Lets you include a list of 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 552). This list is optional if you use the MEMBER field.

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. If you
specify a location name, which is optional, it must be in the catalog table
SYSIBM.LOCATIONS; the default location is the local DBMS.

You must specify INCLUDE PACKAGE LIST? or MEMBER, or both, as input to the bind plan.

9 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.

10 QUALIFIER

Lets you specify the implicit qualifier for unqualified tables, views and
aliases. You can specify a 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.

11 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. See “Determining the optimal authorization cache size” on page 504 for more information about this option.

12 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.

13 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.

14 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. For more information about this option, see the bind option CURRENTSERVER in Part 3 of *DB2 Command Reference*.

15 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.

Rebind Package panel

The Rebind Package panel is the first of two panels. It specifies options in the rebind process of a package. Figure 160 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	 ==>	(*, Blank, (), or version-id)
6	ADDITIONAL PACKAGES?	 ==>	(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 160. The Rebind Package panel

This panel lets you choose options for rebinding a package. For information about the rebind options that these fields represent, see the REBIND PACKAGE command in Part 3 of *DB2 Command Reference*.

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 552. 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 “The Defaults for Bind or Rebind Package or Plan panels” on page 546.

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 implicit qualifier for all unqualified table names, views, indexes, and aliases in the package. You can specify a qualifier 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 167 on page 551) 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.

Rebind Trigger Package panel

The Rebind Trigger Package panel specifies options you can change when you rebind a trigger package. Figure 161 on page 541 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)	

Figure 161. The Rebind Trigger Package panel

This panel lets you choose options for rebinding a trigger package. For information about the rebind options that these fields represent, see the REBIND TRIGGER PACKAGE command in Part 3 of *DB2 Command Reference*.

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.

Use RR, RS, CS, UR, or NC. For a description of the effects of those values, see "The ISOLATION option" on page 412.

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. For a description of the effects of those values, see "The ACQUIRE and RELEASE options" on page 408.

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_STATEMENT_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.

For information about EXPLAIN and creating a *PLAN_TABLE*, see “Obtaining *PLAN_TABLE* information from EXPLAIN” on page 790.

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. For more information about data currency, see “Maintaining data currency by using cursors” on page 467.

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.

For more information about this option, see the bind option IMMEDWRITE in Part 3 of *DB2 Command Reference*.

Rebind Plan panel

The Rebind Plan panel is the first of two panels. It specifies options in the rebind process of an application plan. Figure 162 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 162. The Rebind Plan panel

This panel lets you specify options for rebinding your plan. For information about the rebind options that these fields represent, see the REBIND PLAN command in Part 3 of *DB2 Command Reference*.

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 552. 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. For information about the defaults for binding plans, see "The Defaults for Bind or Rebind Package or Plan panels" on page 546.

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 implicit qualifier for all unqualified table names, views, indexes, and aliases in the plan. You can specify a qualifier 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. See "Determining the optimal authorization cache size" on page 504 for more information about this option.

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 167 on page 551) 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 552). 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. For more information about this option, see the bind option CURRENTSERVER in Part 3 of *DB2 Command Reference*.

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.

Free Package panel

The Free Package panel is the first of two panels. It specifies options in the free process of a package. Figure 163 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)

Figure 163. 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 552. The default is NO.

Free Plan panel

The Free Plan panel is the first of two panels. It specifies options in the free process of an application plan. Figure 164 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 164. 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 552. The default is NO.

The Defaults for Bind or Rebind Package or Plan panels

On this panel, enter new defaults for binding the package.

```
DSNEBP10          DEFAULTS FOR BIND PACKAGE          SSID: DSN
COMMAND ==> _

Change default options as necessary:

 1 ISOLATION LEVEL ..... ==> (RR, RS, CS, 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, or ONCE)
 9 DEFER PREPARE ..... ==> (NO OR YES)
10 KEEP DYN SQL PAST COMMIT ==> (NO or YES)
11 DBPROTOCOL ..... ==> (DRDA OR PRIVATE)
12 APPLICATION ENCODING ... ==> (Blank, ASCII, EBCDIC,
                                UNICODE, or ccsid)
13 OPTIMIZATION HINT ..... ==> (Blank or 'hint-id')
14 IMMEDIATE WRITE ..... ==> (YES, NO)
15 DYNAMIC RULES ..... ==> (RUN, BIND, DEFINE,
                                or INVOKE)
```

Figure 165. The Defaults for Bind Package panel

This panel lets you change your defaults for BIND PACKAGE options. 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.

On this panel, enter new defaults for binding your 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)
 9 DEFER PREPARE ..... ==> NO        (NO or YES)
10 KEEP DYN SQL PAST COMMIT. ==> NO    (NO or YES)
11 DBPROTOCOL ..... ==>              (Blank, DRDA, OR PRIVATE)
12 APPLICATION ENCODING ... ==>        (Blank, ASCII, EBCDIC,
                                UNICODE, or ccsid)
13 OPTIMIZATION HINT ..... ==>        (Blank or 'hint-id')
14 IMMEDIATE WRITE ..... ==> NO       (YES, NO)
15 DYNAMIC RULES ..... ==> RUN        (RUN or BIND)
16 SQLRULES..... ==> DB2              (DB2 or STD)
17 DISCONNECT ..... ==> EXPLICIT      (EXPLICIT, AUTOMATIC,
                                or CONDITIONAL)
```

Figure 166. The Defaults for Bind Plan panel

This panel lets you change your defaults for options of BIND PLAN. The options on this panel are mostly the same as the options for the defaults for rebinding a

package. However, for REBIND PLAN defaults, you can specify SAME in any field to specify the values used the last time the plan was bound. For rebinding, the default value for all fields is SAME.

Explanations of panel fields: The fields in panels Defaults for Bind Package and Defaults for Bind Plan are:

1 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 plan or package if you are replacing an existing one.

Use RR, RS, CS, or UR. For a description of the effects of those values, see “The ISOLATION option” on page 412.

2 VALIDATION TIME

Lets you specify RUN or BIND to tell whether to check authorization at run time or at bind time. The default is that used for the old plan or package, if you are replacing it. For more information about this option, see the bind option VALIDATE in Part 3 of *DB2 Command Reference*.

3 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 plan or package, if you are replacing it. For a description of the effects of those values, see “The ACQUIRE and RELEASE options” on page 408.

4 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 NO.

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_STATEMENT_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.

For information about EXPLAIN and creating a *PLAN_TABLE*, see “Obtaining *PLAN_TABLE* information from EXPLAIN” on page 790.

5 DATA CURRENCY

Lets you specify YES or NO for whether you need data currency for ambiguous cursors opened at remote locations.

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. For more information about data currency, see “Maintaining data currency by using cursors” on page 467.

6 PARALLEL DEGREE

Lets you specify ANY to run queries using parallel processing (when possible) or 1 to request that DB2 not execute queries in parallel. See Chapter 28, “Parallel operations and query performance,” on page 847 for more information about this option.

8 REOPTIMIZE FOR INPUT VARS

Specifies whether DB2 determines access paths at bind time and again at execution time for statements that contain:

- Input host variables
- Parameter markers

- Special registers

If you specify ALWAYS, DB2 determines the access paths again at execution time. When you specify ALWAYS for this option, you must also specify YES for DEFER PREPARE, or you will receive a bind error. If you specify ONCE, DB2 determines the access path at the first execution or open time. It saves and continues to use that access path for that specific statement until the statement is invalidated or removed from the dynamic statement cache or until the statement needs to be prepared again. The default, NONE, specifies that DB2 does not determine the access path at bind time using input host variables, parameter markers, or special registers.

9 DEFER PREPARE

Lets you defer preparation of dynamic SQL statements until DB2 encounters the first OPEN, DESCRIBE, or EXECUTE statement that refers to those statements. Specify YES to defer preparation of the statement. For information about using this option, see “Using bind options to improve performance for distributed applications” on page 456.

10 KEEP DYN SQL PAST COMMIT

Specifies whether DB2 keeps dynamic SQL statements after commit points. YES causes DB2 to keep dynamic SQL statements after commit points. An application can execute a PREPARE statement for a dynamic SQL statement once and execute that statement after later commit points without executing PREPARE again. For more information, see “Performance of static and dynamic SQL” on page 595.

11 DBPROTOCOL

Specifies whether DB2 uses DRDA protocol or DB2 private protocol to execute statements that contain 3-part names. For more information, see Chapter 20, “Planning to access distributed data,” on page 441.

12 APPLICATION ENCODING

Specifies the application encoding scheme to be used:

blank	Indicates that all host variables in static SQL statements are encoded using the encoding scheme in the DEF ENCODING SCHEME field of installation panel DSNTIPF.
ASCII	Indicates that the CCSIDs for all host variables in static SQL statements are determined by the values in the ASCII CODED CHAR SET and MIXED DATA fields of installation panel DSNTIPF.
EBCDIC	Indicates that the CCSIDs for all host variables in static SQL statements are determined by the values in the EBCDIC CODED CHAR SET and MIXED DATA fields of installation panel DSNTIPF.
UNICODE	Indicates that the CCSIDs of all host variables in static SQL statements are determined by the value in the UNICODE CCSID field of installation panel DSNTIPF.
<i>ccsid</i>	Specifies a CCSID that determines the set of CCSIDs that are used for all host variables in static SQL statements. If you specify <i>ccsid</i> , this value should be a mixed CCSID. For Unicode, the mixed CCSID is a UTF-8 CCSID. DB2 derives the SBCS and DBCS CCSIDs.

13 OPTIMIZATION HINT

Specifies whether you want to use optimization hints to determine access paths. Specify *hint-id* to indicate that you want DB2 to use the optimization hints in *owner.PLAN_TABLE*, where *owner* is the authorization ID of the plan or package owner. *hint-id* is a delimited string of up to 8 characters that DB2 compares to the value of OPTHINT in *owner.PLAN_TABLE* to determine the rows to use for optimization hints. If you specify a nonblank value for *hint-id*, DB2 uses optimization hints only if the value of field OPTIMIZATION HINTS on installation panel DSNTIP8 is YES.

Blank means that you do not want DB2 to use optimization hints. This is the default. For more information, see Part 5 (Volume 2) of *DB2 Administration Guide*.

14 IMMEDIATE WRITE

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:

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. NO is the default.

PH1 is equivalent to NO.

YES Write the changes immediately after group buffer pool-dependent pages are updated.

For more information about this option, see the bind option IMMEDWRITE in Part 3 of *DB2 Command Reference*.

15 DYNAMIC RULES

For plans, lets you specify whether run-time (RUN) or bind-time (BIND) rules apply to dynamic SQL statements at run time.

For packages, lets you specify whether run-time (RUN) or bind-time (BIND) rules apply to dynamic SQL statements at run time. For packages that run under an active user-defined function or stored procedure environment, the INVOKEBIND, INVOKERUN, DEFINEBIND, and DEFINERUN options indicate who must have authority to execute dynamic SQL statements in the package.

For packages, the default rules for a package on the local server are the same as the rules for the plan to which the package appends at run time. For a package on the remote server, the default is RUN.

If you specify rules for a package that are different from the rules for the plan, the SQL statements for the package use the rules you specify for that package. If a package that is bound with DEFINEBIND or INVOKEBIND is not executing under an active stored procedure or user-defined function environment, SQL statements for that package use BIND rules. If a package that is bound with DEFINERUN or INVOKERUN is not executing under an active stored procedure or user-defined function environment, SQL statements for that package use RUN rules.

For more information, see “Using DYNAMICRULES to specify behavior of dynamic SQL statements” on page 502.

For packages:

7 SQLERROR PROCESSING

Lets you specify CONTINUE to continue to create a package after finding SQL errors, or NOPACKAGE to avoid creating a package after finding SQL errors.

For plans:

7 RESOURCE ACQUISITION TIME

Lets you specify when to acquire locks on resources. Use:

USE (default) to open table spaces and acquire locks only when the program bound to the plan first uses them.

ALLOCATE to open all table spaces and acquire all locks when you allocate the plan. This value has no effect on dynamic SQL.

For a description of the effects of those values, see “The ACQUIRE and RELEASE options” on page 408.

16 SQLRULES

Lets you specify whether a CONNECT (Type 2) statement executes according to DB2 rules (DB2) or the SQL standard (STD). For information, see “Specifying the SQL rules” on page 505.

17 DISCONNECT

Lets you specify which remote connections end during a commit or a rollback. Regardless of what you specify, all connections in the released-pending state end during commit.

Use:

EXPLICIT to end connections in the release-pending state only at COMMIT or ROLLBACK

AUTOMATIC to end all remote connections

CONDITIONAL to end remote connections that have no open cursors WITH HOLD associated with them.

See the DISCONNECT option of the BIND PLAN subcommand in Part 3 of *DB2 Command Reference* for more information about these values.

System Connection Types panel

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 167. 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 more information about this option, see the bind options ENABLE and DISABLE in Part 3 of *DB2 Command Reference*.

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 RRSF 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

For each connection type that has a second arrow, under SPECIFY CONNECTION NAMES?, enter Y if you want to list specific connection names of that type. Leave N (the default) if you do not. If you use Y in any of those fields, you see another panel on which you can enter the connection names. For more information, see “Panels for entering lists of values” on page 552.

If you use the DISPLAY command under TSO on this panel, you can determine what you have currently defined as “enabled” or “disabled” in your ISPF DSNPFT library (member DSNCONNS). The information does not reflect the current state of the DB2 Catalog.

If you type DISPLAY ENABLED on the command line, you get the connection names that are currently enabled for your TSO connection types. For example:

Display OF ALL connection name(s) to be ENABLED

CONNECTION	SUBSYSTEM
CICS1	ENABLED
CICS2	ENABLED
CICS3	ENABLED
CICS4	ENABLED
DLI1	ENABLED
DLI2	ENABLED
DLI3	ENABLED
DLI4	ENABLED
DLI5	ENABLED

Panels for entering lists of values

Some fields in DB2I panels are associated with command keywords that accept multiple values. Those fields lead you to a list panel that lets you enter or modify an unlimited number of values. A list panel looks like an ISPF edit session and lets you scroll and use a limited set of commands.

The format of each list panel varies, depending on the content and purpose for the panel. Figure 168 is a generic sample of a list panel:

```
panelid          Specific subcommand function          SSID: DSN
COMMAND ===> _          SCROLL ===>

Subcommand operand values:

CMD
value ...
value ...
value ...
value ...
value ...
value ...
```

Figure 168. Generic example of a DB2I list panel

For the syntax of specifying names on a list panel, see Part 3 of *DB2 Command Reference* for the type of name you need to specify.

All of the list panels let you enter limited commands in two places:

- On the system command line, prefixed by =====>
- In a special command area, identified by """"

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:

I nn Insert nn lines after this one.

D nn Delete this and the following lines for nn lines.

R nn Repeat this line nn number of times.

The default for *nm* 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 second of the Program Preparation panels (Figure 169) lets you do the last two steps in the program preparation process (compile and link-edit), as well as the PL/I MACRO PHASE for programs requiring 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 169. 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 if you wish. 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 if you wish. 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 if you wish. 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 fill in 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 if you wish. There is no default.

The prelink utility applies only to programs using C, C++, and Enterprise COBOL for z/OS. See *z/OS Language Environment Programming Guide* for more information about prelinker options.

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.

The Run panel

You can reach the Run panel only through the DB2I Primary Options Menu. The Run panel enables you to start an application program that can contain SQL statements. 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 170 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 170. 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.

Chapter 22. Testing an application program

This chapter discusses how to set up a test environment, test SQL statements, debug your programs, and read output from the precompiler. It contains the following sections:

- “Establishing a test environment”
- “Testing SQL statements using SPUFI” on page 560
- “Debugging your program” on page 560
- “Locating the problem” on page 566

Establishing a test environment

This section 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 corresponding entries in the SNT and RACF control areas authorize your application to run. The system administrator is responsible for these functions; see Part 3 (Volume 1) of *DB2 Administration Guide* for more information on the functions.

In addition, ensure that the program and its transaction code are defined in the CICS CSD.

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; for details, see Chapter 5 of *DB2 SQL Reference*. See Part 3 (Volume 1) of *DB2 Administration Guide* for more information about authorization IDs.

If your location has a separate DB2 system for testing, you can create the test tables and views on the test system, then test your program thoroughly on that system. This chapter 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.

Analyzing application data needs

To design test tables and views, first analyze your application's data needs.

1. List the data your application accesses and describe how it accesses each data item. For example, suppose you are testing an application that accesses the DSN8810.EMP, DSN8810.DEPT, and DSN8810.PROJ tables. You might record the information about the data as shown in Table 74.

Table 74. Description of the application's data

Table or view name	Insert rows?	Delete rows?	Column name	Data type	Update access?
DSN8810.EMP	No	No	EMPNO	CHAR(6)	No
			LASTNAME	VARCHAR(15)	No
			WORKDEPT	CHAR(3)	Yes
			PHONENO	CHAR(4)	Yes
			JOB	DECIMAL(3)	Yes
DSN8810.DEPT	No	No	DEPTNO	CHAR(3)	No
			MGRNO	CHAR (6)	No
DSN8810.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 you need to test your application.

Create a test table on your list when either:

- The application modifies data in the table
- You need to create a view based on a test table because your application modifies the view's data.

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 DSN8810.PROJ, because the application inserts rows into the DSN8810.PROJ table.

To support the example, create a test view of the DSN8810.DEPT table.

- TEST.DEPT view, with the following format:

DEPTNO	MGRNO
⋮	⋮

Because the application does not change any data in the DSN8810.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.

Obtaining authorization

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 you want to test. Your DBA can grant you the authorization needed 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 you are authorized to access (as well as the privileges you have for each table) from your DBA. See Chapter 2, “Working with tables and modifying data,” on page 19 for more information about creating tables and views.

Creating a comprehensive test structure

The following SQL statements shows 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 SPUFIDB and using 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 ;
```

For details about each CREATE statement, see *DB2 SQL Reference*.

Filling the tables with test data

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. For information about the INSERT statement, see “Inserting rows: INSERT” on page 27.

- INSERT ... SELECT (an SQL statement) obtains data from an existing table (based on a SELECT clause) and puts it into the table identified with the INSERT statement. For information about this technique, see “Inserting rows into a table from another table” on page 29.
- The LOAD utility obtains data from a sequential file (a non-DB2 file), formats it for a table, and puts it into a table. For more details about the LOAD utility, see *DB2 Utility Guide and Reference*.
- The DB2 sample UNLOAD program (DSNTIAUL) can unload data from a table or view and build control statements for the LOAD utility. See Appendix C, “Running the productivity-aid sample programs,” on page 1019 for more information about the sample UNLOAD program.
- The UNLOAD utility can unload data from a table and build control statements for the LOAD utility. See Part 2 of *DB2 Utility Guide and Reference* for more information about the UNLOAD utility.

Testing SQL statements using 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 permit you to:

- Name an input data set to hold the SQL statements passed to DB2 for execution
- Name an output data set to contain the results of executing the SQL statements
- Specify SPUFI processing options.

SQL statements executed under SPUFI operate on actual tables (in this case, the tables you have created for testing). Consequently, before you access DB2 data:

- Make sure that all tables and views 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 to issue the CREATE statements used to create the tables and views you need for testing.

For more information about how to use SPUFI, see Chapter 5, “Using SPUFI to execute SQL from your workstation,” on page 59.

Debugging your program

Many sites have guidelines regarding what to do if your program abends. The following sections suggest some common guidelines:

- “Debugging programs in TSO”
- “Debugging programs in IMS” on page 561
- “Debugging programs in CICS” on page 562

Debugging programs in TSO

Documenting the errors returned from test 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 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.

Language test facilities

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.

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)')
```

For more information about the TEST command, see *z/OS TSO/E Command Reference*.

ISPF Dialog Test is another option to help you in the task of debugging.

Debugging programs in IMS

Documenting the errors returned from test helps you investigate and correct problems in the program. The following information can be useful:

- The program's application plan name
- The input message 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 program's PSB name
- 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's 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 program's termination. 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 sites run a BMP at the end of the day to list all the errors that occurred during the day. If your location does this, you can send a message using an express PCB that has its destination set for that BMP.

Batch Terminal Simulator (BTS): The Batch Terminal Simulator (BTS) allows you to test IMS application programs. BTS traces application program DL/I calls and SQL statements, and simulates data communication functions. It can make a TSO terminal appear as an IMS terminal to the terminal operator, allowing the end user to interact with the application as though it were online. The user can use any application program under the user's control to access any database (whether DL/I or DB2) under the user's control. Access to DB2 databases requires BTS to operate in batch BMP or TSO BMP mode.

Debugging programs in CICS

Documenting the errors returned from test helps you investigate and correct problems in the program. The following information can be useful:

- The program's application plan name
- The input data 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 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. See "CICS execution diagnostic facility" on page 563 for more information. If you are using an earlier version of CICS, the CALL TO RESOURCE MANAGER DSNCSQL screen displays a status of "ABOUT TO EXECUTE" or "COMMAND EXECUTION COMPLETE."

Abend recovery. You can use the HANDLE ABEND command to deal with abend conditions, and 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 generated by application programs; you can have it 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.

For more details about each of these topics, see *CICS Transaction Server for z/OS Application Programming Reference*.

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, making it possible to 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, while the task is in EDF mode. This can be a significant aid in debugging CICS transaction programs containing 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 you need to do to write special error handlers.

EDF before execution: Figure 171 is 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 171. 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, such as COMMIT, DROP TABLE, EXPLAIN, FETCH, or OPEN.
- DBRM=*dbrm name*
The name of the database request module (DBRM) 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 it to determine which statement is processing. This number is a source line counter that includes host language statements. A statement number greater than 32,767 displays as 0.
- SECT=*section number*
The section number of the plan that the SQL statement uses.

SQL statements containing input host variables: The IVAR (input host variables) section and its attendant fields only appear 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 being prepared. The address of the input variable is AT 'nnnnnnnnnn'.

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. For additional information, see "Data types" on page 4.
- LEN=*length*
Length of the host variable.
- IND=*indicator variable status number*
Represents the indicator variable 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. For additional information, see "Using indicator variables with host variables" on page 83.
- DATA=*host variable data*
The data, displayed in hexadecimal format, 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 more data is present.

EDF after execution: Figure 172 on page 565 shows an example of the first EDF screen 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'03C920A0'
          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 172. 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*

If the RACF list of group options is not active, then 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, then DB2 ignores the connected group name that the CICS attachment facility supplies, but the value appears in the DB2 list of secondary authorization IDs.

- PLAN=*plan name*

The name of plan that is currently running. The PLAN represents the control structure produced during the bind process and used by DB2 to process SQL statements encountered while the application is running.

- SQL Communication Area (SQLCA)

The SQLCA contains information about errors, if any occur. After returning from DB2, the information is available. 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 only appear when the executing statement returns output host variables.

Figure 173 on page 566 contains the rest of the EDF output for our example.

Figure 173. EDF screen after a DB2 SQL statement, continued

The attachment facility automatically displays SQL information while in the EDF mode. (You can start EDF as outlined in the appropriate CICS application programmer's reference manual.) If this is not the case, contact your installer and see Part 2 of *DB2 Installation Guide*.

Locating the problem

If your program does not run correctly, you need to isolate the problem. If the DB2 did not invalidate the program's application plan, you should check the following items:

- Output from the compiler or assembler. 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 for TSO
 - DFSLI000 for IMS
 - DSNCLI for CICS
 - DSNALI for the call 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 you want to process the DBRM for diagnostic purposes, but do not want to produce an application plan.
 - Have you specified all the DBRMs and packages 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 large enough for the storage required 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 instance, 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 FREE the dump data set.

Analyzing error and warning messages from the precompiler

Under some circumstances, the statements that the DB2 precompiler generates can produce compiler or assembly error messages. You must know why the messages occur when you compile DB2-produced source statements. For more information about warning messages, see the following host language sections:

- “Coding SQL statements in an assembler application” on page 143
- “Coding SQL statements in a C or C++ application” on page 158
- “Coding SQL statements in a COBOL application” on page 186
- “Coding SQL statements in a Fortran application” on page 220
- “Coding SQL statements in a PL/I application” on page 230.

SYSTEM output from the precompiler

The DB2 precompiler provides SYSTEM output when you allocate the ddname SYSTEM. If you use the Program Preparation panels to prepare and run your program, DB2I allocates SYSTEM according to the TERM option you specify.

The SYSTEM output provides a brief summary of the results from the precompiler, all error messages that the precompiler generated, and the statement in error, when possible. Sometimes, the error messages by themselves are not enough. In such cases, you can use the line number provided in each error message to locate the failing source statement.

Figure 174 on page 568 shows the format of SYSTEM 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 174. DB2 precompiler SYSTERM output

Notes for Figure 174:

1. Error message.
2. Source SQL statement.
3. Summary statements of source statistics.
4. Summary statement of the number of errors detected.
5. Summary statement indicating the number of errors detected but not printed. That value might occur if you specify a FLAG option other than I.
6. Storage requirement statement telling you 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 is what the DB2 precompiler provides when you use a procedure to precompile your program. See Table 70 on page 513 for a list of JCL procedures that DB2 provides.

When you use the Program Preparation panels to prepare and run your program, DB2 allocates SYSPRINT according to TERM option 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 (Figure 175) that are in effect during the precompilation (if you did not specify NOOPTIONS).

```

DB2 SQL PRECOMPILER          Version 8

OPTIONS SPECIFIED: HOST(PLI),XREF,SOURCE1

OPTIONS USED - SPECIFIED OR DEFAULTED2
  APOST
  APOSTSQL
  CONNECT(2)
  DEC(15)
  FLAG(I)
  NOGRAPHIC
  HOST(PLI)
  NOT KATAKANA
  LINECOUNT(60)
  MARGINS(2,72)
  ONEPASS
  OPTIONS
  PERIOD
  SOURCE
  STDSQL(NO)
  SQL(DB2)
  XREF

```

Figure 175. DB2 precompiler SYSPRINT output: Options section

Notes for Figure 175:

1. This section lists the options 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 you specified. The DB2 precompiler overrides or ignores any options you specify that are inappropriate for the host language.
- A listing (Figure 176) of your source statements (only if you specified the SOURCE option).

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		00133600
1337	EXEC SQL DELETE FROM PROJ	00133700
1338	WHERE PROJNO = :PROJ_NO;	00133800
:		
1523	END;	00152300

Figure 176. DB2 precompiler SYSPRINT output: Source statements section

Notes for Figure 176:

- 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 of sequence numbers come from the sequence numbers supplied with your source statements.
- A list (Figure 177) of the symbolic names used in SQL statements (this listing appears only if you specify the XREF option).

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 177. DB2 precompiler SYSPRINT output: Symbol cross-reference section

Notes for Figure 177:

DATA NAMES

Identifies the symbolic names used in source statements. Names enclosed in 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: what the source program defines the symbolic name to be, 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 STRUCTURE.

- A summary (Figure 178) of the errors detected by the DB2 precompiler and a list of the error messages generated by the precompiler.

```

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

```

Figure 178. DB2 precompiler SYSPRINT output: Summary section

Notes for Figure 178:

1. Summary statement indicating the number of source lines.
2. Summary statement indicating the number of symbolic names in the symbol table (SQL names and host names).
3. Storage requirement statement indicating the number of bytes for the symbol table.
4. Summary statement indicating the number of messages printed.
5. Summary statement indicating the number of errors detected but not printed. You might get this statement if you specify the option FLAG.
6. Storage requirement statement indicating the number of bytes of working storage 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).

Chapter 23. Processing DL/I batch applications

This chapter describes DB2 support for DL/I batch applications under these headings:

- “Planning to use DL/I batch applications”
- “Program design considerations ” on page 574
- “Input and output data sets for DL/I batch jobs” on page 576
- “Preparation guidelines for DL/I batch programs” on page 578
- “Restart and recovery ” on page 580

Planning to use DL/I batch applications

Features and functions of DB2 DL/I batch support tells what you can do in a DL/I batch program. “Requirements for using DB2 in a DL/I batch job” on page 574 tells, in general, what you must do to make it happen.

Features and functions of DB2 DL/I batch support

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 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 abend units of recovery
- The DB2 IMS attachment facility, which handles the two-phase commit protocol and allows both systems to synchronize a unit of recovery during a restart after a failure
- The IMS log, used to record the instant of commit.

In a data sharing environment, DL/I batch supports group 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. See “DB2 DL/I batch input” on page 576 for information about the SSN parameter and the DDITV02 data set.

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:

- You must 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 plan or package, as described in Chapter 21, “Preparing an application program to run,” on page 471.
- 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. For detailed information about the contents of the subsystem member and the DDITV02 data set, see “DB2 DL/I batch input” on page 576.

- 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 and diagnostic information.

Authorization

When the 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 later 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. It is the TSO logon name if the job is submitted. Otherwise, it 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 abend. The batch job is rejected if you try to change the authorization ID in an exit routine.

Program design considerations

Using DL/I batch can affect your application design and programming in the areas described in the sections that follow.

Address spaces

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

Commit IMS batch applications frequently so that you do not use resources for an extended time. If you need coordinated commits for recovery, see Part 4 (Volume 1) of *DB2 Administration Guide*.

SQL statements and IMS calls

You cannot use the SQL COMMIT and ROLLBACK statements, which return an SQL error code. You also cannot use ROLS, SETS, and SYNC calls, which cause the application program to abend.

Checkpoint calls

Write your program with SQL statements and DL/I calls, and use checkpoint calls. All checkpoints issued by a batch application program must be unique. The frequency of checkpoints depends on the application design. At a checkpoint, DL/I positioning is lost, DB2 cursors are closed (with the possible exception of cursors 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

You can design an application program without using IMS checkpoints. In that case, if the program abends 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 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, then 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.

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. When you restart the application program, you must use the XRST call to obtain checkpoint information and resolve any DB2 indoubt work units.

Checkpoint and XRST considerations

If you use an XRST call, DB2 assumes that any checkpoint 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, then DB2 assumes that any checkpoint call issued is a basic checkpoint.

Checkpoint IDs must be EBCDIC characters to make restart easier.

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 issued and must occur before any SQL statement. Also, you must use only one XRST call.

Synchronization call abends

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.

Input and output data sets for DL/I batch jobs

Two data sets need your attention:

- DDITV02 for input
- DDOTV02 for output.

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 abends 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	The name of the DB2 subsystem 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. See Chapter 2 of *DB2 Data Sharing: Planning and Administration* for more information about group attachment.

LIT	DB2 requires a language interface token to route SQL statements when operating in the online IMS environment. Because a batch application

program can only connect to one DB2 system, DB2 does not use the LIT value. It is recommended that you specify the value as SYS1; however, you can omit it (enter SSN,,ESMT).

The LIT value can be from zero to four characters long.

ESMT The name of the DB2 initialization module, DSNMIN10, is required.

The ESMT value must be eight characters long.

RTT Specifying the resource translation table is optional.

The RTT can be from zero to eight characters long.

REO The region error option determines what to do if DB2 is not operational or the plan is not available. There are three options:

- *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, it 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, then 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 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

The connection name is optional. It represents the name of the job step that coordinates DB2 activities. 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 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 1 to 8 characters long.

PLAN The DB2 plan name is optional. If you do not specify the plan name, then the application program module name is checked against the optional resource translation table. If there is a match in the resource translation

table, the translated name is used as the DB2 plan name. If there is no match, then the application program module name is used as the plan name.

The PLAN value can be from 0 to 8 characters long.

PROG The application program name is required. It identifies the application program that is to be loaded and to receive control.

The PROG value can be from 1 to 8 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 specified in the DDOTV02 DD statement. The output data set should have 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.

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:

- “Precompiling ”
- “Binding ”
- “Link-editing ” on page 579
- “Loading and running ” on page 579

Precompiling

When you add SQL statements to an application program, you must precompile the application program and bind the resulting DBRM into a plan or package, as described in Chapter 21, “Preparing an application program to run,” on page 471.

Binding

The owner of the plan or package must have all the privileges required to execute the SQL statements embedded in it. Before a batch program can issue SQL statements, a DB2 plan must exist.

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 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, then DB2 uses the passed name as the plan name. If the name exists in the RTT, then the name translates to the plan specified for the RTT.

Recommendation: Give the DB2 plan the same name as that of the application load module, which is the IMS attach default. The plan name must be the same as the program name.

Link-editing

DB2 has language interface routines for each unique supported environment. DB2 requires the IMS language interface routine for DL/I batch. It is also necessary to have DFSLI000 link-edited with the application program.

Loading and running

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.

There are two ways to submit DL/I batch applications to DB2:

- The DL/I batch procedure can run module DSNMTV01 as the application program. DSNMTV01 loads the “real” application program. See “Submitting a DL/I batch application using DSNMTV01” for an example of JCL used to submit a DL/I batch application by this method.
- The DL/I batch procedure can run your application program without using module DSNMTV01. To accomplish this, do the following:
 - Specify SSM= in the DL/I batch procedure.
 - In the batch region of your application’s JCL, specify the following:
 - MBR=*application-name*
 - SSM=DB2 *subsystem name*

See “Submitting a DL/I batch application without using DSNMTV01” on page 580 for an example of JCL used to submit a DL/I batch application by this method.

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
//JOB LIB 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,BK0=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.STEPCAT DD DSN=IMSCAT,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=DFSERA10,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
/*

```

Restart and recovery

To restart a batch program that updates data, you must first run the IMS batch backout utility, followed by a restart job indicating the last successful checkpoint ID.

- Sample JCL for the utility is in “JCL example of a batch backout” on page 581.
- Sample JCL for a restart job is in “JCL example of restarting a DL/I batch job” on page 581.
- For guidelines on finding the last successful checkpoint, see “Finding the DL/I batch checkpoint ID” on page 582.

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
//STPCAT DD DSN=IMSCAT,DISP=SHR
//IMS DD DSN=IMS.PSBLIB,DISP=SHR
// DD DSN=IMS.DBDLIB,DISP=SHR
//*
//* IMSLOGR DD data set is required
//* IEFORDER 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, then 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.STEPCAT DD DSN=IMSCAT,DISP=SHR
/* 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.DDOTV02 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     DSN=&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 issued. Use the last checkpoint ID listed.
- Look at the z/OS console log to find message(s) DFS0540I containing the checkpoint ID issued for this batch program. Use the last checkpoint ID listed.
- 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, the operator can 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, it could work because the checkpoint is recorded on the IMS log; however, it 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)
```

Part 6. Additional programming techniques

Chapter 24. Coding dynamic SQL in application programs	593
Choosing between static and dynamic SQL	593
Flexibility of static SQL with host variables	594
Flexibility of dynamic SQL	594
Limitations of dynamic SQL	594
Dynamic SQL processing	594
Performance of static and dynamic SQL	595
Static SQL statements with no input host variables	595
Static SQL statements with input host variables	595
Dynamic SQL statements	595
Caching dynamic SQL statements	596
Using the dynamic statement cache	597
Conditions for statement sharing	598
# Using the statement cache table	599
Keeping prepared statements after commit points	599
Limiting dynamic SQL with the resource limit facility	601
Writing an application to handle reactive governing	602
Writing an application to handle predictive governing	602
Handling the +495 SQLCODE	602
Using predictive governing and down-level DRDA requesters	603
Using predictive governing and enabled requesters	603
Choosing a host language for dynamic SQL applications	603
Dynamic SQL for non-SELECT statements	603
Dynamic execution using EXECUTE IMMEDIATE	604
Declaring the host variable	604
Dynamic execution using PREPARE and EXECUTE	605
Using parameter markers with PREPARE and EXECUTE	606
Using the PREPARE statement	606
Using the EXECUTE statement	607
Preparing and executing the example DELETE statement	607
Using more than one parameter marker	607
Dynamic execution of a multiple-row INSERT statement	608
Using EXECUTE with host variable arrays	608
Using EXECUTE with a descriptor	608
Using DESCRIBE INPUT to put parameter information in an SQLDA	610
Dynamic SQL for fixed-list SELECT statements	610
Declaring a cursor for the statement name	611
Preparing the statement	611
Opening the cursor	612
Fetching rows from the result table	612
Closing the cursor	612
Dynamic SQL for varying-list SELECT statements	613
What your application program must do	613
Preparing a varying-list SELECT statement	613
An SQL descriptor area	614
Obtaining information about the SQL statement	614
Declaring a cursor for the statement	615
Preparing the statement using the minimum SQLDA	615
SQLN determines what SQLVAR gets	615
If the statement is not a SELECT	616
Acquiring storage for a second SQLDA if needed	616
Describing the SELECT statement again	617
Acquiring storage to hold a row	617
Putting storage addresses in the SQLDA	619
Changing the CCSID for retrieved data	619
Using column labels	621

	Describing tables with LOB and distinct type columns	621
	Executing a varying-list SELECT statement dynamically	623
	Open the cursor	623
	Fetch rows from the result table	623
	Close the cursor	623
	Executing arbitrary statements with parameter markers	624
	When the number and types of parameters are known	624
	When the number and types of parameters are not known	624
	Using the SQLDA with EXECUTE or OPEN	625
	How bind options REOPT(ALWAYS) and REOPT(ONCE) affect dynamic SQL.	625
	Using dynamic SQL in COBOL	627
	Chapter 25. Using stored procedures for client/server processing	629
	Introduction to stored procedures	629
	An example of a simple stored procedure	630
	Setting up the stored procedures environment	634
	Defining your stored procedure to DB2.	635
	Passing environment information to the stored procedure	636
	Example of a stored procedure definition	638
	Refreshing the stored procedures environment (for system administrators)	639
	Moving stored procedures to a WLM-established environment (for system administrators)	640
	Writing and preparing an external stored procedure	641
	Language requirements for the stored procedure and its caller	641
	Calling other programs	642
	Using reentrant code	642
	Writing a stored procedure as a main program or subprogram	643
	Restrictions on a stored procedure	646
	Using COMMIT and ROLLBACK statements in a stored procedure	646
	Using special registers in a stored procedure	647
	Accessing other sites in a stored procedure	649
	Writing a stored procedure to access IMS databases	650
	Writing a stored procedure to return result sets to a DRDA client	650
	Preparing a stored procedure	652
	Binding the stored procedure	653
	Writing a REXX stored procedure	654
	Writing and preparing an SQL procedure	657
	Comparison of an SQL procedure and an external procedure	658
	Statements that you can include in a procedure body	659
#	Declaring and using variables, parameters, and conditions in an SQL procedure	661
	Parameter style for an SQL procedure	662
	Terminating statements in an SQL procedure	662
	Handling SQL conditions in an SQL procedure	663
	Using handlers in an SQL procedure	663
	Using the RETURN statement for the procedure status	665
	Using SIGNAL or RESIGNAL to raise a condition	665
	Forcing errors in an SQL procedure when called by a trigger	667
	Examples of SQL procedures	667
	Preparing an SQL procedure	669
	Using the DB2 UDB for z/OS SQL procedure processor to prepare an SQL procedure	670
	Using JCL to prepare an SQL procedure	680
	Sample programs to help you prepare and run SQL procedures	680
	Writing and preparing an application to use stored procedures.	681
	Forms of the CALL statement	682
	Authorization for executing stored procedures	683
	Linkage conventions	684
	Example of stored procedure linkage convention GENERAL	686
	Example of stored procedure linkage convention GENERAL WITH NULLS	689
	Example of stored procedure linkage convention SQL	694
	Special considerations for C	702
	Special considerations for PL/I	703
	Using indicator variables to speed processing	703

	Declaring data types for passed parameters	703
	Writing a DB2 UDB for z/OS client program or SQL procedure to receive result sets	708
	Accessing transition tables in a stored procedure	714
	Calling a stored procedure from a REXX procedure	714
	Preparing a client program	718
	Running a stored procedure	719
	How DB2 determines which version of a stored procedure to run	720
	Using a single application program to call different versions of a stored procedure	720
	Running multiple stored procedures concurrently	722
#	Multiple instances of a stored procedure	722
	Accessing non-DB2 resources	723
	Testing a stored procedure	725
	Debugging the stored procedure as a stand-alone program on a workstation	725
	Debugging with the Debug Tool and IBM VisualAge COBOL	725
	Debugging an SQL procedure or C language stored procedure with the Debug Tool and C/C++ Productivity Tools for z/OS	726
	Debugging with Debug Tool for z/OS interactively and in batch mode	727
	Using the MSGFILE run-time option	728
	Using driver applications	729
	Using SQL INSERT statements	729
	Chapter 26. Tuning your queries	731
	General tips and questions	731
	Is the query coded as simply as possible?	731
	Are all predicates coded correctly?	731
	Are there subqueries in your query?.	732
	Does your query involve aggregate functions?	733
	Do you have an input variable in the predicate of an SQL query?	734
	Do you have a problem with column correlation?	734
	Can your query be written to use a noncolumn expression?	734
	Can materialized query tables help your query performance?	734
	Does the query contain encrypted data?	735
	Writing efficient predicates	735
	Properties of predicates	735
	Predicate types	736
	Indexable and nonindexable predicates	737
	Stage 1 and stage 2 predicates	737
	Boolean term (BT) predicates	738
	Predicates in the ON clause	738
	General rules about predicate evaluation	739
	Order of evaluating predicates	739
	Summary of predicate processing	740
	Examples of predicate properties	745
	Predicate filter factors	746
	Default filter factors for simple predicates	747
	Filter factors for uniform distributions	747
	Interpolation formulas	748
	Filter factors for all distributions	749
	Using multiple filter factors to determine the cost of a query	751
	Column correlation	752
	How to detect column correlation	752
	Impacts of column correlation	753
	What to do about column correlation	755
	DB2 predicate manipulation	755
	Predicate modifications for IN-list predicates	756
	When DB2 simplifies join operations	756
	Predicates generated through transitive closure	757
	Predicates with encrypted data	759
	Using host variables efficiently	759
	Changing the access path at run time	760
	The REOPT(ALWAYS) bind option	760

	The REOPT(ONCE) bind option	761
	The REOPT(NONE) bind option	762
	Rewriting queries to influence access path selection	763
	Writing efficient subqueries.	766
	Correlated subqueries	766
	Noncorrelated subqueries	767
	Single-value subqueries	767
	Multiple-value subqueries	768
	Conditions for DB2 to transform a subquery into a join	768
	Subquery tuning	770
	Using scrollable cursors efficiently	771
	Writing efficient queries on tables with data-partitioned secondary indexes.	772
	Special techniques to influence access path selection	774
	Obtaining information about access paths	775
	Fetching a limited number of rows: FETCH FIRST <i>n</i> ROWS ONLY	775
	Minimizing overhead for retrieving few rows: OPTIMIZE FOR <i>n</i> ROWS	776
	Favoring index access	778
#	Using a subsystem parameter to control outer join processing	779
	Using the CARDINALITY clause to improve the performance of queries with user-defined table function	
	references	779
	Reducing the number of matching columns	780
	Creating indexes for efficient star-join processing	782
	Recommendations for creating indexes for star-join queries	782
	Determining the order of columns in an index for a star schema design	783
	Rearranging the order of tables in a FROM clause	784
	Updating catalog statistics	785
	Using a subsystem parameter	786
	Using a subsystem parameter to favor matching index access	786
	Using a subsystem parameter to optimize queries with IN-list predicates	787
	Chapter 27. Using EXPLAIN to improve SQL performance	789
	Obtaining PLAN_TABLE information from EXPLAIN	790
	Creating PLAN_TABLE	791
	Populating and maintaining a plan table	798
	Executing the SQL statement EXPLAIN	798
	Binding with the option EXPLAIN(YES)	799
	Maintaining a plan table.	799
	Reordering rows from a plan table	799
	Retrieving rows for a plan	799
	Retrieving rows for a package.	800
	Asking questions about data access	800
	Is access through an index? (ACCESSTYPE is I, I1, N or MX)	801
	Is access through more than one index? (ACCESSTYPE=M)	801
	How many columns of the index are used in matching? (MATCHCOLS=n)	802
	Is the query satisfied using only the index? (INDEXONLY=Y)	802
	Is direct row access possible? (PRIMARY_ACCESSTYPE = D)	803
	Which predicates qualify for direct row access?	803
	Reverting to ACCESSTYPE.	804
	Using direct row access and other access methods	804
	Example: Coding with row IDs for direct row access	805
	Is a view or nested table expression materialized?	806
	Was a scan limited to certain partitions? (PAGE_RANGE=Y)	806
	What kind of prefetching is expected? (PREFETCH = L, S, D, or blank)	807
	Is data accessed or processed in parallel? (PARALLELISM_MODE is I, C, or X)	807
	Are sorts performed?	807
	Is a subquery transformed into a join?	808
	When are aggregate functions evaluated? (COLUMN_FN_EVAL)	808
	How many index screening columns are used?	808
	Is a complex trigger WHEN clause used? (QBLOCKTYPE=TRIGGR)	809
	Interpreting access to a single table	809
	Table space scans (ACCESSTYPE=R PREFETCH=S)	809

Table space scans of nonsegmented table spaces	810
Table space scans of segmented table spaces	810
Table space scans of partitioned table spaces	810
Table space scans and sequential prefetch	810
Index access paths.	810
Matching index scan (MATCHCOLS>0)	811
Index screening.	811
Nonmatching index scan (ACCESSTYPE=I and MATCHCOLS=0)	812
IN-list index scan (ACCESSTYPE=N)	812
Multiple index access (ACCESSTYPE is M, MX, MI, or MU).	812
One-fetch access (ACCESSTYPE=I1)	814
Index-only access (INDEXONLY=Y)	814
Equal unique index (MATCHCOLS=number of index columns)	814
UPDATE using an index	815
Interpreting access to two or more tables (join)	815
Definitions and examples of join operations	815
Nested loop join (METHOD=1)	818
Method of joining	818
Performance considerations.	818
When nested loop join is used.	818
Merge scan join (METHOD=2)	820
Method of joining	820
Performance considerations.	821
When merge scan join is used	821
Hybrid join (METHOD=4)	821
Method of joining	822
Possible results from EXPLAIN for hybrid join	823
Performance considerations.	823
When hybrid join is used	823
Star join (JOIN_TYPE='S')	823
Example of a star schema	824
When star join is used	825
Dedicated virtual memory pool for star join operations	828
Interpreting data prefetch	830
Sequential prefetch (PREFETCH=S)	830
Dynamic prefetch (PREFETCH=D)	831
List prefetch (PREFETCH=L)	831
The access method	831
When list prefetch is used	832
Bind time and execution time thresholds	832
Sequential detection at execution time	832
When sequential detection is used	833
How to tell whether sequential detection was used.	833
How to tell if sequential detection might be used	833
Determining sort activity	834
Sorts of data.	834
Sorts for group by and order by	834
Sorts to remove duplicates	835
Sorts used in join processing	835
Sorts needed for subquery processing	835
Sorts of RIDs	835
The effect of sorts on OPEN CURSOR	835
Processing for views and nested table expressions	836
Merge	836
Materialization	837
Two steps of materialization	837
When views or table expressions are materialized	837
Using EXPLAIN to determine when materialization occurs	839
Using EXPLAIN to determine UNION activity and query rewrite	840
Performance of merge versus materialization	842
Estimating a statement's cost	842

Creating a statement table	843
Populating and maintaining a statement table	845
Retrieving rows from a statement table.	845
The implications of cost categories	846
Chapter 28. Parallel operations and query performance	847
Comparing the methods of parallelism	848
Enabling parallel processing	850
When parallelism is not used	851
Interpreting EXPLAIN output	852
A method for examining PLAN_TABLE columns for parallelism	852
PLAN_TABLE examples showing parallelism.	852
Tuning parallel processing	854
Disabling query parallelism	855
Chapter 29. Programming for the Interactive System Productivity Facility	857
Using ISPF and the DSN command processor	857
Invoking a single SQL program through ISPF and DSN	858
Invoking multiple SQL programs through ISPF and DSN.	859
Invoking multiple SQL programs through ISPF and CAF.	859
Chapter 30. Programming for the call attachment facility	861
CAF capabilities and requirements	861
CAF capabilities	861
Task capabilities	862
Programming language	862
Tracing facility	862
Program preparation	862
CAF requirements.	863
Program size	863
Use of LOAD	863
Using CAF in IMS batch	863
Run environment	863
Running DSN applications under CAF	863
How to use CAF	864
Summary of connection functions	866
Implicit connections	866
Accessing the CAF language interface	867
Explicit load of DSNALI.	867
Link-editing DSNALI.	868
General properties of CAF connections.	868
Task termination	868
DB2 abend	869
CAF function descriptions	869
Register conventions	869
Call DSNALI parameter list	869
CONNECT: Syntax and usage.	871
OPEN: Syntax and usage	875
CLOSE: Syntax and usage	877
DISCONNECT: Syntax and usage	878
TRANSLATE: Syntax and usage	880
Summary of CAF behavior.	881
Sample scenarios	882
A single task with implicit connections.	882
A single task with explicit connections	883
Several tasks	883
Exit routines from your application	883
Attention exit routines	883
Recovery routines	884
Error messages and dsnttrace	884
CAF return codes and reason codes	884

Program examples for CAF	885
Sample JCL for using CAF	886
Sample assembler code for using CAF	886
Loading and deleting the CAF language interface	886
Connecting to DB2 for CAF	886
Checking return codes and reason codes for CAF	888
Using dummy entry point DSNHLI for CAF	890
Variable declarations for CAF	891
Chapter 31. Programming for the Resource Recovery Services attachment facility	893
RRSAF capabilities and requirements	893
RRSAF capabilities	893
Task capabilities	894
Programming language	894
Tracing facility	894
Program preparation	894
RRSAF requirements	894
Program size	895
Use of LOAD	895
Commit and rollback operations	895
Run environment	895
How to use RRSAF	896
Summary of connection functions	896
Implicit connections	897
Accessing the RRSAF language interface	898
Explicit Load of DSNRLI	900
Link-editing DSNRLI	900
General properties of RRSAF connections	900
Task termination	901
DB2 abend	901
Summary of RRSAF behavior	902
RRSAF function descriptions	903
Register conventions	903
Parameter conventions for function calls	904
IDENTIFY: Syntax and usage	904
Usage	906
SWITCH TO: Syntax and usage	907
Usage	908
SIGNON: Syntax and usage	909
Usage	913
AUTH SIGNON: Syntax and usage	914
Usage	917
CONTEXT SIGNON: Syntax and usage.	918
Usage	921
SET_ID: Syntax and usage	922
Usage	923
SET_CLIENT_ID: Syntax and usage	923
Usage	925
CREATE THREAD: Syntax and usage	926
Usage	927
TERMINATE THREAD: Syntax and usage.	928
Usage	929
TERMINATE IDENTIFY: Syntax and usage	930
Usage	930
TRANSLATE: Syntax and usage	931
Usage	932
RRSAF connection examples	932
Example of a single task.	932
Example of multiple tasks	933
Example of calling SIGNON to reuse a DB2 thread.	933
Example of switching DB2 threads between tasks	933

RRSAF return codes and reason codes	934
Program examples for RRSAF	935
Sample JCL for using RRSAF	935
Loading and deleting the RRSAF language interface	935
Using dummy entry point DSNHLI for RRSAF	935
Connecting to DB2 for RRSAF.	936
 Chapter 32. CICS-specific programming techniques	939
Controlling the CICS attachment facility from an application	939
Improving thread reuse	939
Detecting whether the CICS attachment facility is operational	939
 Chapter 33. WebSphere MQ with DB2	941
WebSphere MQ messages	941
WebSphere MQ message handling	941
# WebSphere MQ message handling with the MQI	942
WebSphere MQ message handling with the AMI	943
# WebSphere MQ functions and stored procedures	944
Commit environment for AMI-based DB2 MQ functions and stored procedures	948
Single-phase commit in WebSphere MQ	948
Two-phase commit in WebSphere MQ	949
# DB2 MQ tables	949
# Converting applications to use the MQI functions	958
How to use WebSphere MQ functions	959
Basic messaging	959
Sending messages with WebSphere MQ	960
Retrieving messages	961
Application-to-application connectivity	962
# Asynchronous messaging in DB2 UDB for z/OS and OS/390	966
# MQListener in DB2 for OS/390 and z/OS	967
# Configuring and running MQListener in DB2 UDB for OS/390 and z/OS	968
# Configuring MQListener to run in the DB2 environment	968
# Configuring Websphere MQ for MQListener	969
# Configuring MQListener tasks.	971
# MQListener error processing	972
# Creating a sample stored procedure to use with MQListener	973
# MQListener examples	974
 Chapter 34. Using DB2 as a web services consumer and provider	977
DB2 as a web services consumer	977
The SOAPHTTPV and SOAPHTTPC user-defined functions.	977
# The SOAPHTTPNV and SOAPHTTPNC user-defined functions	978
SQLSTATes for DB2 as a web services consumer	979
DB2 as a web services provider	980
 Chapter 35. Programming techniques: Questions and answers	983
Providing a unique key for a table	983
Scrolling through previously retrieved data	983
Using a scrollable cursor	983
Using a ROWID or identity column	984
Scrolling through a table in any direction	985
Updating data as it is retrieved from the database	986
Updating previously retrieved data	986
Updating thousands of rows	986
Retrieving thousands of rows	987
Using SELECT *	987
Optimizing retrieval for a small set of rows	987
Adding data to the end of a table	988
Translating requests from end users into SQL statements	988
Changing the table definition	988
Storing data that does not have a tabular format	989

Finding a violated referential or check constraint	989

Chapter 24. Coding dynamic SQL in application programs

Before you decide to use dynamic SQL, you should consider whether using static SQL or dynamic SQL is the best technique for your application.

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.

Dynamic SQL prepares and executes the SQL statements within a program, while the program is running. Four types of dynamic SQL are:

- **Interactive SQL**
A user enters SQL statements through SPUFI. 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, but like dynamic statements, they are prepared at run time. DB2 processes deferred embedded SQL statements with bind-time rules. For example, DB2 uses the authorization ID and qualifier determined at bind time as the plan or package owner. Deferred embedded SQL statements are used for DB2 private protocol access to remote data.
- **Dynamic SQL executed through ODBC 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. See *DB2 ODBC Guide and Reference* for information about ODBC.

This chapter contains the following sections:

- “Choosing between static and dynamic SQL”
- “Caching dynamic SQL statements” on page 596
- “Limiting dynamic SQL with the resource limit facility” on page 601
- “Choosing a host language for dynamic SQL applications” on page 603
- “Dynamic SQL for non-SELECT statements” on page 603
- “Dynamic SQL for fixed-list SELECT statements” on page 610
- “Dynamic SQL for varying-list SELECT statements” on page 613

Choosing between static and dynamic SQL

This section contains the following information to help you decide whether you should use dynamic SQL statements in your application:

- “Flexibility of static SQL with host variables” on page 594
- “Flexibility of dynamic SQL” on page 594
- “Limitations of dynamic SQL” on page 594
- “Dynamic SQL processing” on page 594

- “Performance of static and dynamic SQL” on page 595

Flexibility of static SQL with host variables

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 DSN8810.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.

One example of such a program is the DB2 Query Management Facility (DB2 QMF), which provides an alternative interface to DB2 that accepts almost any SQL statement. SPUFI is another example; it accepts SQL statements from an input data set, and then processes and executes them dynamically.

Limitations of dynamic SQL

You cannot use some of the SQL statements dynamically. For information about which DB2 SQL statements you can dynamically prepare, see the table in Appendix H, “Characteristics of SQL statements in DB2 UDB for z/OS,” on page 1113.

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

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 which bind option you specify: REOPT(NONE), REOPT(ONCE), or REOPT(ALWAYS). REOPT(NONE) is the default.

If you specify REOPT(NONE), DB2 determines the access path at bind time, just as it does when there are no input variables.

DB2 ignores REOPT(ONCE) for static SQL statements because DB2 can cache only dynamic SQL statements

If you specify REOPT(ALWAYS), DB2 determines the access path at bind time and again at run time, using the values in these types of input variables:

- Host variables
- Parameter markers
- Special registers

This means that DB2 must spend extra time determining the access path for statements at run time, but if DB2 determines a significantly better access path using the variable values, you might see an overall performance improvement. In general, using REOPT(ALWAYS) can make static SQL statements with input variables perform like dynamic SQL statements with constants. For more information about using REOPT(ALWAYS) to change access paths, see “Using host variables efficiently” on page 759.

Dynamic SQL statements

For dynamic SQL statements, DB2 determines the access path at run time, when the statement is prepared. This 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. See “Performance of static and dynamic SQL” on page 595 for more information.

Dynamic SQL statements with input host variables: When you bind applications that contain dynamic SQL statements with input host variables, use either the REOPT(ALWAYS) or REOPT(ONCE) 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 ensures the best access path for a statement, but using REOPT(ALWAYS) can increase the cost of frequently used dynamic SQL statements.

Use REOPT(ONCE) when you are using the dynamic statements cache. DB2 determines 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 cost of frequently used dynamic SQL statements that contain input host variables.

You should code your PREPARE statements to minimize overhead. With both 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.

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.

Caching dynamic SQL statements

As the DB2 ability to optimize SQL has improved, the cost of preparing a dynamic SQL statement has grown. Applications that use dynamic SQL might be forced to pay this cost more than once. When an application performs a commit operation, it must issue another PREPARE statement if that SQL statement is to be executed

again. For a SELECT statement, the ability to declare a cursor WITH HOLD provides some relief but requires that the cursor be open at the commit point. WITH HOLD also causes some locks to be held for any objects that the prepared statement is dependent on. Also, WITH HOLD offers no relief for SQL statements that are not SELECT statements.

DB2 can save prepared dynamic statements in a cache. The cache is a dynamic statement cache pool that all application processes can use to save and retrieve prepared dynamic statements. After an SQL statement has been prepared and is automatically saved in the cache, subsequent prepare requests for that same SQL statement can avoid the costly preparation process by using the statement that is in the cache. Statements that are saved in the cache can be shared among different threads, plans, or packages.

Example: Assume that your application program contains a dynamic SQL statement, STMT1, which is prepared and executed multiple times. If you are using the dynamic statement cache when STMT1 is prepared for the first time, it is placed in the cache. When your application program encounters the identical PREPARE statement for STMT1, DB2 uses the already prepared STMT1 that is saved in the dynamic statement cache. The following example shows the identical STMT1 that might appear in your application program:

```
PREPARE STMT1 FROM ...      Statement is prepared and the prepared
EXECUTE STMT1               statement is put in the cache.
COMMIT
:
PREPARE STMT1 FROM ...      Identical statement. DB2 uses the prepared
EXECUTE STMT1               statement from the cache.
COMMIT
:
```

Eligible statements: The following SQL statements can be saved in the cache:

```
SELECT
UPDATE
INSERT
DELETE
```

Distributed and local SQL statements are eligible to be saved. Prepared, dynamic statements that use DB2 private protocol access are also eligible to be saved.

Restrictions: Even though static statements that use DB2 private protocol access are dynamic at the remote site, those statements can not be saved in the cache.

Statements in plans or packages that are bound with REOPT(ALWAYS) can not be saved in the cache. Statements in plans and packages that are bound with REOPT(ONCE) can be saved in the cache. See “How bind options REOPT(ALWAYS) and REOPT(ONCE) affect dynamic SQL” on page 625 for more information about REOPT(ALWAYS) and REOPT(ONCE).

Prepared statements cannot be shared among data sharing members. Because each member has its own EDM pool, a cached statement on one member is not available to an application that runs on another member.

Using the dynamic statement cache

To enable the dynamic statement cache to save prepared statements, specify YES on the CACHE DYNAMIC SQL field of installation panel DSNTIP8. See Part 2 of *DB2 Installation Guide* for more information.

Conditions for statement sharing

Suppose that S1 and S2 are source statements, and P1 is the prepared version of S1. P1 is in the dynamic statement cache.

The following conditions must be met before DB2 can use statement P1 instead of preparing statement S2:

- S1 and S2 must be identical. The statements must pass a character by character comparison and must be the same length. If the PREPARE statement for either statement contains an ATTRIBUTES clause, DB2 concatenates the values in the ATTRIBUTES clause to the statement string before comparing the strings. That is, if A1 is the set of attributes for S1 and A2 is the set of attributes for S2, DB2 compares S1 || A1 to S2 || A2.

If the statement strings are not identical, DB2 cannot use the statement in the cache.

For example, assume that S1 and S2 are specified as follows:

```
'UPDATE EMP SET SALARY=SALARY+50'
```

In this case, DB2 can use P1 instead of preparing S2.

However, assume that S1 is specified as follows:

```
'UPDATE EMP SET SALARY=SALARY+50'
```

Assume also that S2 is specified as follows:

```
'UPDATE EMP SET SALARY=SALARY+50 '
```

In this case, DB2 cannot use P1 for S2. DB2 prepares S2 and saves the prepared version of S2 in the cache.

- The authorization ID that was used to prepare S1 must be used to prepare S2:
 - When a plan or package has run behavior, the authorization ID is the current SQLID value.

For secondary authorization IDs:

- The application process that searches the cache must have the same secondary authorization ID list as the process that inserted the entry into the cache or must have a superset of that list.
- If the process that originally prepared the statement and inserted it into the cache used one of the privileges held by the primary authorization ID to accomplish the prepare, that ID must either be part of the secondary authorization ID list of the process searching the cache, or it must be the primary authorization ID of that process.
- When a plan or package has bind behavior, the authorization ID is the plan owner's ID. For a DDF server thread, the authorization ID is the package owner's ID.
- When a package has define behavior, then the authorization ID is the user-defined function or stored procedure owner.
- When a package has invoke behavior, then the authorization ID is the authorization ID under which the statement that invoked the user-defined function or stored procedure executed.

For an explanation of bind, run, define, and invoke behavior, see "Using DYNAMICRULES to specify behavior of dynamic SQL statements" on page 502.

- When the plan or package that contains S2 is bound, the values of these bind options must be the same as when the plan or package that contains S1 was bound:

CURRENTDATA

DYNAMICRULES
ISOLATION
SQLRULES
QUALIFIER

- When S2 is prepared, the values of the following special registers must be the same as when S1 was prepared:

CURRENT DEGREE
CURRENT RULES
CURRENT PRECISION
CURRENT REFRESH AGE
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

Exception: If you set the CACHEDYN_FREELOCAL subsystem parameter to 1 and a storage shortage occurs, DB2 frees the cached dynamic statements. In this case, DB2 cannot use P1 instead of preparing statement S2, because P1 no longer exists in the statement cache.

Using the statement cache table

The EXPLAIN STATEMENT CACHE ALL statement puts statement cache information into the DSN_STATEMENT_CACHE_TABLE table for all queries in the cache that qualify based on the user's SQLID. If the SQLID has SYSADM authority, all statements are put into the table.

The EXPLAIN STATEMENT CACHE ALL statement is useful because it allows you to see all of the statements in the cache and to concentrate on improving specific statements for specific performance characteristics. The statement can be used as part of an online monitoring strategy because EXPLAIN STATEMENT CACHE ALL provides DSN_STATEMENT_CACHE_TABLE with a snapshot of the cache.

Example: If you are concerned about CPU time, you can select from the STAT_CPU column in the DSN_STATEMENT_CACHE_TABLE table to identify the queries that consume the most CPU time. Then you can work to improve the performance of those specific queries. For information about creating the DSN_STATEMENT_CACHE_TABLE table and about the columns in the table, see *DB2 SQL Reference*.

Keeping prepared statements after commit points

The bind option KEEP_DYNAMIC(YES) lets you hold dynamic statements past a commit point for an application process. An application can issue a PREPARE for a statement once and omit subsequent PREPAREs for that statement. Figure 179 illustrates an application that is written to use KEEP_DYNAMIC(YES).

```
PREPARE STMT1 FROM ...      Statement is prepared.
EXECUTE STMT1
COMMIT
:
EXECUTE STMT1                Application does not issue PREPARE.
COMMIT
:
EXECUTE STMT1                Again, no PREPARE needed.
COMMIT
```

Figure 179. Writing dynamic SQL to use the bind option KEEP_DYNAMIC(YES)

To understand how the `KEEPDYNAMIC` bind option works, you need to differentiate between the executable form of a dynamic SQL statement, which is the prepared statement, and the character string form of the statement, which is the statement string.

Relationship between `KEEPDYNAMIC(YES)` and statement caching: When the dynamic statement cache is not active, and you run an application bound with `KEEPDYNAMIC(YES)`, DB2 saves only the statement string for a prepared statement after a commit operation. On a subsequent `OPEN`, `EXECUTE`, or `DESCRIBE`, DB2 must prepare the statement again before performing the requested operation. Figure 180 illustrates this concept.

PREPARE STMT1 FROM ...	Statement is prepared and put in memory.
EXECUTE STMT1	
COMMIT	
:	
EXECUTE STMT1	Application does not issue PREPARE.
COMMIT	DB2 prepares the statement again.
:	
EXECUTE STMT1	Again, no PREPARE needed.
COMMIT	

Figure 180. Using `KEEPDYNAMIC(YES)` when the dynamic statement cache is not active

When the dynamic statement cache is active, and you run an application bound with `KEEPDYNAMIC(YES)`, DB2 retains a copy of both the prepared statement and the statement string. The prepared statement is cached locally for the application process. In general, the statement is globally cached in the EDM pool, to benefit other application processes. If the application issues an `OPEN`, `EXECUTE`, or `DESCRIBE` after a commit operation, the application process uses its local copy of the prepared statement to avoid a prepare and a search of the cache. Figure 181 illustrates this process.

PREPARE STMT1 FROM ...	Statement is prepared and put in memory.
EXECUTE STMT1	
COMMIT	
:	
EXECUTE STMT1	Application does not issue PREPARE.
COMMIT	DB2 uses the prepared statement in memory.
:	
EXECUTE STMT1	Again, no PREPARE needed.
COMMIT	DB2 uses the prepared statement in memory.
:	
PREPARE STMT1 FROM ...	Statement is prepared and put in memory.

Figure 181. Using `KEEPDYNAMIC(YES)` when the dynamic statement cache is active

The local instance of the prepared SQL statement is kept in `ssnmDBM1` storage until one of the following occurs:

- The application process ends.
- A rollback operation occurs.
- The application issues an explicit `PREPARE` statement with the same statement name.

If the application does issue a `PREPARE` for the same SQL statement name that has a kept dynamic statement associated with it, the kept statement is discarded and DB2 prepares the new statement.

- The statement is removed from memory because the statement has not been used recently, and the number of kept dynamic SQL statements reaches the subsystem default as set during installation.

Handling implicit prepare errors: If a statement is needed during the lifetime of an application process, and the statement has been removed from the local cache, DB2 might be able to retrieve it from the global cache. If the statement is not in the global cache, DB2 must implicitly prepare the statement again. The application does not need to issue a PREPARE statement. However, if the application issues an OPEN, EXECUTE, or DESCRIBE for the statement, the application must be able to handle the possibility that DB2 is doing the prepare *implicitly*. Any error that occurs during this prepare is returned on the OPEN, EXECUTE, or DESCRIBE.

How KEEP_DYNAMIC affects applications that use distributed data: If a requester does not issue a PREPARE after a COMMIT, the package at the DB2 UDB for z/OS server must be bound with KEEP_DYNAMIC(YES). If both requester and server are DB2 UDB for z/OS subsystems, the DB2 requester assumes that the KEEP_DYNAMIC value for the package at the server is the same as the value for the plan at the requester.

The KEEP_DYNAMIC option has performance implications for DRDA clients that specify WITH HOLD on their cursors:

- If KEEP_DYNAMIC(NO) is specified, a separate network message is required when the DRDA client issues the SQL CLOSE for the cursor.
- If KEEP_DYNAMIC(YES) is specified, the DB2 UDB for z/OS server automatically closes the cursor when SQLCODE +100 is detected, which means that the client does not have to send a separate message to close the held cursor. This reduces network traffic for DRDA applications that use held cursors. It also reduces the duration of locks that are associated with the held cursor.

Using RELEASE(DEALLOCATE) with KEEP_DYNAMIC(YES): See 409 for information about interactions between bind options RELEASE(DEALLOCATE) and KEEP_DYNAMIC(YES).

Considerations for data sharing: If one member of a data sharing group has enabled the cache but another has not, and an application is bound with KEEP_DYNAMIC(YES), DB2 must implicitly prepare the statement again if the statement is assigned to a member without the cache. This can mean a slight reduction in performance.

Limiting dynamic SQL with the resource limit facility

The resource limit facility (or governor) limits the amount of CPU time an SQL statement can take, which prevents SQL statements from making excessive requests. The predictive governing function of the resource limit facility provides an estimate of the processing cost of SQL statements before they run. To predict the cost of an SQL statement, you execute EXPLAIN to put information about the statement cost in DSN_STATEMENT_TABLE. See “Estimating a statement’s cost” on page 842 for information about creating, populating, and interpreting the contents of DSN_STATEMENT_TABLE.

The governor controls only the dynamic SQL manipulative statements SELECT, UPDATE, DELETE, and INSERT. Each dynamic SQL statement used in a program is subject to the same limits. The limit can be a *reactive* governing limit or a *predictive* governing limit. If the statement exceeds a reactive governing limit, the

statement receives an error SQL code. If the statement exceeds a predictive governing limit, it receives a warning or error SQL code. “Writing an application to handle predictive governing” explains more about predictive governing SQL codes.

Your system administrator can establish the limits for individual plans or packages, for individual users, or for all users who do not have personal limits.

Follow the procedures defined by your location for adding, dropping, or modifying entries in the resource limit specification table. For more information about the resource limit specification tables, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Writing an application to handle reactive governing

When a dynamic SQL statement exceeds a reactive governing threshold, the application program receives SQLCODE -905. The application must then determine what to do next.

If the failed statement involves an SQL cursor, the cursor’s position remains unchanged. The application can then close that cursor. All other operations with the cursor do not run and the same SQL error code occurs.

If the failed SQL statement does not involve a cursor, then all changes that the statement made are undone before the error code returns to the application. The application can either issue another SQL statement or commit all work done so far.

Writing an application to handle predictive governing

If your installation uses predictive governing, you need to modify your applications to check for the +495 and -495 SQLCODEs that predictive governing can generate after a PREPARE statement executes. The +495 SQLCODE in combination with deferred prepare requires that DB2 do some special processing to ensure that existing applications are not affected by this new warning SQLCODE.

For information about setting up the resource limit facility for predictive governing, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Handling the +495 SQLCODE

If your requester uses deferred prepare, the presence of parameter markers determines when the application receives the +495 SQLCODE. When parameter markers are present, DB2 cannot do PREPARE, OPEN, and FETCH processing in one message. If SQLCODE +495 is returned, no OPEN or FETCH processing occurs until your application requests it.

- If there are parameter markers, the +495 is returned on the OPEN (not the PREPARE).
- If there are no parameter markers, the +495 is returned on the PREPARE.

Normally with deferred prepare, the PREPARE, OPEN, and first FETCH of the data are returned to the requester. For a predictive governor warning of +495, you would ideally like to have the option to choose beforehand whether you want the OPEN and FETCH of the data to occur. For down-level requesters, you do not have this option.

Using predictive governing and down-level DRDA requesters

If SQLCODE +495 is returned to the requester, OPEN processing continues but the first block of data is not returned with the OPEN. Thus, if your application does not continue with the query, you have already incurred the performance cost of OPEN processing.

Using predictive governing and enabled requesters

If your application does not defer the prepare, SQLCODE +495 is returned to the requester and OPEN processing does not occur.

If your application does defer prepare processing, the application receives the +495 at its usual time (OPEN or PREPARE). If you have parameter markers with deferred prepare, you receive the +495 at OPEN time as you normally do. However, an additional message is exchanged.

Recommendation: Do not use deferred prepare for applications that use parameter markers and that are predictively governed at the server side.

Choosing a host language for dynamic SQL applications

Programs that use dynamic SQL are usually written in assembler, C, PL/I, REXX, and COBOL. 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.

All SQL in REXX programs is dynamic SQL. For information about how to write SQL REXX applications, see “Coding SQL statements in a REXX application” on page 249

Most of the examples in this section are in PL/I. “Using dynamic SQL in COBOL” on page 627 shows techniques for using COBOL. 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.

Dynamic SQL for non-SELECT statements

The easiest way to use dynamic SQL is not to use SELECT statements dynamically. Because you do not need to dynamically allocate any main storage, you can write your program in any host language, including COBOL and Fortran. For a sample program written in C that contains dynamic SQL with non-SELECT statements, see Figure 279 on page 1044.

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:
 - “Dynamic execution using EXECUTE IMMEDIATE”
 - “Dynamic execution using PREPARE and EXECUTE” on page 605.
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. See “Checking the execution of SQL statements” on page 91 for information about the SQLCA and the fields it contains.

Dynamic execution using EXECUTE IMMEDIATE

Suppose that you design a program to read SQL DELETE statements, similar to these, from a terminal:

```
DELETE FROM DSN8810.EMP WHERE EMPNO = '000190'  
DELETE FROM DSN8810.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, 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.
- In PL/I, it can be a fixed- or varying-length string variable, or any PL/I expression that evaluates to a character string.

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.

For more information about declaring character-string host variables, see Chapter 9, “Embedding SQL statements in host languages,” on page 143.

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 DBLOB host variable: You declare CLOB and DBCLOB host variables according to the rules described in “Declaring LOB host variables and LOB locators” on page 300.

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.

Dynamic execution using PREPARE and EXECUTE

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 DSN8810.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 DSN8810.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

Using 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 function. When the statement executes, DB2 converts the host variable to the data type in the CAST function. A parameter marker that you include in a CAST function is called a *typed* parameter marker. A parameter marker without a CAST function 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 typed parameter markers. See Chapter 5 of *DB2 SQL Reference* for rules for using untyped or typed parameter markers.

Example using parameter markers: Suppose that you want to prepare this statement:

```
DELETE FROM DSN8810.EMP WHERE EMPNO = :EMP;
```

You need to prepare a string like this:

```
DELETE FROM DSN8810.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. For more information about declaring the host variable, see “Declaring the host variable” on page 604.

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 DSN8810.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 allows 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 section 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 DSN8810.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).

Dynamic execution of a multiple-row INSERT statement

Suppose that you want to repeatedly execute a multiple-row INSERT statement using a list of activity IDs, activity keywords, and activity descriptions that are provided by the user. A **static** SQL INSERT statement that inserts multiple rows of data into the activity table looks similar to the following statement:

```
EXEC SQL
    INSERT INTO DSN8810.ACT
    VALUES (:hva_actno, :hva_actkwd, :hva_actdesc)
    FOR :num_rows ROWS;
```

You might be entering the rows of data into different tables or entering different numbers of rows, so you want to construct the INSERT statement dynamically. This section describes the following methods to execute a multiple-row INSERT 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

Using EXECUTE with host variable arrays

You can use the CAST function to explicitly assign a type to parameter markers that represent host variable arrays. For the activity table, the string for the INSERT statement that is to be prepared looks like this:

```
INSERT INTO DSN8810.ACT
    VALUES (CAST(? AS SMALLINT), CAST(? AS CHAR(6)), CAST(? AS VARCHAR(20)))
```

You must specify the FOR *n* ROWS clause on the EXECUTE statement.

Preparing and executing the statement: The code to prepare and execute the INSERT statement looks like this:

```
/* Copy the INSERT string into the host variable sqlstmt */
strcpy(sqlstmt, "INSERT INTO DSN8810.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;
```

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 INSERT statement. You can vary the number of rows, specified by num_rows in the example, without needing to prepare the INSERT statement again.

Using EXECUTE with a descriptor

You can use an SQLDA structure to specify data types and other information about parameters markers. The string for the INSERT statement that is to be prepared looks like this:

```
INSERT INTO DSN8810.ACT VALUES (?, ?, ?)
```

You must specify the FOR *n* ROWS clause on the EXECUTE statement.

Setting the fields in the SQLDA: Assume that your program includes the standard SQLDA structure declaration and declarations for the program variables that point to the SQLDA structure. Before the INSERT statement is prepared and executed, you must set the fields in the SQLDA structure for your INSERT statement. For C application programs, the code to set the fields looks like this:

```

# 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 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 $SQLN \times 44 + 16$, 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.

For more information about the SQLDA, see “Dynamic SQL for varying-list SELECT statements” on page 613. For a complete layout of the SQLDA and the descriptions given by the INCLUDE statement, see Appendix E of *DB2 SQL Reference*.

Preparing and executing the statement: The code to prepare and execute the INSERT statement looks like this:

```

/* Copy the INSERT string into the host variable sqlstmt */
strcpy(sqlstmt, "INSERT INTO DSN8810.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;

```

| The host variable in the USING clause of the EXECUTE statement names the
| SQLDA that describes the parameter markers in the INSERT statement.

Using DESCRIBE INPUT to put parameter information in an SQLDA

You can use the DESCRIBE INPUT statement to let DB2 put the data type information for parameter markers in an SQLDA.

Before you execute DESCRIBE INPUT, you must allocate an SQLDA with enough instances of SQLVAR to represent all parameter markers in the SQL statements you want to describe.

After you execute DESCRIBE INPUT, you code the application in the same way as any other application in which you execute a prepared statement using an SQLDA. First, you obtain the addresses of the input host variables and their indicator variables and insert those addresses into the SQLDATA and SQLIND fields. Then you execute the prepared SQL statement.

Example using the SQLDA: Suppose that you want to execute this statement dynamically:

```
DELETE FROM DSN8810.EMP WHERE EMPNO = ?
```

The code to set up an SQLDA, obtain parameter information using DESCRIBE INPUT, and execute the statement looks like this:

```

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 DSN8810.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;

```

Dynamic SQL for fixed-list SELECT statements

A *fixed-list* SELECT statement returns rows that contain a known number of values of a known type. When you use one, you know in advance exactly what kinds of host variables you need to declare to store the results. (The contrasting situation, in which you do not know in advance what host-variable structure you might need, is in the section “Dynamic SQL for varying-list SELECT statements” on page 613.)

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.

For a sample program that is written in C and that illustrates dynamic SQL with fixed-list SELECT statements, see Figure 279 on page 1044.

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 as described under “Dynamic SQL for non-SELECT statements” on page 603.
3. Declare a cursor for the statement name as described in “Declaring a cursor for the statement name.”
4. Prepare the statement, as described in “Preparing the statement.”
5. Open the cursor, as described in “Opening the cursor” on page 612.
6. Fetch rows from the result table, as described in “Fetching rows from the result table” on page 612.
7. Close the cursor, as described in “Closing the cursor” on page 612.
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 DSN8810.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. The following sections describe how to do those steps.

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 described under “Dynamic SQL for varying-list SELECT statements” on page 613.

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;
```

Dynamic SQL for varying-list SELECT statements

A *varying-list* SELECT statement returns rows containing an unknown number of values of unknown type. When you use one, you do not know in advance exactly what kinds of host variables you need to declare in order to store the results. (For the much simpler situation, in which you do know, see “Dynamic SQL for fixed-list SELECT statements” on page 610.) 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

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 procedure.
2. Load the input SQL statement into a data area.
Those first two steps are exactly the same as described under “Dynamic SQL for non-SELECT statements” on page 603; the next step is new:
3. Prepare and execute the statement. This step is more complex than for fixed-list SELECTs. For details, see “Preparing a varying-list SELECT statement” and “Executing a varying-list SELECT statement dynamically” on page 623. It involves the following steps:
 - a. Include an SQLDA (SQL descriptor area).
DB2 performs this step for a REXX procedure.
 - 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 procedure.
 - 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.

For a complete layout of the SQLDA and the descriptions given by INCLUDE statements, see Appendix E of *DB2 SQL Reference*.

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.

Table 75 shows the minimum number of SQLVAR instances you need for a result table that contains n columns.

Table 75. 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. Figure 182 shows the contents of the minimum SQLDA in use.

Header →	SQLDAID	SQLDABC	100	SQLD

Figure 182. 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 Table 75 on page 614, 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

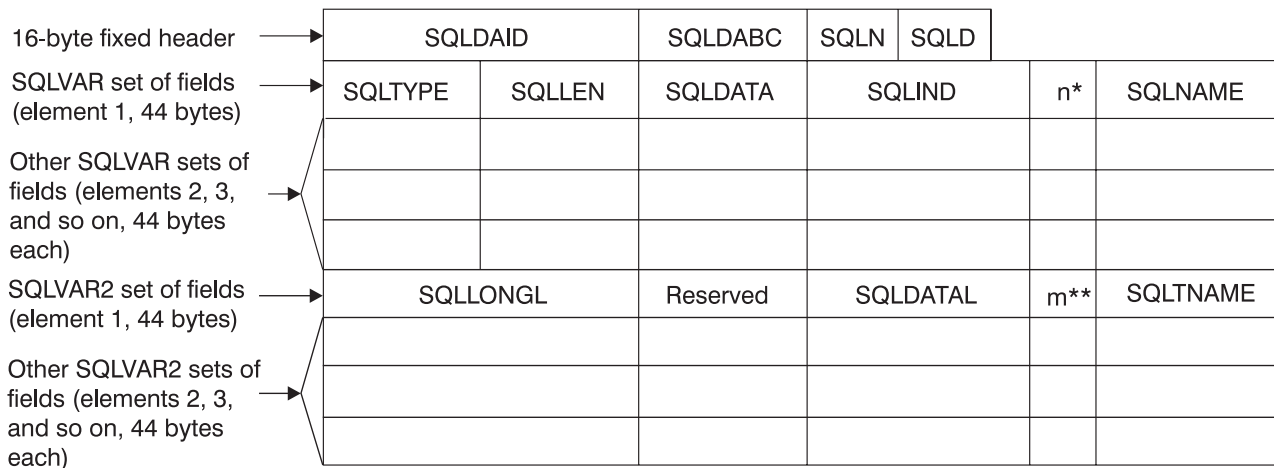
To find out if the statement is a SELECT, your program can query the SQLD field in MINSQLDA. If the field contains 0, the statement is not a SELECT, the statement is already prepared, and your program can execute it. If no parameter markers are in the statement, you can use:

```
EXEC SQL EXECUTE STMT;
```

(If the statement does contain parameter markers, you must use an SQL descriptor area; for instructions, see “Executing arbitrary statements with parameter markers” on page 624.)

Acquiring storage for a second SQLDA if needed

Now you can allocate storage for a second, full-size SQLDA; call it FULSQLDA. Figure 183 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 183. 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. See Appendix E of *DB2 SQL Reference* for details about the two SQLVAR formats. The number of SQLVAR and SQLVAR2 elements you need is in the SQLD field of MINSQLDA, and the total length you need for FULSQLDA ($16 + \text{SQLD} * 44$) 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.

Figure 184 shows an SQLDA that describes two columns that are not LOB columns or distinct type columns. See “Describing tables with LOB and distinct type columns” on page 621 for an example of describing a result table with 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 184. 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 Figure 185, 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.

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 185. SQL descriptor area after executing *DESCRIBE*

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. For information about SQLTYPE values, see Appendix E of *DB2 SQL Reference*.

Figure 186 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

FLDA CHAR(3)	FLDB CHAR(4)	Indicator variables (halfword)	
		FLDAI	FLDBI

Figure 186. SQL descriptor area after analyzing descriptions and acquiring storage

Figure 187 shows the SQLDA after your program executes a *FETCH* statement.

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

FLDA CHAR(3)	FLDB CHAR(4)	Indicator variables (halfword)	
E11	4502	FLDAI	FLDBI
		0	0

Figure 187. SQL descriptor area after executing *FETCH*

Table 76 describes the values in the descriptor area.

Table 76. Values inserted in the SQLDA

Value	Field	Description
SQLDA	SQLDAID	An "eye-catcher"
8816	SQLDABC	The size of the SQLDA in bytes (16 + 44 * 200)
200	SQLN	The number of occurrences of SQLVAR, set by the program

Table 76. Values inserted in the SQLDA (continued)

Value	Field	Description
200	SQLD	The number of occurrences of SQLVAR actually used by the DESCRIBE statement
452	SQLTYPE	The value of SQLTYPE in the first occurrence of SQLVAR. 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 CCSID 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 186 on page 618 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 “Changing the coded character set ID of host variables” on page 85 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:
 - Set the length field of SQLNAME to 8.
 - Set the first two bytes of the data field of SQLNAME to X'0000'.
 - 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 UDB 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 Figure 188.

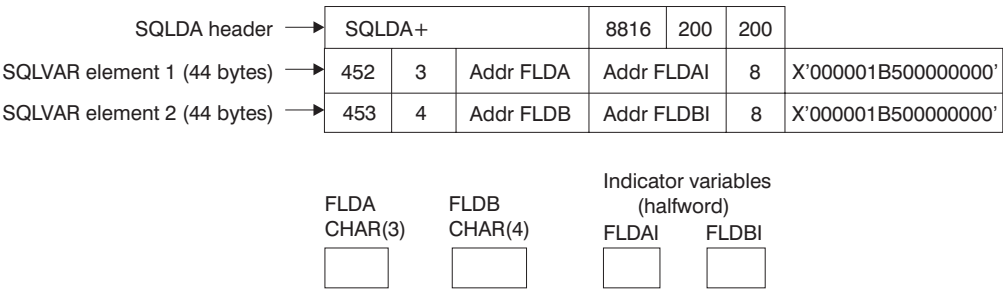


Figure 188. SQL descriptor area for retrieving data in ASCII CCSID 437

Using column labels

By default, DESCRIBE describes each column in the SQLNAME field by the column name. You can tell it to use column labels instead by writing:

```
EXEC SQL
  DESCRIBE STMT INTO :FULSQLDA USING LABELS;
```


#

In this case, SQLNAME contains nothing for a column with no label. If you prefer to use labels wherever they exist, but column names where there are no labels, write USING ANY. (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.)

You can also write USING BOTH to obtain the name and the label when both exist. However, to obtain both, you need a second set of occurrences of SQLVAR in FULSQLDA. The first set contains descriptions of all the columns using names; the second set contains descriptions using labels. This means that you must allocate a longer SQLDA for the second DESCRIBE statement ((16 + SQLD * 88 bytes) instead of (16 + SQLD * 44)). You must also put double the number of columns (SQLD * 2) in the SQLN field of the second SQLDA. Otherwise, if not enough space is available, DESCRIBE does not enter descriptions of any of the columns.

Describing tables with LOB and distinct type columns

In general, the steps you perform when you prepare an SQLDA to select rows from a table with LOB or distinct type columns are similar to the steps 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.

To illustrate this, 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 Figure 189.

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 189. 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 SQLLONGL 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.
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.

Figure 190 shows the contents of FULSQLDA after you fill in pointers to the storage locations.

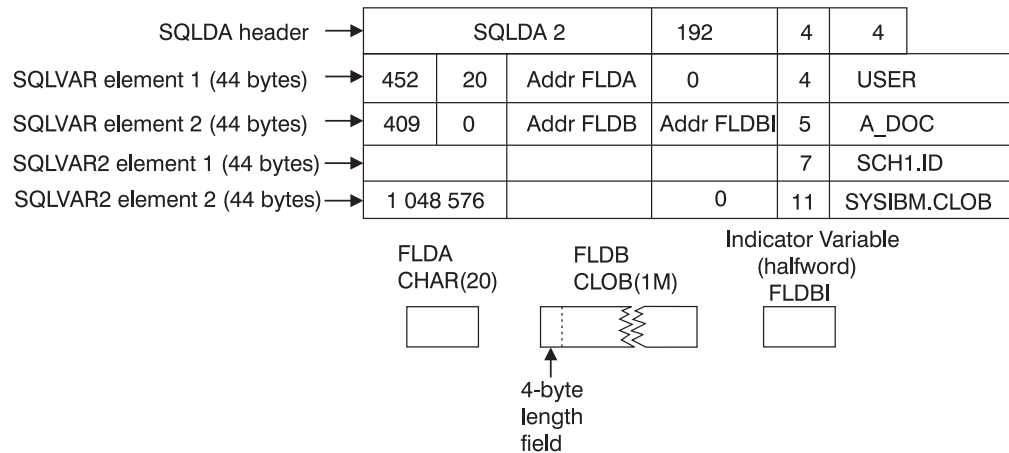


Figure 190. SQL descriptor area after analyzing CLOB and distinct type descriptions and acquiring storage

Figure 191 on page 623 shows the contents of FULSQLDA after you execute a FETCH statement.

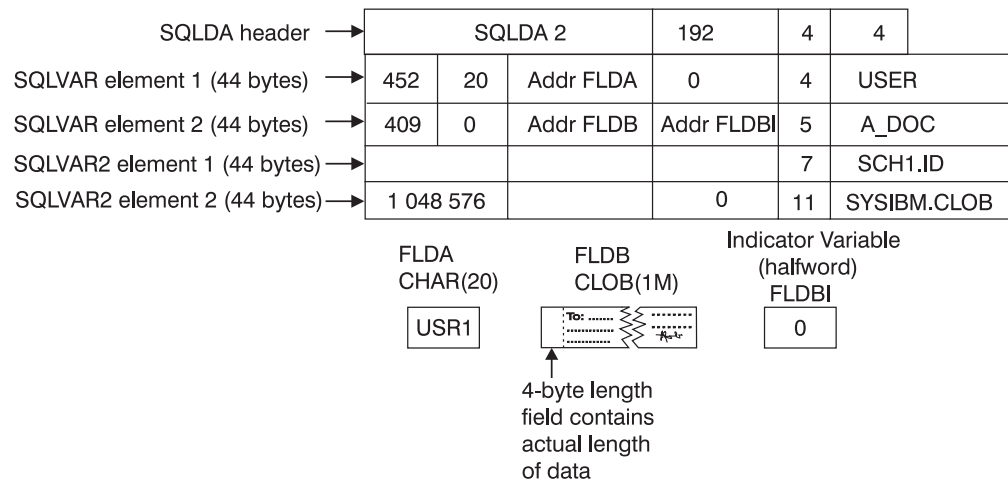


Figure 191. SQL descriptor area after executing *FETCH* on a table with CLOB and distinct type columns

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;
```

For cases when there are parameter markers, see “Executing arbitrary statements with parameter markers” on page 624 below.

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 185 on page 618.

Figure 187 on page 618 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 allows 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 DSN8810.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 fill in 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) 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(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.

Using dynamic SQL in COBOL

You can use all forms of dynamic SQL in all supported versions of COBOL. For a detailed description and a working example of the method, see “Sample COBOL dynamic SQL program” on page 1031.

Chapter 25. Using stored procedures for client/server processing

This chapter covers the following topics:

- “Introduction to stored procedures”
- “An example of a simple stored procedure” on page 630
- “Setting up the stored procedures environment” on page 634
- “Writing and preparing an external stored procedure” on page 641
- “Writing and preparing an SQL procedure” on page 657
- “Writing and preparing an application to use stored procedures” on page 681
- “Running a stored procedure” on page 719
- “Testing a stored procedure” on page 725

This chapter contains information that applies to all stored procedures and specific information about stored procedures in languages other than Java. For information about writing, preparing, and running Java stored procedures, see *DB2 Application Programming Guide and Reference for Java*.

Introduction to stored procedures

A *stored procedure* is a compiled program, stored at a DB2 local or remote server, that can execute SQL statements. A typical stored procedure contains two or more SQL statements and some manipulative or logical processing in a host language. A client application program uses the SQL statement CALL to invoke the stored procedure.

Consider using stored procedures for a client/server application that does at least one of the following things:

- Executes 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.
- Accesses 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.
- Accesses 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.
- Creates a result set of rows to return to the client application.

Figure 192 on page 630 and Figure 193 on page 630 illustrate the difference between using stored procedures and not using stored procedures for processing in a client/server environment.

Figure 192 on page 630 shows processing without using stored procedures. A client application embeds SQL statements and communicates with the server separately

for each statement. This results in increased network traffic and processor costs.

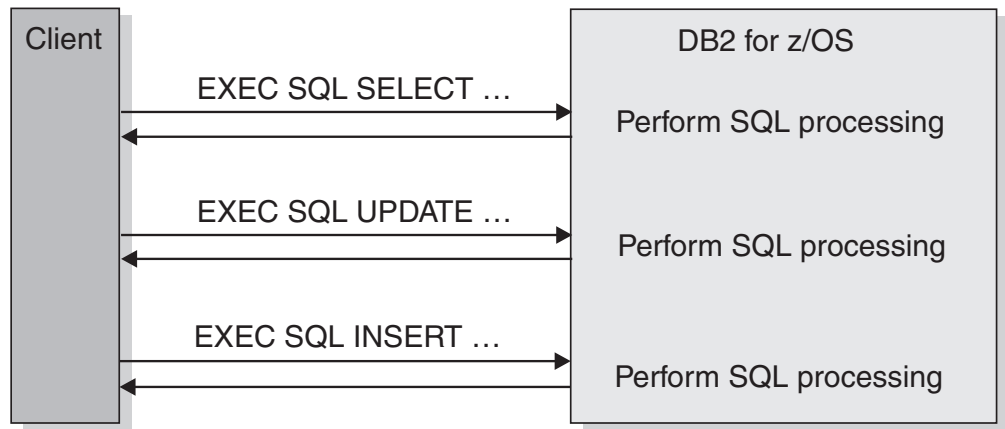


Figure 192. Processing without stored procedures

Figure 193 shows processing with stored procedures. The same series of SQL statements that are illustrated in Figure 192 uses a single send and receive operation, reducing network traffic and the cost of processing these statements.

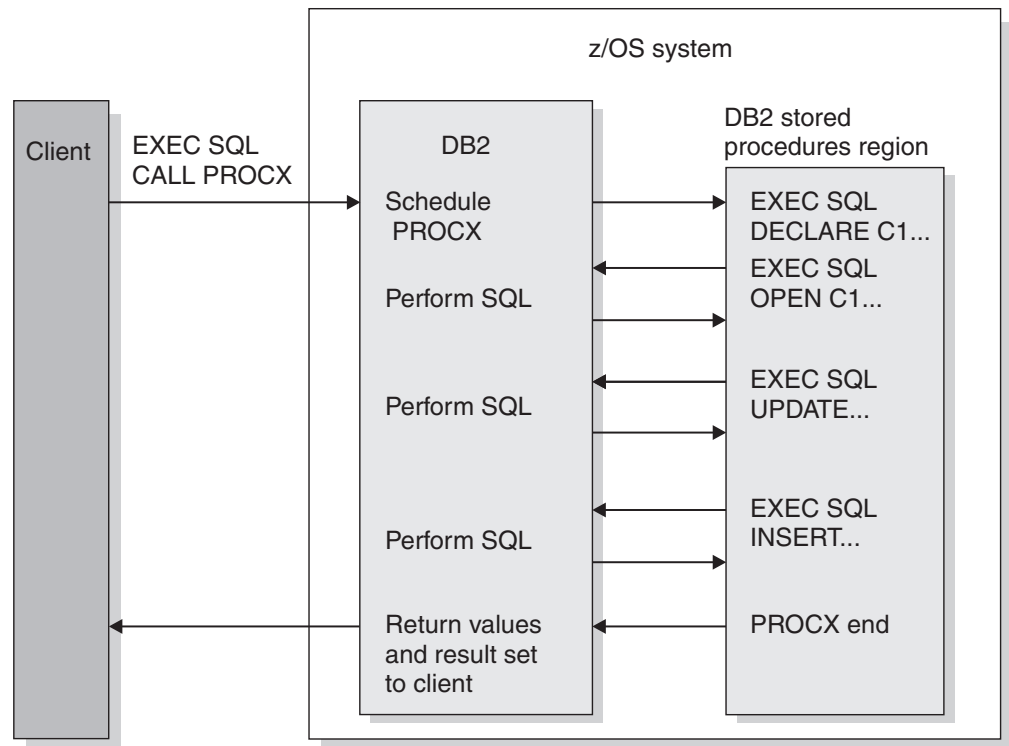


Figure 193. Processing with stored procedures

An example of a simple stored procedure

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 (DSN8810.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.
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.
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.

Figure 194 on page 632 illustrates the steps that are involved in executing this stored procedure.

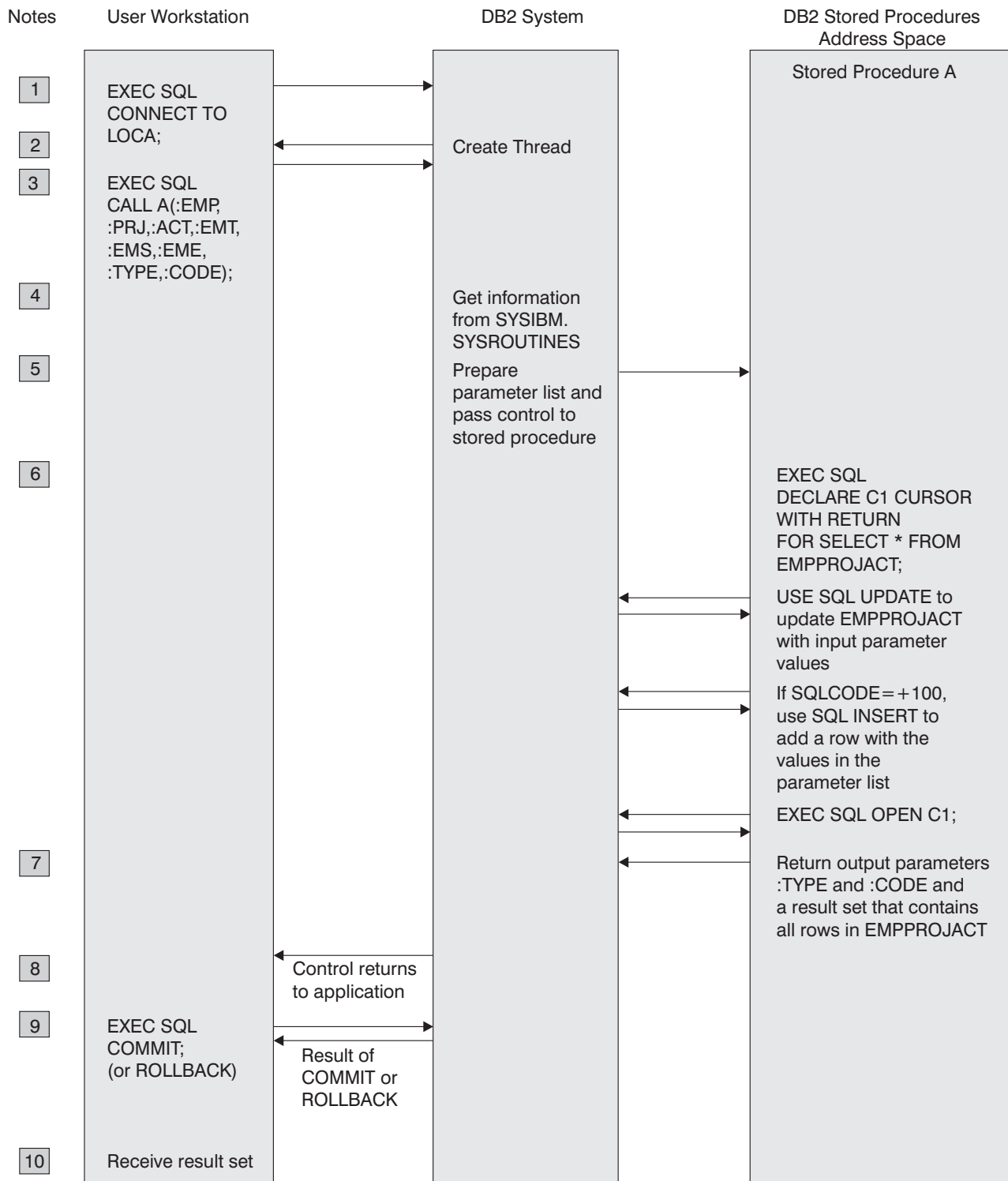


Figure 194. Stored procedure overview

Notes to Figure 194:

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 SYSROUTINES about stored procedure A. DB2 caches all rows in the table associated with A, so future references to A do not require I/O to the table.
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.

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.

Setting up the stored procedures environment

This section discusses the tasks that must be performed before stored procedures can run. Most of this information is for system administrators, but application programmers should read “Defining your stored procedure to DB2” on page 635. That section explains how to use the CREATE PROCEDURE statement to define a stored procedure to DB2.

The system administrator needs to perform these tasks to prepare the DB2 subsystem to run stored procedures:

- Move existing stored procedures to a WLM environment, or set up WLM environments for new stored procedures.

You can run only existing stored procedures in a DB2-established stored procedure address space; the support for this type of address space is being deprecated. If you are currently using DB2-established address spaces, see “Moving stored procedures to a WLM-established environment (for system administrators)” on page 640 for information about what needs to be done.

- Define JCL procedures for the stored procedures address spaces.

Member DSNTIJMV of data set DSN810.SDSNSAMP contains sample JCL procedures for starting WLM-established and DB2-established address spaces. If you enter a WLM procedure name or a DB2 procedure name in installation panel DSNTIPX, DB2 customizes a JCL procedure for you. See Part 2 of *DB2 Installation Guide* for details.

- For WLM-established address spaces, define WLM application environments for groups of stored procedures and associate a JCL startup procedure with each application environment.

See Part 5 (Volume 2) of *DB2 Administration Guide* for information about how to do this.

- If you plan to execute stored procedures that use the ODBA interface to access IMS databases, modify the startup procedures for the address spaces in which those stored procedures will run in the following way:
 - Add the data set name of the IMS data set that contains the ODBA callable interface code (usually IMS.RESLIB) to the end of the STEPLIB concatenation.
 - After the STEPLIB DD statement, add a DFSRESLB DD statement that names the IMS data set that contains the ODBA callable interface code.
- If you plan to execute LANGUAGE JAVA stored procedures, set up the JCL and install the software prerequisites, as described in *DB2 Application Programming Guide and Reference for Java*.
- Install Language Environment and the appropriate compilers.

See *z/OS Language Environment Customization* for information about installing Language Environment.

See “Language requirements for the stored procedure and its caller” on page 641 for minimum compiler and Language Environment requirements.

The system administrator needs to perform these tasks for each stored procedure:

- Be sure that the library in which the stored procedure resides is the STEPLIB concatenation of the startup procedure for the stored procedures address space.
- Use the CREATE PROCEDURE statement to define the stored procedure to DB2 and ALTER PROCEDURE to modify the definition.

See “Defining your stored procedure to DB2” on page 635 for details.

- Perform security tasks for the stored procedure.

See Part 3 of *DB2 Administration Guide* for more information.

Defining your stored procedure to DB2

Before a stored procedure can run, you must define it to DB2. Use the SQL statement `CREATE PROCEDURE` to define a stored procedure to DB2. To alter the definition, use the `ALTER PROCEDURE` statement.

Table 77 lists the characteristics of a stored procedure and the `CREATE PROCEDURE` and `ALTER PROCEDURE` parameters that correspond to those characteristics.

Table 77. Characteristics of a stored procedure

Characteristic	CREATE/ALTER PROCEDURE parameter
Stored procedure name	PROCEDURE
Parameter declarations	
Parameter types and encoding schemes	PROCEDURE
External name	EXTERNAL NAME
Language	LANGUAGE ASSEMBLE LANGUAGE C LANGUAGE COBOL LANGUAGE JAVA LANGUAGE PLI LANGUAGE REXX LANGUAGE SQL
Deterministic or not deterministic	NOT DETERMINISTIC DETERMINISTIC
Types of SQL statements in the stored procedure	NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA
Parameter style	PARAMETER STYLE SQL ¹ PARAMETER STYLE GENERAL PARAMETER STYLE GENERAL WITH NULLS PARAMETER STYLE JAVA
Address space for stored procedures	FENCED
Package collection	NO COLLID COLLID <i>collection-id</i>
WLM environment	WLM ENVIRONMENT <i>name</i> WLM ENVIRONMENT <i>name,*</i>
How long a stored procedure can run	ASUTIME NO LIMIT ASUTIME LIMIT <i>integer</i>
Load module stays in memory	STAY RESIDENT NO STAY RESIDENT YES
Program type	PROGRAM TYPE MAIN PROGRAM TYPE SUB ²
Security	SECURITY DB2 SECURITY USER SECURITY DEFINER
Run-time options	RUN OPTIONS <i>options</i> ³
Maximum number of returned result sets	DYNAMIC RESULT SETS <i>integer</i>

Table 77. Characteristics of a stored procedure (continued)

Characteristic	CREATE/ALTER PROCEDURE parameter
Commit work on return from stored procedure	COMMIT ON RETURN YES COMMIT ON RETURN NO
Call with null arguments	CALLED ON NULL INPUT
Pass DB2 environment information	NO DBINFO DBINFO ⁴
Encoding scheme for all string parameters	PARAMETER CCSID EBCDIC PARAMETER CCSID ASCII PARAMETER CCSID UNICODE
# For procedures that are defined as LANGUAGE C # procedures, the representation of VARCHAR parameters.	PARAMETER VARCHAR NULTERM PARAMETER VARCHAR STRUCTURE ⁵
Number of abnormal terminations before the stored procedure is stopped	STOP AFTER SYSTEM DEFAULT FAILURES STOP AFTER <i>n</i> FAILURES CONTINUE AFTER FAILURE

Notes:

1. This value is invalid for a REXX stored procedure.
2. This value is ignored for a REXX stored procedure. Specifying PROGRAM TYPE SUB with REXX will not result in an error; however, a value of MAIN will be stored in the DB2 catalog and used at runtime.
3. This value is ignored for a REXX stored procedure.
4. DBINFO is valid only with PARAMETER STYLE SQL.
5. The PARAMETER VARCHAR clause can be specified in CREATE PROCEDURE statements only.

For a complete explanation of the parameters in a CREATE PROCEDURE or ALTER PROCEDURE statement, see Chapter 2 of *DB2 SQL Reference*.

Passing environment information to the stored procedure

If you specify the DBINFO parameter when you define a stored procedure with PARAMETER STYLE SQL, DB2 passes a structure to the stored procedure that contains environment information. Because the structure is also used for user-defined functions, some fields in the structure are not used for stored procedures. 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 UDB for z/OS

QSQ DB2 UDB for iSeries

SQL DB2 UDB 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 zSeries
	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.

See "Linkage conventions " on page 684 for an example of coding the DBINFO parameter list in a stored procedure.

Example of a stored procedure definition

Suppose that you write and prepare a stored procedure that has these characteristics:

- The name is B.
- It takes two parameters:
 - An integer input parameter named V1
 - A character output parameter of length 9 named V2
- It is written in the C language.
- It contains no SQL statements.
- The same input always produces the same output.
- The load module name is SUMMOD.
- The package collection name is SUMCOLL.
- It should run for no more than 900 CPU service units.
- The parameters can have null values.
- It should be deleted from memory when it completes.
- The Language Environment run-time options it needs are:

MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)

- It is part of the WLM application environment named PAYROLL.
- It runs as a main program.
- It does not access non-DB2 resources, so it does not need a special RACF environment.
- It can return at most 10 result sets.
- When control returns to the client program, DB2 should not commit updates automatically.

This 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;
```

Later, you need to make the following changes to the stored procedure definition:

- It 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 the stored procedure runs should not be limited.
- 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.

Execute this ALTER PROCEDURE statement to make the necessary changes:

```
ALTER PROCEDURE B
READS SQL DATA
ASUTIME NO LIMIT
PARAMETER STYLE SQL
WLM ENVIRONMENT (PAYROLL,*);
```

Refreshing the stored procedures environment (for system administrators)

Depending on what has changed in a stored procedures environment, you might need to perform one or more of these tasks:

- Refresh Language Environment.

Do this when someone has modified a load module for a stored procedure, and that load module is cached in a stored procedures address space. When you refresh Language Environment, the cached load module is purged. On the next invocation of the stored procedure, the new load module is loaded.

- Restart a stored procedures address space.

You might stop and then start a stored procedures address space because you need to make a change to the startup JCL for a stored procedures address space.

The method that you use to perform these tasks depends on whether you are using WLM-established or DB2-established address spaces.

For DB2-established address spaces: Use the DB2 commands START PROCEDURE and STOP PROCEDURE to perform all of these tasks.

For WLM-established address spaces:

- If WLM is operating in goal mode:
 - Use this z/OS command to refresh a WLM environment when you need to load a new version of a stored procedure. Refreshing the WLM environment starts a new instance of each address space that is active for this WLM environment. Existing address spaces stop when the current requests that are executing in those address spaces complete.

VARY WLM,APPLENV=*name*,REFRESH

name is the name of a WLM application environment associated with a group of stored procedures. When you execute this command, you affect all stored procedures that are associated with the application environment.

You can call the DB2-supplied stored procedure WLM_REFRESH to refresh a WLM environment from a remote workstation. For information about WLM_REFRESH, see “WLM environment refresh stored procedure (WLM_REFRESH)” on page 1129.

- Use this z/OS command to stop all stored procedures address spaces that are associated with WLM application environment *name*. The address spaces stop when the current requests that are executing in those address spaces complete.

VARY WLM,APPLENV=*name*,QUIESCE

- Use this z/OS command to start all stored procedures address spaces that are associated with WLM application environment *name*. New address spaces start when all JCL changes are established. Until that time, work requests that use the new address spaces are queued.

VARY WLM,APPLENV=*name*,RESUME

See *z/OS MVS Planning: Workload Management* for more information about the command VARY WLM.

- If WLM is operating in compatibility mode:
 - Use this z/OS command to stop a WLM-established stored procedures address space.
- Use this z/OS command to start a WLM-established stored procedures address space.

CANCEL *address-space-name*

START *address-space-name*

In compatibility mode, you must stop and start stored procedures address spaces when you refresh Language Environment.

Moving stored procedures to a WLM-established environment (for system administrators)

If you have existing stored procedures that use DB2-established address spaces, you need to move as many as possible to a WLM environment. To move stored procedures from a DB2-established environment to a WLM-established environment, follow these steps:

1. Define JCL procedures for the stored procedures address spaces.
Member DSNTIJMV of data set DSN810.SDSNSAMP contains sample JCL procedures for starting WLM-established address spaces.

2. Define WLM application environments for groups of stored procedures and associate a JCL startup procedure with each application environment.
See Part 5 (Volume 2) of *DB2 Administration Guide* for information about how to do this.
3. Enter the DB2 command STOP PROCEDURE(*) to stop all activity in the DB2-established stored procedures address space.
4. For each stored procedure, execute ALTER PROCEDURE with the WLM ENVIRONMENT parameter to specify the name of the application environment.
5. Relink all of your existing stored procedures with DSNRLI, the language interface module for the Resource Recovery Services attachment facility (RRSAF). Use JCL and linkage editor control statements similar to those shown in Figure 195.

```
#          //LINKRRS EXEC PGM=IEWL,
#          //          PARM='LIST,XREF,MAP'
#          //SYSPRINT DD SYSOUT=*
#          //SYSLIB DD DISP=SHR,DSN=USER.RUNLIB.LOAD
#          //          DD DISP=SHR,DSN=DSN810.SDSNLOAD
#          //SYSLMOD DD DISP=SHR,DSN=USER.RUNLIB.LOAD
#          //SYSUT1 DD SPACE=(1024,(50,50)),UNIT=SYSDA
#          //SYSLIN DD *
#          ENTRY STORPROC
#          REPLACE DSNALI
#          INCLUDE SYSLIB(DSNRLI)
#          INCLUDE SYSLMOD(STORPROC)
#          NAME STORPROC(R)
```

Figure 195. Linking existing stored procedures with RRSAF

6. If WLM is operating in compatibility mode, start the new WLM-established stored procedures address spaces by using this z/OS command:

START *address-space-name*

If WLM is operating in goal mode, the address spaces start automatically.

Writing and preparing an external stored procedure

A stored procedure is a DB2 application program that runs in a stored procedures address space.

Two types of stored procedures are *external* stored procedures and *SQL* procedures:

- External stored procedures are written in a host language. The source code for an external stored procedure is separate from the definition for the stored procedure. An external stored procedure is much like any other SQL application. It can include static or dynamic SQL statements, IFI calls, and DB2 commands issued through IFI.
- SQL procedures are written using SQL procedures statements, which are part of a CREATE PROCEDURE statement.

This section discusses writing and preparing external stored procedures. “Writing and preparing an SQL procedure” on page 657 discusses writing and preparing SQL procedures.

Language requirements for the 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. See “Coding considerations for C and C++” on page 186 and “Coding considerations for object-oriented extensions in COBOL” on page 219 for information about including object-oriented extensions in SQL applications. For a list of the minimum compiler and Language Environment requirements, see *DB2 Release Planning Guide*. For information about writing Java stored procedures, see *DB2 Application Programming Guide and Reference for Java*. For information about writing REXX stored procedures, see “Writing a REXX stored procedure” on page 654.

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.

Calling other programs

A stored 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 called program’s package belongs to in one of the following ways:

- 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, the called program’s package comes from the list of collections in the CURRENT PACKAGE PATH special register. For example, if 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, the called program’s package comes from the collection that is specified in the CURRENT PACKAGESET special register.
- If the stored procedure does not execute SET CURRENT PACKAGE PATH, SET CURRENT PACKAGESET, and if the stored procedure definition contains the NO COLLID option, DB2 uses the collection ID of the package that contains the SQL statement CALL.

When control returns from the stored procedure, DB2 restores the value of the CURRENT PACKAGESET special register to the value it contained before the client program executed the SQL statement CALL.

Using reentrant code

Whenever possible, prepare your stored procedures to be reentrant. Using reentrant stored procedures can lead to improved performance for the following reasons:

- A reentrant stored procedure does not have to be loaded into storage every time it is called.

- A single copy of the stored procedure can be shared by multiple tasks in the stored procedures address space. This decreases the amount of virtual storage used for code in the stored procedures address space.

To prepare a stored procedure as reentrant, compile it as reentrant and link-edit it as reentrant and reusable.

#

For instructions on compiling programs to be reentrant, see the appropriate language manual. For information about using the binder to produce reentrant and reusable load modules, see *z/OS MVS: Program Management User's Guide and Reference*.

To make a reentrant stored procedure remain resident in storage, specify STAY RESIDENT YES in the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

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. 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 the stored procedure.

Writing a stored procedure as a main program or subprogram

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-INT-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.

#

Table 78 on page 644 summarizes the characteristics that define a main program and a subprogram.

Table 78. 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)

Figure 196 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(cfnc,fetchable)
#include <stdlib.h>
void cfnc(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;

```

Figure 196. A C stored procedure coded as a subprogram (Part 1 of 2)

```

/*****
/* 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;
}

```

Figure 196. A C stored procedure coded as a subprogram (Part 2 of 2)

Figure 197 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;

```

Figure 197. A C++ stored procedure coded as a subprogram (Part 1 of 2)

```

/*****
/* 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;
}

```

Figure 197. A C++ stored procedure coded as a subprogram (Part 2 of 2)

A stored procedure that runs in a DB2-established address space must contain a main program.

Restrictions on a stored procedure

Do not include explicit attachment facility calls in a stored procedure. Stored procedures running in a DB2-established address space use call attachment facility (CAF) calls implicitly. Stored procedures running in a WLM-established address space use Resource Recovery Services attachment facility (RRSAF) calls implicitly. If a stored procedure makes an explicit attachment facility call, DB2 rejects the call.

```
# Do not include SRRCMIT or SRRBACK service calls in a WLM-managed stored
# procedure. If a WLM-managed stored procedure invokes either SRRCMIT or
# SRRBACK, DB2 puts the transaction in a must roll back state and the CALL
# statement returns SQLCODE -919.
```

Using COMMIT and ROLLBACK statements in a stored procedure

When you execute 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, as well as DB2 work that the stored procedure does.

A stored procedure that includes COMMIT or ROLLBACK statements must be defined with the CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA clause. There is no interaction between the COMMIT ON RETURN clause in a stored procedure definition and 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 when control returns from the stored procedure. This 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 cursors open that are declared WITH HOLD. The effect of COMMIT or ROLLBACK on cursors applies to cursors that are declared in the calling application, as well as cursors that are declared in the stored procedure.

Under the following conditions, you **cannot** include COMMIT or ROLLBACK statements in a stored procedure:

- The stored procedure is nested within a trigger or a 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 the previous conditions, DB2 puts the transaction in a must-rollback state, and the CALL statement returns a -751 SQLCODE.

Using 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 they had before invocation.

Table 79 shows information that you need to use special registers in a stored procedure.

Table 79. Characteristics of special registers in a stored procedure

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Procedure can use SET to modify?
CURRENT CLIENT_ACCTNG	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT CLIENT_APPLNAME	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT CLIENT_USERID	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT CLIENT_WRKSTNNAME	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
CURRENT APPLICATION ENCODING SCHEME	The value of bind option ENCODING for the stored procedure package ¹	The value of bind option ENCODING for the stored procedure package ¹	Yes
CURRENT DATE	New value for each SQL statement in the stored procedure package ²	New value for each SQL statement in the stored procedure package ²	Not applicable ⁵
CURRENT DEGREE	Inherited from invoking application ³	The value of field CURRENT DEGREE on installation panel DSNTIP8	Yes
CURRENT LOCALE LC_CTYPE	Inherited from invoking application	The value of field CURRENT DEGREE on installation panel DSNTIP8	Yes

Table 79. Characteristics of special registers in a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Procedure can use SET to modify?
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Inherited from invoking application	The value of field CURRENT MAINT TYPES on installation panel DSNTIP8	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	No
CURRENT OPTIMIZATION HINT	The value of bind option OPTHINT for the stored procedure package or inherited from invoking application ⁶	The value of bind option OPTHINT for the stored procedure package	Yes
CURRENT PACKAGESET	Inherited from invoking application ⁴	Inherited from invoking application ⁴	Yes
CURRENT PACKAGE PATH	Inherited from invoking application ⁹	Inherited from invoking application ⁹	Yes
CURRENT PATH	The value of bind option PATH for the stored procedure package or inherited from invoking application ⁶	The value of bind option PATH for the stored procedure package	Yes
CURRENT PRECISION	Inherited from invoking application	The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4	Yes
CURRENT REFRESH AGE	Inherited from invoking application	The value of field CURRENT REFRESH AGE on installation panel DSNTIP8	Yes
CURRENT RULES	Inherited from invoking application	The value of bind option SQLRULES for the stored procedure package	Yes
CURRENT SCHEMA	Inherited from invoking application	The value of CURRENT SQLID when the stored procedure is entered	Yes
CURRENT SERVER	Inherited from invoking application	Inherited from invoking application	Yes
CURRENT SQLID	The primary authorization ID of the application process or inherited from invoking application ⁷	The primary authorization ID of the application process	Yes ⁸
CURRENT TIME	New value for each SQL statement in the stored procedure package ²	New value for each SQL statement in the stored procedure package ²	Not applicable ⁵
CURRENT TIMESTAMP	New value for each SQL statement in the stored procedure package ²	New value for each SQL statement in the stored procedure package ²	Not applicable ⁵
CURRENT TIMEZONE	Inherited from invoking application	Inherited from invoking application	Not applicable ⁵
ENCRYPTION PASSWORD	Inherited from invoking application	A string of 0 length	Yes
USER	Primary authorization ID of the application process	Primary authorization ID of the application process	Not applicable ⁵

Table 79. Characteristics of special registers in a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Procedure can use SET to modify?
Notes:			
1. If the ENCODING bind option is not specified, the initial value is the value that was specified in field APPLICATION ENCODING of installation panel DSNTIPF.			
2. If the 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 function package.			
3. 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.			
4. If the stored procedure definer specifies a value for COLLID in the CREATE PROCEDURE statement, DB2 sets CURRENT PACKAGESET to the value of COLLID.			
5. Not applicable because no SET statement exists for the special register.			
6. If a program within the scope of the invoking application issues a SET statement for the special register before the 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 stored procedure package.			
7. If a program within the scope of the invoking application issues a SET CURRENT SQLID statement before the 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 stored procedure package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed but does not affect the authorization ID that is used for the dynamic SQL statements in the stored procedure package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements. See “Using DYNAMICRULES to specify behavior of dynamic SQL statements” on page 502 for more information about DYNAMICRULES values and authorization IDs.			
9. If the stored procedure definer specifies a value for COLLID in the CREATE PROCEDURE statement, DB2 sets CURRENT PACKAGE PATH to an empty string.			

Accessing other sites in a stored procedure

Stored procedures can access tables at other DB2 locations using three-part object names or CONNECT statements. If you use CONNECT statements, you use DRDA access to access tables. If you use three-part object names or aliases for three-part object names, the distributed access method depends on the value of DBPROTOCOL you specified when you bound the stored procedure package. If you did not specify the DBPROTOCOL bind parameter, the distributed access method depends on the value of field DATABASE PROTOCOL on installation panel DSNTIP5. A value of PRIVATE tells DB2 to use DB2 private protocol access to access remote data for the stored procedure. DRDA tells DB2 to use DRDA access.

When a local DB2 application calls a stored procedure, the stored procedure cannot have DB2 private protocol access to any DB2 sites already connected to the calling program by DRDA access.

The local DB2 application cannot use DRDA access to connect to any location that the stored procedure has already accessed using DB2 private protocol access. Before making the DB2 private protocol connection, the local DB2 application must first execute the RELEASE statement to terminate the DB2 private protocol connection, and then commit the unit of work.

See “Using three-part table names to access distributed data” on page 448 for information about using three-part names to connect to a remote location.

Writing a stored 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. See *IMS Application Programming: Database Manager* and *IMS Application Programming: Transaction Manager* for information about writing DL/I applications.

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 DSN810.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. See “Setting up the stored procedures environment” on page 634 for more information.

Writing a stored procedure to return result sets to a DRDA client

Your stored procedure can return multiple query result sets to a DRDA client if the following conditions are satisfied:

- The client supports the DRDA code points used to return query result sets.
- 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 DSN8810.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. See “Writing a DB2 UDB for z/OS client program or SQL procedure to receive result sets” on page 708 for more information.

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
- Tables, synonyms, views, created temporary tables, and aliases defined at remote DB2 UDB for z/OS systems that are accessible through DB2 private protocol access

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.

Preparing a stored procedure

There are a number of tasks that must be completed before a stored procedure can run on a z/OS server. You share these tasks with your system administrator. Part 2 of *DB2 Installation Guide* and “Defining your stored procedure to DB2” on page 635 describe what the system administrator needs to do.

Complete the following steps:

1. Precompile and compile the application.

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(RRSF) precompiler option.
- Copy the DSNRLI module into a load library that is concatenated in front of the DB2 libraries. Use the member name DSNHLL.

2. Link-edit the application. Your stored procedure must either link-edit or load one of these language interface modules:

DSNALI

The language interface module for the call attachment facility. Link-edit or load this module if your stored procedure runs in a DB2-established address space. For more information, see “Accessing the CAF language interface” on page 867.

DSNRLI

The language interface module for the Resource Recovery Services attachment facility. Link-edit or load this module if your stored procedure runs in a WLM-established address space. If the stored procedure references LOBs or distinct types, you must link-edit or load DSNRLI. For more information, see “Accessing the RRSF language interface” on page 898.

If your stored procedure runs in a WLM-established address space, you must specify the parameter AMODE(31) when you link-edit it.

3. Bind the DBRM to DB2 using the command BIND PACKAGE. Stored procedures require only a package at the server. You do not need to bind a plan. For more information, see “Binding the stored procedure” on page 653.
4. Define the stored procedure to DB2.
5. Use GRANT EXECUTE to authorize the appropriate users to use the stored procedure. For example,
GRANT EXECUTE ON PROCEDURE SPSHEMA.STORPRCA TO JONES;

That allows an application running under authorization ID JONES to call stored procedure SPSHEMA.STORPRCA.

Preparing a stored procedure to run as an authorized program: If your stored procedure runs in a WLM-established address space, you can run it as a z/OS authorized program. To prepare a stored procedure to run as an authorized program, do these additional things:

- When you link-edit the stored procedure:
 - 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.
- Be sure that the stored procedure runs in an address space with a startup procedure in which all libraries in the STEPLIB concatenation are APF-authorized. Specify an application environment WLM ENVIRONMENT parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure that ensures that the stored procedure runs in an address space with this characteristic.

Binding the stored procedure

A stored procedure does not require a DB2 plan. A stored procedure runs under the caller's thread, using the plan from the client program that calls it.

The calling application can use a DB2 package or plan to execute the CALL statement. The stored procedure must use a DB2 package as Figure 198 shows.

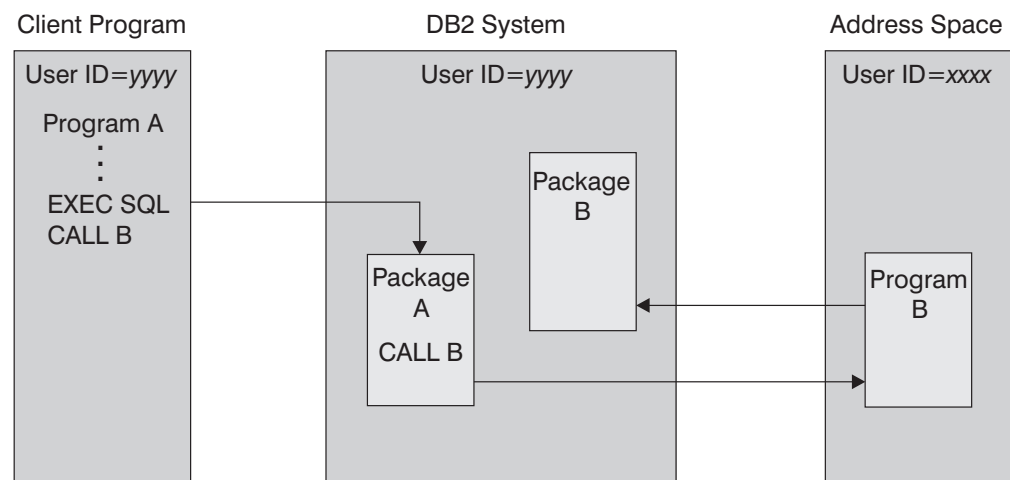


Figure 198. Stored procedures run-time environment

When you bind a stored procedure:

- Use the command BIND PACKAGE to bind the stored procedure. If you use the option ENABLE to control access to a stored procedure package, you must enable the system connection type of the application that executes the CALL statement.
- The package for the stored procedure does not need to be bound with the plan for the program that calls it.
- The owner of the package that contains the SQL statement CALL must have the EXECUTE privilege on all packages that the stored procedure accesses, including packages named in SET CURRENT PACKAGESET.

The following must exist at the server, as shown in Figure 198:

- A plan or package containing the SQL statement CALL. This package is associated with the client program.
- A package associated with the stored procedure.

The server program might use more than one package. These packages come from two sources:

- A DBRM that you bind several times into several versions of the same package, all with the same package name, which can then reside in different collections.

Your stored procedure can switch from one version to another by using the statement SET CURRENT PACKAGESET.

- A package associated with another program that contains SQL statements that the stored procedure calls.

Important: A package for a subprogram that contains SQL statements must exist at the location where the stored procedure is defined and at the location where the SQL statements are executed.

Writing a REXX stored procedure

A REXX stored procedure is much like any other REXX procedure and follows the same rules as stored procedures in other languages. It receives input parameters, executes REXX commands, optionally executes SQL statements, and returns at most one output parameter. A REXX stored procedure is different from other REXX procedures in the following ways:

- A REXX stored procedure cannot execute the ADDRESS DSNREXX CONNECT and ADDRESS DSNREXX DISCONNECT commands. 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.

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)

Figure 200 on page 655 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.

Figure 199 on page 655 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;

```

Figure 199. Definition for REXX stored procedure COMMAND

Figure 200 shows the COMMAND stored procedure that executes DB2 commands.

```

/* REXX */
PARSE UPPER ARG CMD                                /* Get the DB2 command text */
                                                    /* Remove enclosing quotes */
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 "DSNWLIR 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')

```

Figure 200. Example of a REXX stored procedure: COMMAND (Part 1 of 3)


```

/*****
/* 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)

```

Figure 200. Example of a REXX stored procedure: COMMAND (Part 2 of 3)

```

/*****
/* 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';'

```

Figure 200. Example of a REXX stored procedure: COMMAND (Part 3 of 3)

Writing and preparing an SQL procedure

An SQL procedure is a stored procedure in which the source code for the procedure is in an SQL CREATE PROCEDURE statement. The part of the CREATE PROCEDURE statement that contains the code is called the *procedure body*.

Creating an SQL procedure involves writing the source statements for the SQL procedure, creating the executable form of the SQL procedure, and defining the SQL procedure to DB2. There are two ways to create an SQL procedure:

- Use the IBM DB2 Development Center product to specify the source statements for the SQL procedure, define the SQL procedure to DB2, and prepare the SQL procedure for execution.

- Write a CREATE PROCEDURE statement for the SQL procedure. Then use one of the methods in “Preparing an SQL procedure” on page 669 to define the SQL procedure to DB2 and create an executable procedure.

This section discusses how to write a and prepare an SQL procedure. The following topics are included:

- “Comparison of an SQL procedure and an external procedure”
- “Statements that you can include in a procedure body” on page 659
- “Terminating statements in an SQL procedure” on page 662
- “Handling SQL conditions in an SQL procedure” on page 663
- “Examples of SQL procedures” on page 667
- “Preparing an SQL procedure” on page 669

For information about the syntax of the CREATE PROCEDURE statement and the procedure body, see *DB2 SQL Reference*.

Comparison of an SQL procedure and an external procedure

Like an external stored procedure, an SQL procedure consists of a stored procedure definition and the code for the stored procedure program.

An external stored procedure definition and an SQL procedure definition specify the following common 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 external stored procedure and an SQL procedure share the same rules for the use of COMMIT and ROLLBACK statements in a procedure. For information about the restrictions for the use of these statements and their effect, see “Using COMMIT and ROLLBACK statements in a stored procedure” on page 646.

An external stored procedure and an SQL stored procedure differ in the way that they handle errors.

- 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 “Passing parameter values to and from a user-defined function” on page 321.
- For an SQL stored 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 stored procedure, see “Handling SQL conditions in an SQL procedure” on page 663.

An external stored procedure and an SQL procedure also differ in the way that they specify the code for the stored procedure. An external stored procedure definition specifies the name of the stored procedure program. An SQL procedure definition contains the source code for the stored procedure.

For an external stored procedure, you define the stored procedure to DB2 by executing the CREATE PROCEDURE statement. You change the definition of the

stored procedure by executing the ALTER PROCEDURE statement. For an SQL procedure, you define the stored procedure to DB2 by preprocessing a CREATE PROCEDURE statement, then executing the CREATE PROCEDURE statement dynamically. As with an external stored procedure, you change the definition by executing the ALTER PROCEDURE statement. You cannot change the procedure body with the ALTER PROCEDURE statement. See “Preparing an SQL procedure” on page 669 for more information about defining an SQL procedure to DB2.

Figure 201 shows a definition for an 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
```

Figure 201. Example of an external stored procedure definition

Notes to Figure 201:

- 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.

Figure 202 shows a definition for an equivalent 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
```

Figure 202. Example of an SQL procedure definition

Notes to Figure 202:

- 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.

Statements that you can include in a procedure body

A procedure body consists of a single simple or compound statement. The types of statements that you can include in a procedure body are:

Assignment statement

Assigns a value to an output parameter or to an SQL variable, which is a variable that is defined and used only within a procedure body. The right side of an assignment statement can include SQL built-in functions.

CALL statement

Calls another stored procedure. This statement is similar to the CALL statement described in Chapter 5 of *DB2 SQL Reference*, except that the parameters must be SQL variables, parameters for the SQL procedure, or constants.

CASE statement

Selects an execution path based on the evaluation of one or more conditions. This statement is similar to the CASE expression, which is described in Chapter 2 of *DB2 SQL Reference*.

GET DIAGNOSTICS statement

Obtains information about the previous SQL statement that was executed. An example of its usage is shown in “Using GET DIAGNOSTICS in a handler” on page 664.

GOTO statement

Transfers program control to a labelled statement.

IF statement

Selects an execution path based on the evaluation of a condition.

ITERATE statement

Transfers program control to beginning of a labelled loop.

LEAVE statement

Transfers program control out of a loop or a block of code.

LOOP statement

Executes a statement or group of statements multiple times.

REPEAT statement

Executes a statement or group of statements until a search condition is true.

WHILE statement

Repeats the execution of a statement or group of statements while a specified condition is true.

Compound statement

Can contain one or more of any of the other types of statements in this list. In addition, a compound statement can contain SQL variable declarations, condition handlers, or cursor declarations.

The order of statements in a compound statement must be:

1. SQL variable and condition declarations
2. Cursor declarations
3. Handler declarations
4. Procedure body statements (CALL, CASE, IF, LOOP, REPEAT, WHILE, SQL)

SQL statement

A subset of the SQL statements that are described in Chapter 5 of *DB2 SQL Reference*. Certain SQL statements are valid in a compound statement, but not valid if the SQL statement is the only statement in the procedure body. Appendix C of *DB2 SQL Reference* lists the SQL statements that are valid in an SQL procedure.

SIGNAL statement

Enables an SQL procedure to raise a condition with a specific SQLSTATE and message text. This statement is described in “Using SIGNAL or RESIGNAL to raise a condition” on page 665.

RESIGNAL statement

Enables a condition handler within an SQL procedure to raise a condition with a specific SQLSTATE and message text, or to return the same condition that activated the handler. This statement is described in “Using SIGNAL or RESIGNAL to raise a condition” on page 665.

RETURN statement

Returns an integer status value for the SQL procedure. This statement is described in “Using the RETURN statement for the procedure status” on page 665.

See the discussion of the procedure body in *DB2 SQL Reference* for detailed descriptions and syntax of each of these statements.

Declaring and using variables, parameters, and conditions 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. For a discussion of data types and lengths, see the CREATE PROCEDURE discussion in Chapter 5 of *DB2 SQL Reference*.

The general form of an SQL variable declaration is:

```
DECLARE SQL-variable-name data-type;
```

The general form of a declaration for an SQL variable that you use as a result set locator is:

```
DECLARE SQL-variable-name data-type RESULT_SET_LOCATOR VARYING;
```

SQL variables have these restrictions:

- SQL variable names can be up to 128 bytes in length. They can include alphanumeric characters and the underscore character. Condition names and label names also have these restrictions.
- Because DB2 folds all SQL variables to uppercase, you cannot declare two SQL variables that are the same except for case. For example, you cannot declare two SQL variables named varx and VARX.
- Although it 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.
- When you call a user-defined function from an SQL procedure, and the user-defined function definition includes parameters of type CHAR, you need to cast the corresponding parameter values in the user-defined function invocation to CHAR to ensure that DB2 invokes the correct function. For example, suppose that an SQL procedure calls user-defined function CVRTNUM, which takes one

input parameter of type CHAR(6). Also suppose that you declare SQL variable EMPNUMBR in the SQL procedure. When you invoke CVRTNUM, cast EMPNUMBR to CHAR:

```
UPDATE EMP
SET EMPNO=CVRTNUM(CHAR(EMPNUMBR))
WHERE EMPNO = EMPNUMBR;
```

- 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 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 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.
 - You can use a parameter that you define as INOUT on the left 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.

You can perform any operations on SQL variables that you can perform on host variables in SQL statements.

Qualifying SQL variable names and other object names is a good way to avoid ambiguity. Use the following guidelines to determine when to qualify variable 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 SQL variable names in the compound statement with that label.
- Qualify column names with the associated table or view names.

Recommendation: Because the way that DB2 determines the qualifier for unqualified names might change in the future, qualify all SQL variable names to avoid changing your code later.

Parameter style for an SQL procedure

DB2 supports only the GENERAL WITH NULLS linkage convention for SQL procedures. This means that when you call an SQL procedure, you must include an indicator variable with each parameter in the CALL statement. See “Linkage conventions” on page 684 for more information about stored procedure linkage conventions.

Terminating statements in an SQL procedure

The way that you terminate a statement in an SQL procedure depends on the use of the statement in that procedure:

- A procedure body has no terminating character. Therefore, if an SQL procedure statement is the outermost of a set of nested statements, or if the statement is the only statement in the procedure body, that statement does not have a terminating character.
- If a statement is nested within other statements in the procedure body, that statement ends with a semicolon.

Handling SQL conditions in an SQL procedure

You can handle SQL errors and SQL warnings in an SQL procedure by using the following techniques:

- You can include statements called *handlers* to tell the procedure to perform some other action when an error occurs; see “Using handlers in an SQL procedure.”
- You can include a RETURN statement in an SQL procedure to return an integer status value to the caller; see “Using the RETURN statement for the procedure status” on page 665.
- 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.
- You can include a SIGNAL statement or a RESIGNAL statement to raise a specific SQLSTATE and to define the message text for that SQLSTATE; see “Using SIGNAL or RESIGNAL to raise a condition” on page 665.
- You can force a negative SQLCODE to be returned by a procedure if a trigger calls the procedure; see “Forcing errors in an SQL procedure when called by a trigger” on page 667.

Using handlers in an SQL procedure

If an SQL error occurs when an SQL procedure executes, the SQL procedure ends unless you include statements called handlers to tell the procedure to perform some other action.

Handlers are similar to WHENEVER statements in external SQL application programs. Handlers tell the SQL procedure what to do when an SQL error or SQL 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';
```

Referencing SQLCODE and SQLSTATE in a handler: When an SQL error or warning occurs in an SQL procedure, you might want a handler to reference the SQLCODE or SQLSTATE value and assign the value to an output parameter to be passed back to the caller.

Before you can reference SQLCODE or SQLSTATE values in a handler, you must declare the SQLCODE and SQLSTATE as SQL variables. The definitions are:

```
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
```

If you want to pass the SQLCODE or SQLSTATE values to the caller, your SQL procedure definition needs to include output parameters for those values. After an error occurs, and before control returns to the caller, you can assign the value of SQLCODE or SQLSTATE to the corresponding output parameter.

Example: Assigning SQLCODE to output parameter: Include assignment statements in an SQLEXCEPTION handler to assign the SQLCODE value to an output parameter:

```
CREATE PROCEDURE UPDATESALARY1
(IN EMPNUMBR CHAR(6),
OUT SQLCPARM INTEGER)
LANGUAGE SQL
:
:
BEGIN:
DECLARE SQLCODE INTEGER;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET SQLCPARM = SQLCODE;
:
:
```

Every statement in an SQL procedure sets the SQLCODE and SQLSTATE. Therefore, if you need to preserve SQLCODE or SQLSTATE values after a statement executes, use a simple assignment statement to assign the SQLCODE and SQLSTATE values to other variables. A statement like the following one does not preserve SQLCODE:

```
IF (1=1) THEN SET SQLCDE = SQLCODE;
```

Because the IF statement is true, the SQLCODE value is reset to 0, and you lose the previous SQLCODE value.

Using GET DIAGNOSTICS in a handler: 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(70))
LANGUAGE SQL
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    GET DIAGNOSTICS CONDITION 1 divide_error = MESSAGE_TEXT;
  SET divide_result = numerator / denominator;
END

```

Using the RETURN statement for the procedure status

You can use the RETURN statement in an SQL procedure to return an integer status value. If you include a RETURN statement, DB2 sets the SQLCODE in the SQLCA to 0, and the caller must retrieve the return status of the procedure in either of the following ways:

- By using the RETURN_STATUS item of GET DIAGNOSTICS to retrieve the return value of the RETURN statement
- By retrieving SQLERRD(0) of the SQLCA, which contains the return value of the RETURN statement

If you do not include a RETURN statement in an SQL procedure, by default, DB2 sets the return status to 0 for an SQLCODE that is greater than or equal to 0, and to -1 for an SQLCODE less than 0.

Example: Using GET DIAGNOSTICS to retrieve the return status: Suppose that you create an SQL procedure, named TESTIT, that calls another SQL procedure, named TRYIT. The TRYIT procedure returns a status value, and the TESTIT procedure retrieves that value with the RETURN_STATUS item of GET DIAGNOSTICS:

```

CREATE PROCEDURE TESTIT ()
LANGUAGE SQL
A1:BEGIN
  DECLARE RETVAL INTEGER DEFAULT 0;
  ...
  CALL TRYIT;
  GET DIAGNOSTICS RETVAL = RETURN_STATUS;
  IF RETVAL <> 0 THEN
    ...
    LEAVE A1;
  ELSE
    ...
  END IF;
END A1

```

Using SIGNAL or RESIGNAL to raise a condition

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 “Trigger body” on page 283.
- 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; however, using the range of SQLSTATE values reserved for applications is recommended.

Using the SIGNAL statement in an SQL procedure: You can use the SIGNAL statement anywhere within an SQL procedure. 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; see the example in “Using the RESIGNAL statement in a handler.”

Using the RESIGNAL statement in a handler: You can use a RESIGNAL statement to assign an SQLSTATE value (to the condition that activated the handler) that is different from the SQLSTATE value that DB2 defined for that condition.

Example: Using RESIGNAL to set an SQLSTATE value: 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

```

Forcing errors in an SQL procedure when called by a trigger

Suppose that a trigger in your application invokes an SQL stored procedure, and the body of the procedure contains an SQL statement that returns a warning. In some circumstances, you might want the procedure to return a negative SQLCODE so that the trigger will fail.

You can force a negative SQLCODE by issuing a COMMIT or ROLLBACK statement within the procedure. These statements are accepted at CREATE PROCEDURE time, but, at run time, they violate the restriction that COMMIT and ROLLBACK statements are not allowed in procedures that are called from a trigger. For information about restrictions for the use of these statements, see “Using COMMIT and ROLLBACK statements in a stored procedure” on page 646.

Examples of SQL procedures

This section contains examples of how to use each of the statements that can appear in 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, the FETCH statement that retrieves employee records receives SQLCODE 100. 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 procedure body receives a negative SQLCODE, the SQLEXCEPTION handler receives control. This handler sets output parameter DEPTSALARY to NULL and ends execution of the SQL procedure. When this handler is invoked, the SQLCODE and SQLSTATE are set to 0.

```
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 CREATEDPTTABLE (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

```

Preparing an SQL procedure

After you create the source statements for an SQL procedure, you need to prepare the procedure to run. This process involves three basic tasks:

- Using the DB2 SQL precompiler to convert the SQL procedure source statements into a C language program
- Creating an executable load module and a DB2 package from the C language program

This task includes:

- Precompiling the C language program to generate a DBRM and a modified C language program
- Binding the DBRM to generate a DB2 package
- Defining the stored procedure to DB2

This task is done by executing the CREATE PROCEDURE statement for the SQL procedure statically or dynamically. If you prepare an SQL procedure through the SQL procedure processor or the IBM DB2 Development Center, this task is performed for you.

The three methods available for preparing an SQL procedure to run are:

- Using IBM DB2 Development Center, which runs on Windows NT, Windows 95, Windows 98, Windows 2000, and AIX.
- Using the DB2 UDB for z/OS SQL procedure processor. See “Using the DB2 UDB for z/OS SQL procedure processor to prepare an SQL procedure” on page 670.
- Using JCL. See “Using JCL to prepare an SQL procedure” on page 680.

To run an SQL procedure, you must call it from a client program, using the SQL CALL statement. See the description of the CALL statement in Chapter 2 of *DB2 SQL Reference* for more information.

To debug an SQL procedure, you must prepare and call it from a client
development platform that includes the SQL Debugger feature. For additional
information about IBM DB2 Development Center, see [http://](http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp)
publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp.

Using the DB2 UDB for z/OS SQL procedure processor to prepare an SQL procedure

| The SQL procedure processor, DSNTPSMP, is a REXX stored procedure that you
| can use to prepare an SQL procedure for execution. You can also use DSNTPSMP
| to perform selected steps in the preparation process or delete an existing SQL
| procedure. DSNTPSMP is the only preparation method that supports the SQL
| Debugger.

DSNTPSMP requires that the default EBCDIC CCSID that is used by DB2 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.

The following sections contain information about invoking DSNTPSMP.

Environment for calling and running DSNTPSMP: You can invoke DSNTPSMP only through an SQL CALL statement in an application program or through IBM DB2 Development Center.

Before you can run DSNTPSMP, you need to perform the following steps to set up the DSNTPSMP environment:

1. Install DB2 UDB for z/OS REXX Language Support feature.
Contact your IBM service representative for more information.
2. If you plan to call DSNTPSMP directly, write and prepare an application program that executes an SQL CALL statement for DSNTPSMP.
See “Invoking DSNTPSMP in an application program” on page 673 for more information.
If you plan to invoke DSNTPSMP through the IBM DB2 Development Center, see the following URL for information about installing and using the IBM DB2 Development Center.
<http://www.redbooks.ibm.com/abstracts/sg247083.html>
3. Set up a WLM environment in which to run DSNTPSMP. See Part 5 (Volume 2) of *DB2 Administration Guide* for general information about setting up WLM application environments for stored procedures and “Setting up a WLM application environment for DSNTPSMP” for specific information for DSNTPSMP.

| **Setting up a WLM application environment 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 (see the first note for Figure 203 on page 671 for
| information regarding NUMTCB).

| Figure 203 on page 671 shows sample JCL for a startup procedure for the address
| space in which DSNTPSMP runs.
|


```

//DSN8WLMP PROC DB2SSN=DSN,NUMTCB=1,APPLENV=WLMTPSMP
/*
//WLMTPSMP EXEC PGM=DSNX9WLM,TIME=1440,
//          PARM='&DB2SSN,&NUMTCB,&APPLENV',
//          REGION=0M,DYNAMNBR=10
//STEPLIB DD DISP=SHR,DSN=DSN810.SDSNEXIT
//          DD DISP=SHR,DSN=DSN810.SDSNLOAD
//          DD DISP=SHR,DSN=CBC.SCCNCMP
//          DD DISP=SHR,DSN=CEE.SCEERUN
//SYSEXEC DD DISP=SHR,DSN=DSN810.SDSNCLST
//SYSTSPT DD SYSOUT=A
//CEEDUMP DD SYSOUT=A
//SYSABEND DD DUMMY
/*
//SQLDBRM DD DISP=SHR,DSN=DSN810.DBRMLIB.DATA
//SQLCSRC DD DISP=SHR,DSN=USER.PSMLIB.DATA
//SQLLMOD DD DISP=SHR,DSN=DSN810.RUNLIB.LOAD
//SQLLIBC DD DISP=SHR,DSN=CEE.SCEEH.H
//          DD DISP=SHR,DSN=CEE.SCEEH.SYS.H
//SQLLIBL DD DISP=SHR,DSN=CEE.SCEELKED
//          DD DISP=SHR,DSN=DSN810.SDSNLOAD
//SYSMSGSG DD DISP=SHR,DSN=CEE.SCEMSGP(EDCPMSGGE)
/* DSNTPSMP Configuration File - CFGTPSMP (optional)
/*          A site-provided sequential dataset or member, used to
/*          define customized operation of DSNTPSMP in this APPLNV
/*
/*CFGTPSMP DD DISP=SHR,DSN=
/*
//SQLSRC DD UNIT=SYSALLDA,SPACE=(800,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLPRINT DD UNIT=SYSALLDA,SPACE=(16000,(20,20)),
//          DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SQLTERM DD UNIT=SYSALLDA,SPACE=(4000,(20,20)),
//          DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SQLOUT DD UNIT=SYSALLDA,SPACE=(16000,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLCPRT DD UNIT=SYSALLDA,SPACE=(16000,(20,20)),
//          DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SQLUT1 DD UNIT=SYSALLDA,SPACE=(16000,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLUT2 DD UNIT=SYSALLDA,SPACE=(16000,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLCIN DD UNIT=SYSALLDA,SPACE=(8000,(20,20))
//SQLLIN DD UNIT=SYSALLDA,SPACE=(3200,(30,30)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSMOD DD UNIT=SYSALLDA,SPACE=(8000,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLDUMMY DD DUMMY

```

Figure 203. Startup procedure for a WLM address space in which DSNTPSMP runs

Notes to Figure 203:

- 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 SQL procedure, you can set up different application environments for preparing different types of SQL procedures. 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 WLMTSPMP specifies the address space in which DSNTPSMP runs.

DYNAMNBR allows for dynamic allocation.

3 STEPLIB specifies the Language Environment run-time library that DSNTPSMP uses when it runs.

4 SYSEXEC specifies the library that contains DSNTPSMP.

5 SQLDBRM specifies the library into which DSNTPSMP puts the DBRM that it generates when it precompiles your SQL procedure.

6 SQLCSRC specifies the library into which DSNTPSMP puts the C source code that it generates from the SQL procedure source code. This data set should have a logical record length of 80.

7 SQLLMOD specifies the library into which DSNTPSMP puts the load module that it generates when it compiles and link-edits your 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 SQL procedure:

- Language Environment link-edit library
- DB2 load library

10 SYMSGSGS 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.

Authorizations to execute DSNTPSMP: You must have the following authorizations to invoke DSNTPSMP:

- 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:
CREATE ON COLLECTION *collection-id*
You can use an asterisk (*) as the identifier for a collection.
- Package privilege to use BIND or REBIND to bind packages in the specified collection:
BIND ON PACKAGE *collection-id.**
- 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:

CREATEIN, ALTERIN, DROPIN ON SCHEMA *schema-name*

The BUILDOWNER authorization id must have the CREATEIN privilege on the schema. You can use an asterisk (*) as the identifier for a schema.

- Table privileges to select or delete from, insert into, or update these 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

In addition, the authorizations must include any privileges required for the SQL statements that are contained within the SQL *procedure-body*. These privileges must be associated with the OWNER authorization-id that is specified in your bind options. The default owner is the user that is invoking DSNTPSMP.

Invoking DSNTPSMP in an application program: In an application program, you can invoke DSNTPSMP through an SQL CALL statement. To prepare the program that calls DSNTPSMP, you need to precompile, compile, and link-edit the application program as usual, and then bind a package for that program.

Figure 204 and Figure 205 shows the syntax of invoking DSNTPSMP through the SQL CALL statement:

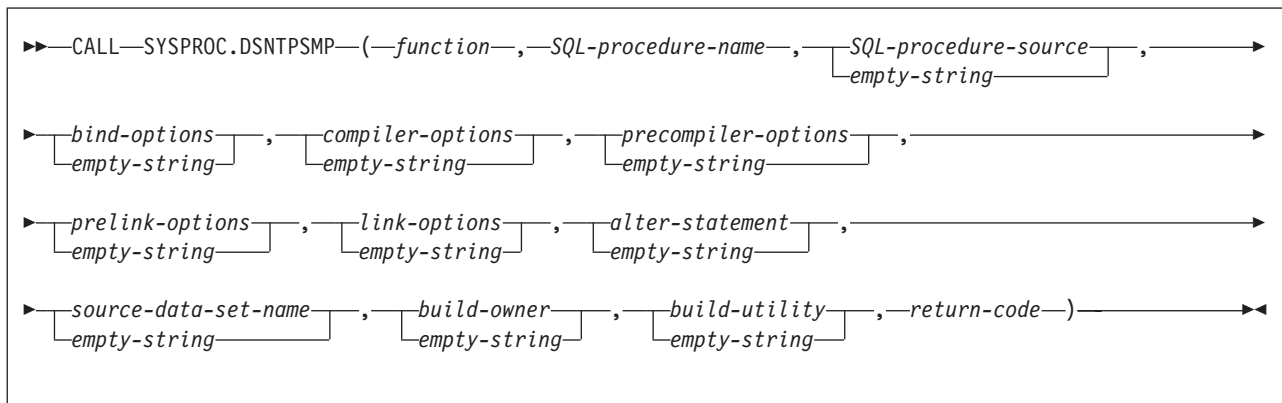


Figure 204. DSNTPSMP syntax

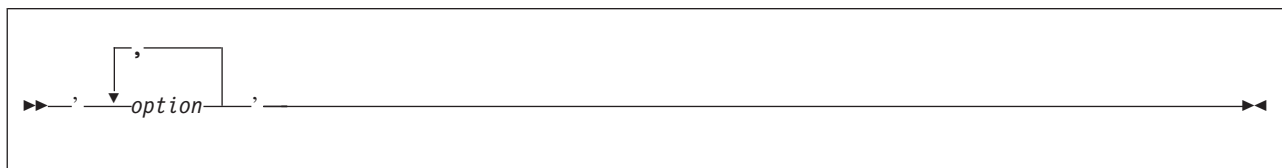


Figure 205. CALL DSNTPSMP bind-options, compiler-options, precompiler-options, prelink-options, link-options

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 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 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 SQL procedure with name *SQL-procedure-name* already exists, DSNTPSMP issues an error message and terminates.

BUILD_DEBUG

Creates the following objects for an SQL procedure and includes the preparation necessary to debug the SQL procedure with the SQL 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 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 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 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 SQL procedure, if it exists, otherwise creates those objects, and includes the preparation necessary to debug the SQL procedure with the SQL 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 SQL procedure package for an existing SQL procedure.

The following input parameter is required for the REBIND function:

SQL-procedure name

DESTROY

Deletes the following objects for an existing 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 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 SQL procedure.

The following input parameters are required for the ALTER function:

SQL-procedure name

alter-statement

ALTER_REBUILD

Updates an existing 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 SQL procedure, and includes the preparation necessary to debug the SQL procedure with the SQL 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 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 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 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 SQL procedure source code.

bind-options

A VARCHAR(1024) input parameter that contains the options that you want to specify for binding the SQL procedure package. Do not specify the MEMBER or LIBRARY option for the DB2 BIND PACKAGE command. For a list of valid bind options for the DB2 BIND PACKAGE command, see Part 3 of *DB2 Command Reference*.

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 SQL procedure. For a list of valid compiler options, see *z/OS C/C++ User's Guide*.

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 SQL procedure. Do not specify the HOST option. For a list of valid precompiler options, see Table 64 on page 484.

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 SQL procedure. For a list of valid prelink options, see *z/OS C/C++ User's Guide*.

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 SQL procedure. For a list of valid link options, see *z/OS MVS: Program Management User's Guide and Reference*.

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 SQL procedure. If you specify an empty string for this parameter, you need to provide the 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.
- 999 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.

1.20

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.

Result set that 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. This technique is shown in “Writing a DB2 UDB for z/OS client program or SQL procedure to receive result sets” on page 708.

Each row of the result set contains the following information:

Processing step

The step in the *function* process to which the message applies.

ddname

The ddname of 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, ddname, and sequence number.

Completing the requested DSNTPSMP action: The calling application must issue either an SQL COMMIT statement or an SQL ROLLBACK statement after the DSNTPSMP request. A return value of '0' or '4' requires the COMMIT statement. Any other return value requires the 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.

Examples of DSNTPSMP invocation: The following examples illustrate invoking the BUILD, DESTROY, REBUILD, and REBIND functions of DSNTPSMP.

DSNTPSMP BUILD function: Call DSNTPSMP to build an SQL procedure. The information that DSNTPSMP needs is listed in Table 80:

Table 80. The functions DSNTPSMP needs to BUILD an SQL stored procedure

Function	BUILD
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 SQL procedure definition and the associated load module. The information that DSNTPMSP needs is listed in Table 81:

Table 81. The functions DSNTPSMP needs to DESTROY an SQL stored procedure

Function	DESTROY
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 recreate an existing SQL procedure. The information that DSNTPMSP needs is listed in Table 82:

Table 82. The functions DSNTPSMP needs to REBUILD an SQL stored procedure

Function	REBUILD
SQL procedure name	MYSHEMA.SQLPROC

Table 82. The functions DSNTPSMP needs to REBUILD an SQL stored procedure (continued)

Function	REBUILD
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 DSN810.SDSNSAMP
Return value	String returned in varying-length host variable returnval

The CALL statement is:

```
EXEC SQL CALL SYSPROC.DSNTPSMP('REBUILD','MYSCHEMA.SQLPROC','',
    'VALIDATE(BIND)',
    'SOURCE,LIST,LONGNAME,RENT',
    'SOURCE,XREF,STDSQL(NO)',
    '',
    'AMODE=31,RMODE=ANY,MAP,RENT',
    '', 'DSN810.SDSNSAMP(PROCSRC)', '', '',
    :returnval);
```

If you want to recreate an existing SQL procedure for debugging with the SQL 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',
    '', 'DSN810.SDSNSAMP(PROCSRC)', '', '',
    :returnval);
```

DSNTPSMP REBIND function: Call DSNTPSMP to rebind the package for an existing SQL procedure. The information that DSNTPMSP needs is listed in Table 83:

Table 83. The functions DSNTPSMP needs to REBIND an SQL stored procedure

Function	REBIND
SQL 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);
```


Using JCL to prepare an SQL procedure

No support is provided for using JCL to prepare an SQL procedure for debugging with the SQL Debugger.

Use the following steps to prepare an SQL procedure using JCL.

1. Preprocess the CREATE PROCEDURE statement.
To do this, execute program DSNHPC, with the HOST(SQL) option. This process converts the SQL procedure source statements into a C language program.
2. Precompile the C language source program that was generated in step 1.
This process produces a DBRM and modified C language source statements. When you perform this step, ensure that you do the following things:
 - Give the DBRM the same name as the name of the load module for the SQL procedure.
 - Specify MARGINS(1,80) for the MARGINS precompiler option.
3. Compile and link-edit the modified C source statements that were produced in step 1.
This process produces an executable C language program. When you compile the C language program, ensure that the compiler options include the option NOSEQ.
4. Bind the DBRM that was produced in step 1 into a package.
5. Define the stored procedure to DB2.
To do this, execute the CREATE PROCEDURE statement for the SQL procedure. You can execute the CREATE PROCEDURE statement dynamically, using an application such as SPUFI or DSNTEP2. Executing the CREATE PROCEDURE statement puts the stored procedure definition in the DB2 catalog. If the routine body of the CREATE PROCEDURE statement contains embedded semicolons, change the default SQL terminator character in SPUFI, DSNTEP2, or DSNTIAD from a semicolon to some other special character, such as the percent sign (%). For information about to change the default SQL terminator character, see one of the following sections:
 - “Running DSNTIAD” on page 1024
 - “Running DSNTEP2 and DSNTEP4” on page 1026
 - “Changing SPUFI defaults” on page 62

Sample programs to help you prepare and run SQL procedures

Table 84 lists the sample jobs that DB2 provides to help you prepare and run SQL procedures. All samples are in data set DSN810.SDSNSAMP. Before you can run the samples, you must customize them for your installation. See the prolog of each sample for specific instructions.

Table 84. SQL procedure samples shipped with DB2

Member that contains source code	Contents	Purpose
DSNHSQL	JCL procedure	Precompiles, compiles, prelink-edits, and link-edits an SQL procedure
DSNTEJ63	JCL job	Invokes JCL procedure DSNHSQL to prepare SQL procedure DSN8ES1 for execution

Table 84. SQL procedure samples shipped with DB2 (continued)

Member that contains source code	Contents	Purpose
DSN8ES1	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
DSN8ED3	C program	Calls SQL procedure DSN8ES1
DSN8ES2	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 SQL procedure updates the BONUS column of DSN810.SDSNSAMP. If no SQL error occurs when the 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 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.
DSNTIJSJ	JCL job	Prepares a DB2 UDB for z/OS server for operation with the SQL Debugger

#

Writing and preparing an application to use stored procedures

Use the SQL statement CALL to call a stored procedure and to pass a list of parameters to the procedure. See Chapter 5 of *DB2 SQL Reference* for the syntax and a complete description of the CALL statement.

An application program that calls a stored procedure can:

- Call more than one stored procedure.
- Call a single stored procedure more than once at the same or at different levels of nesting.
- Execute the CALL statement locally or send the CALL statement to a server. The application executes a CONNECT statement to connect to the server and then executes the CALL statement, or uses a three-part name to identify and implicitly connect to the server where the stored procedure is located.
- After connecting to a server, mix CALL statements with other SQL statements.

|
|

Use either of these methods to execute the CALL statement:

- Execute the CALL statement statically.
- Use an escape clause in an ODBC application to pass the CALL statement to DB2.
- Use any of the DB2 attachment facilities.

Forms of the CALL statement

The simplest form of a CALL statement looks like this:

```
EXEC SQL CALL A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, :CODE);
```

where :EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, and :CODE are host variables that you have declared earlier in your application program. Your CALL statement might vary from the preceding statement in the following ways:

- Instead of passing each of the employee and project parameters separately, you could pass them together as a host structure. For example, assume that you define a host structure like this:

```
struct {  
    char EMP[7];  
    char PRJ[7];  
    short ACT;  
    short EMT;  
    char EMS[11];  
    char EME[11];  
} empstruc;
```

The corresponding CALL statement looks like this:

```
EXEC SQL CALL A (:empstruc, :TYPE, :CODE);
```

- Suppose that A is in schema SCHEMAA at remote location LOCA. To access A, you could use either of these methods:

- Execute a CONNECT statement to LOCA, and then execute the CALL statement:

```
EXEC SQL CONNECT TO LOCA;  
EXEC SQL CALL SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME,  
    :TYPE, :CODE);
```

- Specify the three-part name for A in the CALL statement:

```
EXEC SQL CALL LOCA.SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS,  
    :EME, :TYPE, :CODE);
```

The advantage of using the second form is that you do not need to execute a CONNECT statement. The disadvantage is that this form of the CALL statement is not portable to other operating systems.

If your program executes 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.

- The preceding examples assume that none of the input parameters can have null values. To allow null values, code a statement like this:

```
EXEC SQL CALL A (:EMP :IEMP, :PRJ :IPRJ, :ACT :IACT,  
    :EMT :IEMT, :EMS :IEMS, :EME :IEME,  
    :TYPE :ITYPE, :CODE :ICODE);
```

where :IEMP, :IPRJ, :IACT, :IEMT, :IEMS, :IEME, :ITYPE, and :ICODE are indicator variables for the parameters.

- You might pass integer or character string constants or the null value to the stored procedure, as in this example:

```
EXEC SQL CALL A ('000130', 'IF1000', 90, 1.0, NULL, '1982-10-01',
                :TYPE, :CODE);
```

- You might use 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'. You should use this technique if you do not know in advance the name of the stored procedure, but you do know the parameter list convention.

- If you prefer to pass your parameters in a single structure, rather than as separate host variables, you might use this form:

```
EXEC SQL CALL A USING DESCRIPTOR :sqlda;
```

sqlda is the name of an SQLDA.

One advantage of using this form 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 desired 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, which is described in “Changing the CCSID for retrieved data” on page 619. 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, or the default encoding scheme for the subsystem, if you do not specify an encoding scheme in the CREATE PROCEDURE statement.

- You might execute the CALL statement by using a host variable name for the stored procedure with an SQLDA:

```
EXEC SQL CALL :procnm USING DESCRIPTOR :sqlda;
```

This form gives you extra flexibility because you can use the same CALL statement to call different stored procedures with different parameter lists.

Your client program must assign a stored procedure name to the host variable *procnm* and load the SQLDA with the parameter information before making the SQL CALL.

Each of the preceding CALL statement examples uses an SQLDA. If you do not explicitly provide an SQLDA, the precompiler generates the SQLDA based on the variables in the parameter list.

Authorization for executing stored procedures

To execute a stored procedure, you need two types of authorization:

- Authorization to execute the CALL statement
- Authorization to execute the stored procedure package and any packages under the stored procedure package.

The authorizations you need depend on whether the form of the CALL statement is CALL *literal* or CALL *:host-variable*.

If the stored procedure invokes user-defined functions or triggers, you need additional authorizations to execute the trigger, the user-defined function, and the user-defined function packages.

For more information, see the description of the CALL statement in Chapter 5 of *DB2 SQL Reference*.

Linkage conventions

When an application executes the CALL statement, DB2 builds a parameter list for the stored procedure, using the parameters and values provided in the statement. DB2 obtains information about parameters from the stored procedure definition you create when you execute CREATE PROCEDURE. Parameters are defined as one of these 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/output parameters, which provide values to or return values from the stored procedure.

If a stored procedure fails to set one or more of the output-only parameters, DB2 does not detect the error in the stored procedure. Instead, DB2 returns the output parameters to the calling program, with the values established on entry to the stored procedure.

Initializing output parameters: For a stored procedure that runs locally, you do not need to initialize the values of output parameters before you call the stored procedure. However, when you call a stored procedure at a remote location, the local DB2 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.

It is recommended that you initialize the length of LOB output parameters to zero. Doing so can improve your performance.

DB2 supports three parameter list conventions. DB2 chooses the parameter list convention based on the value of the PARAMETER STYLE parameter in the stored procedure definition: GENERAL, GENERAL WITH NULLS, or SQL.

- Use **GENERAL** when you do not want the calling program to pass null values for input parameters (IN or INOUT) to the stored procedure. The stored procedure must contain a variable declaration for each parameter passed in the CALL statement.

Figure 206 on page 685 shows the structure of the parameter list for PARAMETER STYLE GENERAL.

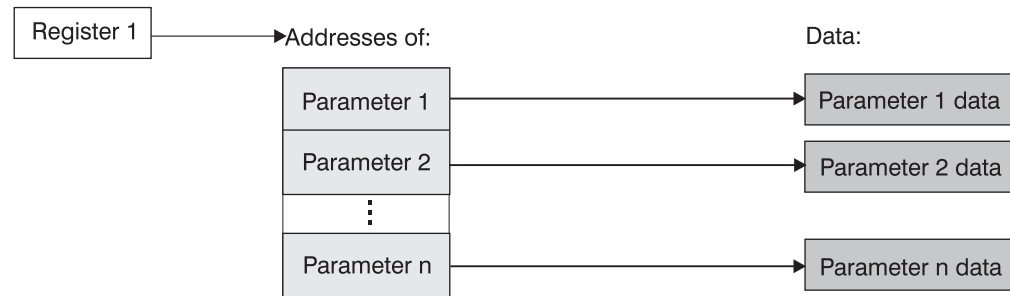


Figure 206. Parameter convention *GENERAL* for a stored procedure

- Use **GENERAL WITH NULLS** to allow the calling program to supply a null value for any parameter passed to the stored procedure. For the **GENERAL WITH NULLS** linkage convention, the stored procedure must do the following tasks:
 - Declare a variable for each parameter passed in the **CALL** statement.
 - Declare a null indicator structure containing an indicator variable for each parameter.
 - On entry, examine all indicator variables associated with input parameters to determine which parameters contain null values.
 - On exit, assign values to all indicator variables associated with output variables. An indicator variable for an output variable that returns a null value to the caller must be assigned a negative number. Otherwise, the indicator variable must be assigned the value 0.

In the **CALL** statement, follow each parameter with its indicator variable, using one of the following forms:

```

host-variable :indicator-variable
or
host-variable INDICATOR :indicator-variable.
  
```

Figure 207 shows the structure of the parameter list for **PARAMETER STYLE GENERAL WITH NULLS**.

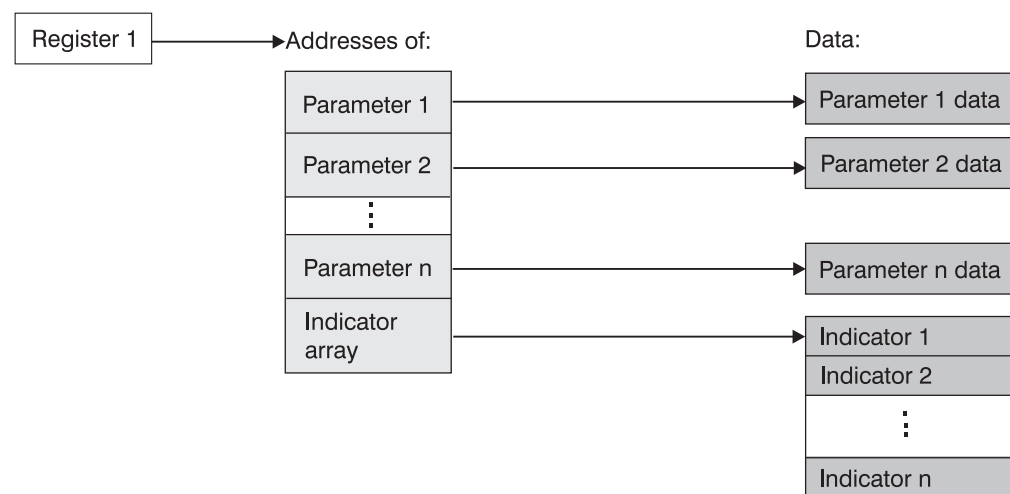
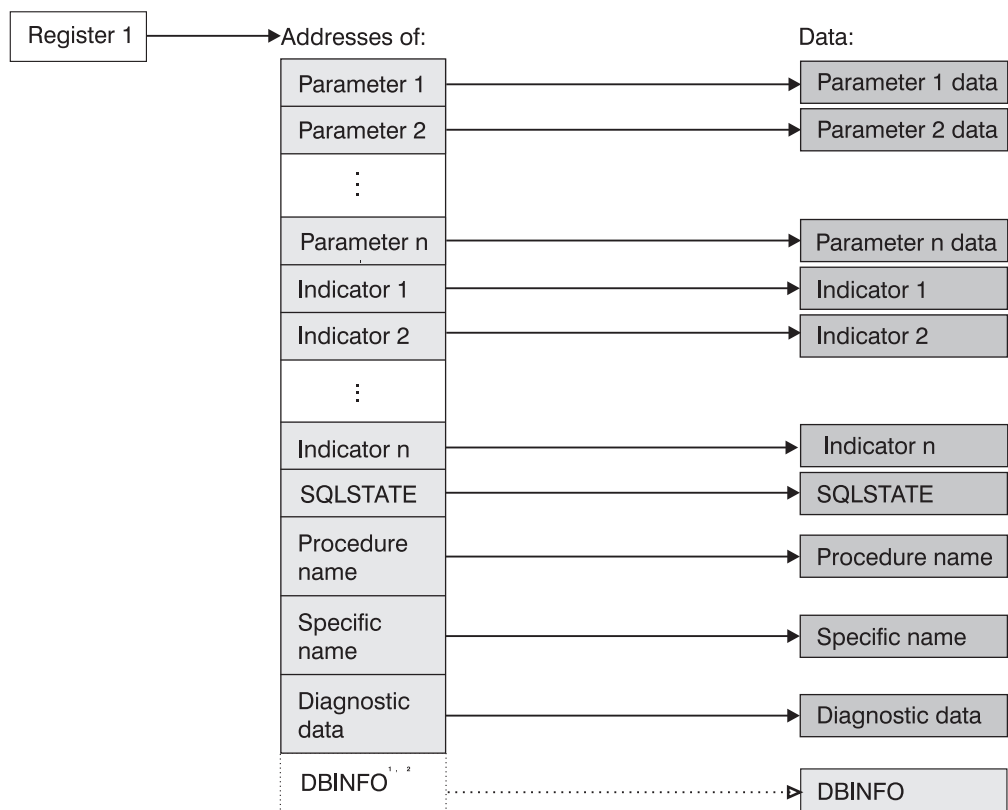


Figure 207. Parameter convention *GENERAL WITH NULLS* for a stored procedure

- Like **GENERAL WITH NULLS**, option **SQL** lets you supply a null value for any parameter that is passed to the stored procedure. In addition, DB2 passes input and output parameters to the stored procedure that contain this information:

- The SQLSTATE that is to be returned to DB2. This is a CHAR(5) parameter that 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. Refer to *DB2 Codes* for more information about the SQLSTATE values that an application can generate.
- The qualified name of the stored procedure. This is a VARCHAR(27) value.
- The specific name of the stored procedure. The specific name is a VARCHAR(18) value that is the same as the unqualified name.
- The SQL diagnostic string that is to be returned to DB2. This is a VARCHAR(70) value. Use this area to pass descriptive information about an error or warning to the caller.

SQL is not a valid linkage convention for a REXX language stored procedure. Figure 208 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 208. Parameter convention SQL for a stored procedure

Example of stored procedure linkage convention GENERAL

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL linkage convention to receive parameters. See “Examples of using stored procedures” on page 1075 for examples of complete stored procedures and application programs that call them.

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 like this:
IN V1 INT, OUT V2 CHAR(9)

The following figures show how an assembler, C, COBOL, and PL/I stored procedure uses the GENERAL linkage convention to receive parameters.

Figure 209 shows how a stored procedure in assembler language receives these parameters.

```
*****
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES  *
* THE GENERAL LINKAGE CONVENTION.                             *
*****
A      CEEENTRY AUTO=PROG SIZE,MAIN=YES,PLIST=0S
      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),LOC2  MOVE A VALUE INTO OUTPUT VAR V2
:
      CEE TERM 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
LOC1    DS  F       LOCAL COPY OF PARAMETER V1
LOC2    DS  CL9     LOCAL COPY OF PARAMETER V2
:
PROG SIZE EQU  *-PROGAREA
        CEEDSA ,     MAPPING OF THE DYNAMIC SAVE AREA
        CEECAA ,     MAPPING OF THE COMMON ANCHOR AREA
        END  A
```

Figure 209. An example of GENERAL linkage in assembler

Figure 210 shows how a stored procedure 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
    :
}
}
```

Figure 210. An example of GENERAL linkage in C

Figure 211 shows how a stored procedure 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.

```

Figure 211. An example of *GENERAL* linkage in COBOL

Figure 212 shows how a stored procedure 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 */

```

Figure 212. An example of *GENERAL* linkage in PL/I

Example of stored procedure linkage convention **GENERAL WITH NULLS**

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the **GENERAL WITH NULLS** linkage convention to receive parameters. See “Examples of using stored procedures” on page 1075 for examples of complete stored procedures and application programs that call them.

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 like this:

```
IN V1 INT, OUT V2 CHAR(9)
```

The following figures show how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL WITH NULLS linkage convention to receive parameters.

Figure 213 shows how a stored procedure 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

```

Figure 213. An example of GENERAL WITH NULLS linkage in assembler

Figure 214 shows how a stored procedure 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 1st indicator variable
        if(locind[0]<0)           /* is negative, V1 is null
        {
            :
        }

:
    strcpy(argv[2],locv2);      /* Assign a value to V2
    *(++tempint) = 0;           /* Assign 0 to V2's indicator
                             /* variable
    }
}

```

Figure 214. An example of GENERAL WITH NULLS linkage in C

Figure 215 shows how a stored procedure 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).

```

Figure 215. An example of GENERAL WITH NULLS linkage in COBOL

Figure 216 shows how a stored procedure 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 */

```

Figure 216. An example of GENERAL WITH NULLS linkage in PL/I

Example of stored procedure linkage convention SQL

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the SQL linkage convention to receive parameters. These examples also show how a stored procedure receives the DBINFO structure.

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 variables
*/
*****/
short int ind1;
short int ind2;

:
ind1 = 0;              /* Remember to initialize the
                       /* input parameter's indicator
                       /* variable before executing
                       /* the CALL statement
                       */
EXEC SQL CALL B (:v1 :ind1, :v2 :ind2);

:

```

In the CREATE PROCEDURE statement, the parameters are defined like this:

IN V1 INT, OUT V2 CHAR(9)

The following figures show how an assembler, C, COBOL, or PL/I stored procedure uses the SQL linkage convention to receive parameters.

Figure 217 shows how a stored procedure in assembler language receives these parameters.

```

*****
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES *
* THE SQL 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 SQL LINKAGE *
* CONVENTION IS AS FOLLOWS: *
* ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS. IF N *
* PARAMETERS ARE PASSED, THERE ARE 2N+4 POINTERS. THE FIRST *
* N POINTERS ARE THE ADDRESSES OF THE N PARAMETERS, JUST AS *
* WITH THE GENERAL LINKAGE CONVENTION. THE NEXT N POINTERS ARE *
* 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 LOC1(4),0(R7)    MOVE VALUE INTO LOCAL COPY OF V1
      L R7,8(R1)          GET POINTER TO 1ST INDICATOR VARIABLE
      MVC LOC1(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,LOC11          MOVE VALUE INTO LOCAL STORAGE
      LTR R7,R7            GET INDICATOR VARIABLE FOR V1
      BM NULLIN           CHECK IF IT IS NEGATIVE
                        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

```

Figure 217. An example of SQL linkage in assembler (Part 1 of 2)


```

*****
*   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
LOC11    DS  H          LOCAL COPY OF INDICATOR 1
LOC12    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,CL70     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
ENCODE   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

```

Figure 217. An example of SQL linkage in assembler (Part 2 of 2)

Figure 218 shows how a stored procedure 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 statment 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 */
    }
}

```

Figure 218. An example of SQL linkage for a C stored procedure written as a main program (Part 1 of 2)

```

/*****
/* 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  */
/* 'yyyyyyyy' to the diagnostic string returned   */
/* in the SQLDA associated with the CALL statement.*/
*****/
strcpy(argv[10],"yyyyyyyy/0");
:
}

```

Figure 218. An example of SQL linkage for a C stored procedure written as a main program (Part 2 of 2)

Figure 219 shows how a stored procedure 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[71],     /* Diagnostic string */
           struct sqludf_dbinfo *udf_dbinfo); /* DBINFO */
{
    int l_p1;
    char[11] 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));
    :
    :
}

```

Figure 219. An example of SQL linkage for a C stored procedure written as a subprogram

Figure 220 shows how a stored procedure 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(70).
*****
* 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(20).

```

Figure 220. An example of SQL linkage in COBOL (Part 1 of 2)

```

* 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
*   05 RESD0 PIC X(2).
*   platform type
*   05 PLATFORM PIC 9(9) USAGE BINARY.
*   number of entries in the TF column list array (tfcolumn, below)
*   05 NUMTFCOL PIC 9(4) USAGE BINARY.
*   reserved
*   05 RESD1 PIC X(26).
*   tfcolumn will be allocated dynamically of it is defined
*   otherwise this will be a null pointer
*   05 TFCOLUMN USAGE IS POINTER.
*   application identifier
*   05 APPL-ID USAGE IS POINTER.
*   reserved
*   05 RESD2 PIC X(20).
*
*
*
PROCEDURE DIVISION USING PARM1, PARM2,
                        P-IND1, P-IND2,
                        P-SQLSTATE, P-PROC, P-SPEC, P-DIAG,
                        SQLUDF-DBINFO.
*
*

```

Figure 220. An example of SQL linkage in COBOL (Part 2 of 2)

Figure 221 shows how a stored procedure in the PL/I language receives these parameters.

```

#
# *PROCESS SYSTEM(MVS);
# MYMAIN: PROC(PARM1, PARM2, ...,
#             P_IND1, P_IND2, ...,
#             P_SQLSTATE, P_PROC, P_SPEC, P_DIAG, DBINFO)
#             OPTIONS(MAIN NOEXECOPS REENTRANT);
#
# DCL PARM1 ...           /* first parameter */
# DCL PARM2 ...           /* second parameter */
#
#
# DCL P_IND1 BIN FIXED(15);/* indicator for 1st parm */
# DCL P_IND2 BIN FIXED(15);/* indicator for 2nd parm */
#
#
# DCL P_SQLSTATE CHAR(5); /* SQLSTATE to return to DB2 */
# 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(70) /* Diagnostic string */
# VARYING;
# DCL DBINFO PTR;
#
#

```

Figure 221. An example of SQL linkage in PL/I (Part 1 of 2)

```

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 UDB 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 */
:
#
|

```

Figure 221. An example of SQL linkage in PL/I (Part 2 of 2)

Special considerations for C

In order for the linkage conventions to work correctly when a C language stored procedure runs on z/OS, you must include the following line in your source code:

```
#pragma runopts(PLIST(OS))
```

This option is not applicable to other platforms, however. If you plan to use a C stored procedure on other platforms besides z/OS, use conditional compilation, as shown in Figure 222, to include this option only when you compile on z/OS.

```

#ifdef MVS
#pragma runopts(PLIST(OS))
#endif

-- or --

#ifdef WKSTN
#pragma runopts(PLIST(OS))
#endif

```

Figure 222. Using conditional compilation to include or exclude a statement

Special considerations for PL/I

In order for the linkage conventions to work correctly when a PL/I language stored procedure runs on z/OS, you must do the following:

- Include the run-time option NOEXECOPS in your source code.
- Specify the compile-time option SYSTEM(MVS).

For information about specifying PL/I compile-time and run-time options, see *IBM Enterprise PL/I for z/OS Programming Guide*.

Using indicator variables to speed processing

If any of your output parameters occupy a great deal of storage, it is wasteful to pass the entire storage areas to your stored procedure. You can use indicator variables in the program that call the stored procedure to pass only a two byte area to the stored procedure and receive the entire area from the stored procedure. To accomplish this, declare an indicator variable for every large output parameter in your SQL statement CALL. (If you are using the GENERAL WITH NULLS or SQL linkage convention, you must declare indicator variables for all of your parameters, so you do not need to declare another indicator variable.) Assign a negative value to each indicator variable associated with a large output variable. Then include the indicator variables in the CALL statement. This technique can be used whether the stored procedure linkage convention is GENERAL, GENERAL WITH NULLS, or SQL.

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. A PL/I program containing these statements passes only two 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);
```

Declaring data types for passed parameters

A stored procedure in any language except REXX must declare each parameter passed to it. In addition, the stored procedure definition must contain a compatible SQL data type declaration for each parameter. For information about creating a stored procedure definition, see “Defining your stored procedure to DB2” on page 635.

For REXX: See “Calling a stored procedure from a REXX procedure” on page 714 for information about DB2 data types and corresponding parameter formats.

For languages other than REXX: For all data types except LOBs, ROWIDs, locators, and VARCHARs (for C language), see the tables listed in Table 85 on page 704 for the host data types that are compatible with the data types in the stored procedure definition.

Table 85. Listing of tables of compatible data types

Language	Compatible data types table
Assembler	Table 12 on page 153
C	Table 14 on page 177
COBOL	Table 17 on page 211
PL/I	Table 21 on page 243

For LOBs, ROWIDs, VARCHARs, and locators, Table 86 shows compatible
declarations for the assembler language.

Table 86. 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;

Table 86. Compatible assembler language declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	Assembler declaration

Notes:

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, Table 87 shows compatible declarations for the C language.

Table 87. 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[n]; } var;
CLOB(<i>n</i>)	struct {unsigned long length; char var_data[n]; } var;
DBCLOB(<i>n</i>)	struct {unsigned long length; sqlbchar data[n]; } var;
ROWID	struct {short int length; char data[40]; } var;

For LOBs, ROWIDs, and locators, Table 88 shows compatible declarations for COBOL.

Table 88. 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) USAGE IS BINARY.

Table 88. Compatible COBOL declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	COBOL declaration
BLOB(<i>n</i>)	<pre> If n <= 32767: 01 var. 49 var-LENGTH PIC 9(9) USAGE COMP. 49 var-DATA PIC X(<i>n</i>). If n > 32767: 01 var. 02 var-LENGTH PIC S9(9) USAGE COMP. 02 var-DATA. 49 FILLER PIC X(32767). 49 FILLER PIC X(32767). : 49 FILLER PIC X(mod(<i>n</i>,32767)). </pre>
CLOB(<i>n</i>)	<pre> If n <= 32767: 01 var. 49 var-LENGTH PIC 9(9) USAGE COMP. 49 var-DATA PIC X(<i>n</i>). If n > 32767: 01 var. 02 var-LENGTH PIC S9(9) USAGE COMP. 02 var-DATA. 49 FILLER PIC X(32767). 49 FILLER PIC X(32767). : 49 FILLER PIC X(mod(<i>n</i>,32767)). </pre>
DBCLOB(<i>n</i>)	<pre> If n <= 32767: 01 var. 49 var-LENGTH PIC 9(9) USAGE COMP. 49 var-DATA PIC G(<i>n</i>) USAGE DISPLAY-1. If n > 32767: 01 var. 02 var-LENGTH PIC S9(9) USAGE COMP. 02 var-DATA. 49 FILLER PIC G(32767) USAGE DISPLAY-1. 49 FILLER PIC G(32767). USAGE DISPLAY-1. : 49 FILLER PIC G(mod(<i>n</i>,32767)) USAGE DISPLAY-1. </pre>

Table 88. Compatible COBOL declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	COBOL declaration
ROWID	01 var. 49 var-LEN PIC 9(4) USAGE COMP. 49 var-DATA PIC X(40).

For LOBs, ROWIDs, and locators, Table 89 shows compatible declarations for PL/I.

Table 89. 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));
CLOB(<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 89. Compatible PL/I declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	PL/I
DBCLOB(<i>n</i>)	If <i>n</i> <= 16383: <pre>01 var, 03 var_LENGTH BIN_FIXED(31), 03 var_DATA GRAPHIC(<i>n</i>);</pre> If <i>n</i> > 16383: <pre>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 a stored procedure to return result sets to a DRDA client” on page 650 and “Writing a DB2 UDB for z/OS client program or SQL procedure to receive result sets” 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, allowing the stored procedure to return a 24x80 screen image to the calling program.

Writing a DB2 UDB for z/OS client program or SQL procedure to receive result sets

You can write a program to receive result sets given either of the following alternatives:

- For a fixed number of result sets, for which you know the contents
This is the only alternative in which you can write an SQL procedure to return result sets.
- For a variable number of result sets, for which you do not know the contents

The first alternative is simpler to write, but if you use the second alternative, you do not need to make major modifications to your client program if the stored procedure changes.

The basic steps for receiving result sets are as follows:

1. Declare a locator variable for each result set that will be returned.
If you do not know how many result sets will 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, you can 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 returned by the stored procedure.
- Each SQLVAR entry gives information about a result set. In an SQLVAR entry:
 - The SQLNAME field contains the name of the SQL cursor used by the stored procedure to return the result set.
 - The SQLIND field contains the value -1. This 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.

You can use the SQL statement ASSOCIATE LOCATORS to link result set locators to result sets. The ASSOCIATE LOCATORS statement assigns values to the result set locator variables. If you specify more locators than the number of result sets returned, DB2 ignores the extra locators.

To use the ASSOCIATE LOCATORS statement, you must embed it in an application or SQL procedure.

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 or DESCRIBE PROCEDURE statement must match the stored procedure name in the CALL statement that returns the result sets. That is:

- If the stored procedure name in ASSOCIATE LOCATORS or DESCRIBE PROCEDURE is unqualified, the stored procedure name in the CALL statement must be unqualified.
- If the stored procedure name in ASSOCIATE LOCATORS or DESCRIBE PROCEDURE is qualified with a schema name, the stored procedure name in the CALL statement must be qualified with a schema name.

- If the stored procedure name in ASSOCIATE LOCATORS or DESCRIBE PROCEDURE is qualified with a location name and a schema name, the stored procedure name in the CALL statement must be qualified with a location name and a 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 be different 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, you can 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 big enough to hold descriptions of all columns in the result set. You can use DESCRIBE CURSOR only for 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 the eighth byte 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.
 If you executed the DESCRIBE CURSOR statement, perform these steps before you fetch the rows:
 - a. 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.
 - b. Put the address of the storage for each host variable in the appropriate SQLDATA field of the SQLDA.
 - c. Put the address of the storage for each indicator variable in the appropriate SQLIND field of the SQLDA.
 Fetching rows from a result set is the same as fetching rows from a table.

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

For the syntax of result set locators in each host language, see Chapter 9, "Embedding SQL statements in host languages," on page 143. For the syntax of result set locators in SQL procedures, see Chapter 6 of *DB2 SQL Reference*. For the syntax of the ASSOCIATE LOCATORS, DESCRIBE PROCEDURE, ALLOCATE CURSOR, and DESCRIBE CURSOR statements, see Chapter 5 of *DB2 SQL Reference*.

Figure 223 on page 711 and Figure 224 on page 712 show C language code that accomplishes each of these steps. Coding for other languages is similar. For a more

complete example of a C language program that receives result sets, see “Examples of using stored procedures” on page 1075.

Figure 223 demonstrates how you 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;

:
}
}

```

Figure 223. Receiving known result sets

Figure 224 on page 712 demonstrates how you 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;

:
:

```

Figure 224. Receiving unknown result sets (Part 1 of 3)

```

/*****
/* 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;

```

Figure 224. Receiving unknown result sets (Part 2 of 3)

```

/*****
/* 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;

:
}
}

```

Figure 224. Receiving unknown result sets (Part 3 of 3)

Figure 225 on page 714 demonstrates how you can use an SQL procedure to receive result sets.

```

DECLARE RESULT1 RESULT_SET_LOCATOR VARYING;
DECLARE RESULT2 RESULT_SET_LOCATOR VARYING;
:
:
CALL TARGETPROCEDURE();
ASSOCIATE RESULT SET LOCATORS(RESULT1,RESULT2)
  WITH PROCEDURE TARGETPROCEDURE;
ALLOCATE RSCUR1 CURSOR FOR RESULT1;
ALLOCATE RSCUR2 CURSOR FOR RESULT2;
WHILE AT_END = 0 DO
  FETCH RSCUR1 INTO VAR1;
  SET TOTAL1 = TOTAL1 + VAR1;
END WHILE;
WHILE AT_END = 0 DO
  FETCH RSCUR2 INTO VAR2;
  SET TOTAL2 = TOTAL2 + VAR2;
END WHILE;
:
:

```

Figure 225. Receiving result sets in an SQL procedure

Accessing transition tables in a stored procedure

When you write a user-defined function, external stored procedure, or SQL procedure that is to be invoked from a trigger, you might need access to transition tables for the trigger. The technique for accessing the transition tables is the same for user-defined functions and stored procedures, and is described in “Accessing transition tables in a user-defined function or stored procedure” on page 345.

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. Table 90 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 90. Parameter formats for a CALL statement in a REXX procedure

SQL data type	REXX format
SMALLINT INTEGER	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	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.
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. See Chapter 2 of <i>DB2 SQL Reference</i> for valid date string formats.

Table 90. Parameter formats for a CALL statement in a REXX procedure (continued)

SQL data type	REXX format
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. See Chapter 2 of <i>DB2 SQL Reference</i> for valid time string formats.
TIMESTAMP	A string of length 26, enclosed in single quotation marks. The string has the format <i>yyyy-mm-dd-hh.mm.ss.nnnnnn</i> .

Figure 226 on page 716 demonstrates how a REXX procedure calls the stored procedure in Figure 200 on page 655. 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"||"

```

Figure 226. Example of a REXX procedure that calls a stored procedure (Part 1 of 3)

```

/*****
/* Display the IFI return area in hexadecimal.
*****/
OFFSET = 4+1
TOTLEN = LENGTH(RESET)
DO WHILE ( OFFSET < TOTLEN )
    LEN = C2D(SUBSTR(RESET,OFFSET,2))
    SAY SUBSTR(RESET,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.SQD
    SAY "SQLDA."I".SQLNAME      ="SQLDA.I.SQLNAME";"
    SAY "SQLDA."I".SQLTYPE     ="SQLDA.I.SQLTYPE";"
    SAY "SQLDA."I".SQLLOCATOR  ="SQLDA.I.SQLLOCATOR";"
    SAY "SQLDA."I".SQLESTIMATE="SQLDA.I.SQLESTIMATE";"
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
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

```

Figure 226. Example of a REXX procedure that calls a stored procedure (Part 2 of 3)

```

/*****
/* 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

```

Figure 226. Example of a REXX procedure that calls a stored procedure (Part 3 of 3)

Preparing a client program

You must prepare the calling program by precompiling, compiling, and link-editing it on the client system.

Before you can call a stored procedure from your embedded SQL application, you must bind a package for the client program on the remote system. You can use the remote DRDA bind capability on your DRDA client system to bind the package to the remote system.

If you have packages that contain SQL CALL statements that you bound before DB2 Version 6, you can get better performance from those packages if you rebind them in DB2 Version 6 or later. 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.

For an ODBC or CLI application, the DB2 packages and plan associated with the ODBC driver must be bound to DB2 before you can run your application. For information about building client applications on platforms other than DB2 UDB for z/OS to access stored procedures, see one of these documents:

- *DB2 Universal Database Application Development Guide: Building and Running Applications*
- *DB2 Universal Database for iSeries SQL Programming with Host Languages*

A z/OS client can bind the DBRM to a remote server by specifying a location name on the command BIND PACKAGE. For example, suppose 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)
```

The plan for the package resides only at the client system.

Running a stored procedure

Stored procedures run as either main programs or subprograms. “Writing a stored procedure as a main program or subprogram” on page 643 contains information about the requirements for each type of stored procedure.

If a stored procedure runs as a main program, before each call, Language Environment reinitializes the storage 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 because:

- 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 procedures address spaces.
- The z/OS operator might refresh Language Environment between two executions of your stored procedure.

DB2 runs stored procedures under the DB2 thread of the calling application, making the stored procedures part of the caller’s unit of work.

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 will fail.

If a stored procedure abnormally terminates:

- 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:
 - If the number of abnormal terminations equals the STOP AFTER *n* FAILURES value for the stored procedure
 - If the number of abnormal terminations equals the default MAX ABEND COUNT value for the subsystem

#

- If the stored procedure does not handle the abend condition, DB2 refreshes the Language Environment environment to recover the storage that the application uses. In most cases, the Language Environment environment does not need to restart.
- If a data set is allocated to the DD name CEEDUMP in the JCL procedure that starts the stored procedures address space, Language Environment writes a small diagnostic dump to this data set. See your system administrator to obtain the dump information. See “Testing a stored procedure” on page 725 for techniques that you can use to diagnose the problem.
- 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.

How DB2 determines which version of a stored procedure to run

The combination of the schema name and stored procedure name uniquely identify a stored procedure. If you qualify the stored procedure name when you execute a CALL statement to call a stored procedure, there is only one candidate to run. However, if you do not qualify the stored 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 *literal*

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.

Using a single application program to call different versions of a stored procedure

If you want to use the same application program to call different versions of a stored procedure that have the same load module name, follow these steps:

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).

Running multiple stored procedures concurrently

Multiple stored procedures can run concurrently, each under its own z/OS task (TCB). The maximum number of stored procedures that can run concurrently in a single address space is set at DB2 installation time, on panel DSNTIPX.

See Part 2 of *DB2 Installation Guide* for more information.

You can override that value in the following ways:

- For WLM-established or DB2-established stored procedures address spaces, edit the JCL procedures that start stored procedures address spaces, and modify the value of the NUMTCB parameter.
- For WLM-established address spaces, when you set up a WLM application environment, specify the following parameter in field Start Parameters of panel Create An Application Environment:

NUMTCB=number-of-TCBs

For REXX stored procedures, you must set NUMTCB to 1.

To maximize the number of stored procedures that can run concurrently, use the following guidelines:

- Set REGION size to 0 in startup procedures for the stored procedures address spaces to obtain the largest possible amount of storage below the 16MB line.
- Limit storage required by application programs below the 16MB line by:
 - Link editing programs above the line with AMODE(31) and RMODE(ANY) attributes
 - Using the RENT and DATA(31) compiler options for COBOL programs.
- Limit storage required by IBM Language Environment by using these run-time options:
 - HEAP(,ANY) to allocate program heap storage above the 16MB line
 - STACK(,ANY,) to allocate program stack storage above the 16MB line
 - STORAGE(,,,4K) to reduce reserve storage area below the line to 4KB
 - BELOWHEAP(4K,,) to reduce the heap storage below the line to 4KB
 - LIBSTACK(4K,,) to reduce the library stack below the line to 4KB
 - ALL31(ON) to indicate all programs contained in the stored procedure run with AMODE(31) and RMODE(ANY).

You can list these options in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement, if they are not Language Environment installation defaults. For example, the RUN OPTIONS parameter could specify:

H(,ANY),STAC(,ANY,),STO(,,,4K),BE(4K,,),LIBS(4K,,),ALL31(ON)

For more information about creating a stored procedure definition, see “Defining your stored procedure to DB2” on page 635.

- If you use WLM-established address spaces for your stored procedures, assign stored procedures to WLM application environments according to guidelines that are described in Part 5 (Volume 2) of *DB2 Administration Guide*.

Multiple instances of a stored procedure

Your application program can issue multiple CALL statements to the same local or remote stored procedure. If that stored procedure returns result sets and the calling application leaves those result sets open before the next call to that same stored procedure, each CALL statement invokes a unique instance of the stored procedure. Each instance of the stored procedure 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.

In compatibility mode and enabling-new-function mode, multiple calls to the same
 # stored procedure do not produce multiple instances of the applications.

To invoke multiple instances of remote stored procedures or local stored
 # procedures that have SQL to access a remote site, both the client and server must
 # be in DB2 Version 8 new-function mode or later. For local stored procedures that
 # issue remote SQL, instances of the applications are created at the remote server site
 # regardless of whether result sets exist or are left open between calls.

DB2 storage shortages and EDM POOL FULL conditions can occur if you call too
 # many instances of a stored procedure or if you open too many cursors. 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 optimize storage usage, two subsystem parameters control the maximum
 # number of stored procedures instances and the maximum number of open cursors
 # for a thread. 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 open 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.

The calling application for the stored procedure should close the result sets and
 # issue commits often. Even read-only applications should perform these actions.
 # Applications that fail to do so terminate abnormally with DB2 storage shortage
 # and EDM POOL FULL conditions.

You can set the maximum number of stored procedure instances and the maximum
 # number of open cursors on installation panel DSNTIPX. For more information
 # about setting the maximum number of stored procedure instances and the
 # maximum number of open cursors per DB2 thread or connection, see the topic
 # "Routine parameters panel: DSNTIPX" in *DB2 Installation Guide*.

Accessing non-DB2 resources

Applications that run in a stored procedures address space can access any resources available to z/OS address spaces, such as VSAM files, flat files, APPC/MVS conversations, and IMS or CICS transactions.

Consider the following when you develop stored procedures that access non-DB2 resources:

- When a stored procedure runs in a DB2-established stored procedures address space, DB2 does not coordinate commit and rollback activity on recoverable resources such as IMS or CICS transactions, and MQI messages. DB2 has no knowledge of, and therefore cannot control, the dependency between a stored procedure and a recoverable resource.
- When a stored procedure runs in a WLM-established stored procedures address space, the stored procedure uses the Recoverable Resource Manager Services for commitment control. When DB2 commits or rolls back work in this environment, DB2 coordinates all updates made to recoverable resources by other RRS compliant resource managers in the z/OS system.

- When a stored procedure runs in a DB2-established stored procedures address space, z/OS is not aware that the stored procedures address space is processing work for DB2. One consequence of this is that z/OS accesses RACF-protected resources using the user ID associated with the z/OS task (*ssnmSPAS*) for stored procedures, not the user ID of the client.
- When a stored procedure runs in a WLM-established stored procedures address space, DB2 can establish a RACF environment for accessing non-DB2 resources. The authority used when the stored procedure accesses protected z/OS resources depends on the value of SECURITY in the stored procedure definition:
 - If the value of SECURITY is DB2, the authorization ID associated with the stored procedures address space is used.
 - If the value of SECURITY is USER, the authorization ID under which the CALL statement is executed is used.
 - If the value of SECURITY is DEFINER, the authorization ID under which the CREATE PROCEDURE statement was executed is used.
- Not all non-DB2 resources can tolerate concurrent access by multiple TCBs in the same address space. You might need to serialize the access within your application.

CICS

Stored procedure applications can access CICS by one of these methods:

- **Stored procedure DSNACICS**
DSNACICS gives workstation applications a way to invoke CICS server programs while using TCP/IP or SNA as their communication protocol. The workstation applications use DB2 CONNECT to connect to a DB2 for z/OS subsystem, and then call DSNACICS to invoke the CICS server programs.
- **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

For DB2-established address spaces, a CICS application runs as a separate unit of work from the unit of work under which the stored procedure runs. Consequently, results from CICS processing do not affect the completion of stored procedure processing. For example, a CICS transaction in a stored procedure that rolls back a unit of work does not prevent the stored procedure from committing the DB2 unit of work. Similarly, a rollback of the DB2 unit of work does not undo the successful commit of a CICS transaction.

For WLM-established address spaces, if your system is running a release of CICS that uses z/OS RRS, then z/OS RRS controls commitment of all resources.

IMS

If your system is not running a release of IMS that uses z/OS RRS, you can use one of the following methods to access DL/I data from your stored procedure:

- 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 CPI Communications application programming interface

Testing a stored procedure

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where stored procedures run. Here are some alternative testing strategies to consider.

Debugging the stored procedure as a stand-alone program on a workstation

If you have debugging support on a workstation, you might choose to do most of your development and testing on a workstation, before installing a stored procedure on z/OS. This results in very little debugging activity on z/OS.

Debugging 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. For detailed information about the Debug Tool, see *Debug Tool User's Guide and Reference*.

After you write your COBOL stored procedure and set up the WLM environment, follow these steps to test the stored procedure with the Debug Tool:

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=DSN810.RUNLIB.LOAD
//        DD DISP=SHR,DSN=CEE.SCEERUN
//        DD DISP=SHR,DSN=DSN810.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.

Debugging an SQL procedure or C language stored procedure with the Debug Tool and C/C++ Productivity Tools for z/OS

If you have the C/C++ Productivity Tools for z/OS installed on your workstation and the Debug Tool installed on your z/OS system, you can debug an SQL procedure or C or C++ stored procedure that runs in a WLM-established stored procedures address space. 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. For detailed information about the Debug Tool, see *Debug Tool User's Guide and Reference*.

After you write your C++ stored procedure or SQL procedure and set up the WLM environment, follow these steps 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. For an SQL procedure, the modified C source program that is the output from the **second** precompile step must be 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=DSN810.RUNLIB.LOAD
//        DD DISP=SHR,DSN=CEE.SCEERUN
//        DD DISP=SHR,DSN=DSN810.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.

Debugging with Debug Tool for z/OS interactively and in batch mode

You can use the Debug Tool for z/OS to test z/OS stored procedures written in any of the supported languages 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, do the following:

- Compile the stored procedure with option TEST. This places information in the program that the Debug Tool uses during a debugging session.
- 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,JBJONES%SESSNA:)
```

Table 91 lists the effects each parameter has on the Debug Tool:

Table 91. 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.
JBJONES%SESSNA:	The Debug Tool initiates a session on a workstation identified to APPC/MVS as JBJONES with a session ID of SESSNA.

- 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, do the following:

- 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.

For more information about using the Debug Tool for z/OS, see *Debug Tool User's Guide and Reference*.

Using the MSGFILE run-time option

Language Environment supports the run-time option MSGFILE, which identifies a JCL DD statement used for writing debugging messages. You can use the MSGFILE option to direct debugging messages to a disk file or JES spool file. The following considerations apply:

- For each MSGFILE argument, you must add a DD statement to the JCL procedure used to start the DB2 stored procedures address space.
- Execute ALTER PROCEDURE with the RUN OPTIONS parameter to add the MSGFILE option to the list of run-time options for the stored procedure.
- Because multiple TCBs can be active in the DB2 stored procedures address space, you must serialize I/O to the data set associated with the MSGFILE option. For example:
 - Use the ENQ option of the MSGFILE option to serialize I/O to the message file.
 - To prevent multiple procedures from sharing a data set, each stored procedure can specify a unique DD name with the MSGFILE option.
 - If you debug your applications infrequently or on a DB2 test system, you can serialize I/O by temporarily running the DB2 stored procedures address space with NUMTCB=1 in the stored procedures address space start-up procedure. Ask your system administrator for assistance in doing this.

These considerations also apply to a WLM stored procedures address space.

Using driver applications

You can write a small driver application that calls the stored procedure as a subprogram and passes the parameter list supported by the stored procedure. 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.

Using SQL INSERT statements

You can use SQL statements to insert debugging information into a DB2 table. This allows other machines in the network (such as a workstation) to easily access the data in the table 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.

Chapter 26. Tuning your queries

This chapter tells you how to improve the performance of your queries. It begins with “General tips and questions.”

For more detailed information and suggestions, see:

- “Writing efficient predicates” on page 735
- “General rules about predicate evaluation” on page 739
- “Using host variables efficiently” on page 759
- “Writing efficient subqueries” on page 766
- “Using scrollable cursors efficiently” on page 771
- “Writing efficient queries on tables with data-partitioned secondary indexes” on page 772

If you still have performance problems after you have tried the suggestions in these sections, you can use other, more risky techniques. See “Special techniques to influence access path selection” on page 774 for information.

General tips and questions

Recommendation: If you have a query that is performing poorly, first go over the following checklist to see that you have not overlooked some of the basics:

- “Is the query coded as simply as possible?”
- “Are all predicates coded correctly?”
- “Are there subqueries in your query?” on page 732
- “Does your query involve aggregate functions?” on page 733
- “Do you have an input variable in the predicate of an SQL query?” on page 734
- “Do you have a problem with column correlation?” on page 734
- “Can your query be written to use a noncolumn expression?” on page 734
- “Can materialized query tables help your query performance?” on page 734
- “Does the query contain encrypted data?” on page 735

Is the query coded as simply as possible?

Make sure the SQL query is coded as simply and efficiently as possible. Make sure that no unused columns are selected and that there is no unneeded ORDER BY or GROUP BY.

Are all predicates coded correctly?

Indexable predicates: Make sure all the predicates that you think should be indexable are coded so that they can be indexable. Refer to Table 93 on page 740 to see which predicates are indexable and which are not.

Unintentionally redundant or unnecessary predicates: Try to remove any predicates that are unintentionally redundant or not needed; they can slow down performance.

Declared lengths of host variables: For string comparisons other than equal comparisons, ensure that the declared length of a host variable is less than or equal to the length attribute of the table column that it is compared to. For languages in

which character strings are nul-terminated, the string length can be less than or equal to the column length plus 1. If the declared length of the host variable is greater than the column length, the predicate is stage 1 but cannot be a matching predicate for an index scan.

For example, assume that a host variable and an SQL column are defined as follows:

C language declaration	SQL definition
char string_hv[15]	STRING_COL CHAR(12)

A predicate such as WHERE STRING_COL > :string_hv is not a matching predicate for an index scan because the length of string_hv is greater than the length of STRING_COL. One way to avoid an inefficient predicate using character host variables is to declare the host variable with a length that is less than or equal to the column length:

```
char string_hv[12]
```

Because this is a C language example, the host variable length could be 1 byte greater than the column length:

```
char string_hv[13]
```

For numeric comparisons, a comparison between a DECIMAL column and a float or real host variable is stage 2 if the precision of the DECIMAL column is greater than 15. For example, assume that a host variable and an SQL column are defined as follows:

C language declaration	SQL definition
float float_hv	DECIMAL_COL DECIMAL(16,2)

A predicate such as WHERE DECIMAL_COL = :float_hv is not a matching predicate for an index scan because the length of DECIMAL_COL is greater than 15. However, if DECIMAL_COL is defined as DECIMAL(15,2), the predicate is stage 1 and indexable.

Are there subqueries in your query?

If your query uses subqueries, see “Writing efficient subqueries” on page 766 to understand how DB2 executes subqueries. There are no absolute rules to follow when deciding how or whether to code a subquery. But these are general guidelines:

- If efficient indexes are available on the tables in the subquery, then a correlated subquery is likely to be the most efficient kind of subquery.
- If no efficient indexes are available on the tables in the subquery, then a noncorrelated subquery would be likely to perform better.
- If multiple subqueries are in any parent query, make sure that the subqueries are ordered in the most efficient manner.

Example: Assume that MAIN_TABLE has 1000 rows:

```
SELECT * FROM MAIN_TABLE
  WHERE TYPE IN (subquery 1) AND
         PARTS IN (subquery 2);
```

Assuming that subquery 1 and subquery 2 are the same type of subquery (either correlated or noncorrelated) and the subqueries are stage 2, DB2 evaluates the

subquery predicates in the order they appear in the WHERE clause. Subquery 1 rejects 10% of the total rows, and subquery 2 rejects 80% of the total rows.

The predicate in subquery 1 (which is referred to as P1) is evaluated 1000 times, and the predicate in subquery 2 (which is referred to as P2) is evaluated 900 times, for a total of 1900 predicate checks. However, if the order of the subquery predicates is reversed, P2 is evaluated 1000 times, but P1 is evaluated only 200 times, for a total of 1200 predicate checks.

Coding P2 before P1 appears to be more efficient if P1 and P2 take an equal amount of time to execute. However, if P1 is 100 times faster to evaluate than P2, then coding subquery 1 first might be advisable. If you notice a performance degradation, consider reordering the subqueries and monitoring the results. Consult “Writing efficient subqueries” on page 766 to help you understand what factors make one subquery run more slowly than another.

If you are unsure, run EXPLAIN on the query with both a correlated and a noncorrelated subquery. By examining the EXPLAIN output and understanding your data distribution and SQL statements, you should be able to determine which form is more efficient.

This general principle can apply to all types of predicates. However, because subquery predicates can potentially be thousands of times more processor- and I/O-intensive than all other predicates, the order of subquery predicates is particularly important.

Regardless of coding order, DB2 performs noncorrelated subquery predicates before correlated subquery predicates, unless the subquery is transformed into a join.

Refer to “DB2 predicate manipulation” on page 755 to see in what order DB2 will evaluate predicates and when you can control the evaluation order.

Does your query involve aggregate functions?

If your query involves aggregate functions, make sure that they are coded as simply as possible; this increases the chances that they will be evaluated when the data is retrieved, rather than afterward. In general, an aggregate function performs best when evaluated during data access and next best when evaluated during DB2 sort. Least preferable is to have an aggregate function evaluated after the data has been retrieved. Refer to “When are aggregate functions evaluated? (COLUMN_FN_EVAL)” on page 808 for help in using EXPLAIN to get the information you need.

For aggregate functions to be evaluated during data retrieval, the following conditions must be met for all aggregate functions in the query:

- No sort is needed for GROUP BY. Check this in the EXPLAIN output.
- No stage 2 (residual) predicates exist. Check this in your application.
- No distinct set functions exist, such as COUNT(DISTINCT C1).
- If the query is a join, all set functions must be on the last table joined. Check this by looking at the EXPLAIN output.
- All aggregate functions must be on single columns with no arithmetic expressions.
- The aggregate function is not one of the following aggregate functions:
 - STDDEV

- STDDEV_SAMP
- VAR
- VAR_SAMP

If your query involves the functions MAX or MIN, refer to “One-fetch access (ACCESSTYPE=I1)” on page 814 to see whether your query could take advantage of that method.

Do you have an input variable in the predicate of an SQL query?

When host variables or parameter markers are used in a query, the actual values are not known when you bind the package or plan that contains the query. DB2 therefore uses a default filter factor to determine the best access path for an SQL statement. If that access path proves to be inefficient, you can do several things to obtain a better access path.

See “Using host variables efficiently” on page 759 for more information.

Do you have a problem with column correlation?

Two columns in a table are said to be correlated if the values in the columns do not vary independently.

DB2 might not determine the best access path when your queries include correlated columns. If you think you have a problem with column correlation, see “Column correlation” on page 752 for ideas on what to do about it.

Can your query be written to use a noncolumn expression?

The following predicate combines a column, SALARY, with values that are not from columns on one side of the operator:

```
WHERE SALARY + (:hv1 * SALARY) > 50000
```

If you rewrite the predicate in the following way, DB2 can evaluate it more efficiently:

```
WHERE SALARY > 50000/(1 + :hv1)
```

In the second form, the column is by itself on one side of the operator, and all the other values are on the other side of the operator. The expression on the right is called a *noncolumn expression*. DB2 can evaluate many predicates with noncolumn expressions at an earlier stage of processing called *stage 1*, so the queries take less time to run.

For more information on noncolumn expressions and stage 1 processing, see “Properties of predicates” on page 735.

Can materialized query tables help your query performance?

Dynamic queries that operate on very large amounts of data and involve multiple joins might take a long time to run. One way to improve the performance of these queries is to generate the results of all or parts of the queries in advance, and store the results in *materialized query tables*.

Materialized query tables are user-created tables. Depending on how the tables are defined, they are user-maintained or system-maintained. If you have set subsystem parameters or an application sets special registers to tell DB2 to use materialized

query tables, when DB2 executes a dynamic query, DB2 uses the contents of applicable materialized query tables if DB2 finds a performance advantage to doing so.

For information about materialized query tables, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Does the query contain encrypted data?

Encryption and decryption can degrade the performance of some queries. However, you can lessen the performance impact of encryption and decryption by writing your queries carefully and designing your database with encrypted data in mind. For more information about avoiding performance degradation while using encrypted data, see Part 3 (Volume 1) of *DB2 Administration Guide*.

Writing efficient predicates

Definition: *Predicates* are found in the clauses WHERE, HAVING or ON of SQL statements; they describe attributes of data. They are usually based on the columns of a table and either qualify rows (through an index) or reject rows (returned by a scan) when the table is accessed. The resulting qualified or rejected rows are independent of the access path chosen for that table.

Example: The following query has three predicates: an equal predicate on C1, a BETWEEN predicate on C2, and a LIKE predicate on C3.

```
SELECT * FROM T1
  WHERE C1 = 10 AND
        C2 BETWEEN 10 AND 20 AND
        C3 NOT LIKE 'A%'
```

Effect on access paths: This section explains the effect of predicates on access paths. Because SQL allows you to express the same query in different ways, knowing how predicates affect path selection helps you write queries that access data efficiently.

This section describes:

- “Properties of predicates”
- “General rules about predicate evaluation” on page 739
- “Predicate filter factors” on page 746
- “DB2 predicate manipulation” on page 755
- “Column correlation” on page 752

Properties of predicates

Predicates in a HAVING clause are not used when selecting access paths; hence, in this section the term ‘predicate’ means a predicate after WHERE or ON.

A predicate influences the selection of an access path because of:

- Its **type**, as described in “Predicate types” on page 736
- Whether it is **indexable**, as described in “Indexable and nonindexable predicates” on page 737
- Whether it is **stage 1** or **stage 2**
- Whether it contains a ROWID column, as described in “Is direct row access possible? (PRIMARY_ACCESTYPE = D)” on page 803

There are special considerations for “Predicates in the ON clause” on page 738.

Predicate definitions: Predicates are identified as:

Simple or compound

A *compound* predicate is the result of two predicates, whether simple or compound, connected together by AND or OR Boolean operators. All others are *simple*.

Local or join

Local predicates reference only one table. They are local to the table and restrict the number of rows returned for that table. *Join predicates* involve more than one table or correlated reference. They determine the way rows are joined from two or more tables. For examples of their use, see “Interpreting access to two or more tables (join)” on page 815.

Boolean term

Any predicate that is not contained by a compound OR predicate structure is a *Boolean term*. If a Boolean term is evaluated false for a particular row, the whole WHERE clause is evaluated false for that row.

Predicate types

The type of a predicate depends on its operator or syntax. The type determines what type of processing and filtering occurs when the predicate is evaluated. Table 92 shows the different predicate types.

Table 92. Definitions and examples of predicate types

Type	Definition	Example
Subquery	Any predicate that includes another SELECT statement.	C1 IN (SELECT C10 FROM TABLE1)
Equal	Any predicate that is not a subquery predicate and has an equal operator and no NOT operator. Also included are predicates of the form C1 IS NULL and C1 IS NOT DISTINCT FROM.	C1=100
Range	Any predicate that is not a subquery predicate and has an operator in the following list: >, >=, <, <=, LIKE, or BETWEEN.	C1>100
IN-list	A predicate of the form column IN (list of values).	C1 IN (5,10,15)
NOT	Any predicate that is not a subquery predicate and contains a NOT operator. Also included are predicates of the form C1 IS DISTINCT FROM.	COL1 <> 5 or COL1 NOT BETWEEN 10 AND 20

Example: Influence of type on access paths: The following two examples show how the predicate type can influence DB2’s choice of an access path. In each one, assume that a unique index I1 (C1) exists on table T1 (C1, C2), and that all values of C1 are positive integers.

The following query has a range predicate:

```
SELECT C1, C2 FROM T1 WHERE C1 >= 0;
```

However, the predicate does not eliminate any rows of T1. Therefore, it could be determined during bind that a table space scan is more efficient than the index scan.

The following query has an equal predicate:

```
SELECT * FROM T1 WHERE C1 = 0;
```

DB2 chooses the index access in this case because the index is highly selective on column C1.

Indexable and nonindexable predicates

Definition: *Indexable* predicate types can match index entries; other types cannot. Indexable predicates might not become matching predicates of an index; it depends on the indexes that are available and the access path chosen at bind time.

Examples: If the employee table has an index on the column LASTNAME, the following predicate can be a matching predicate:

```
SELECT * FROM DSN8810.EMP WHERE LASTNAME = 'SMITH';
```

The following predicate cannot be a matching predicate, because it is not indexable.

```
SELECT * FROM DSN8810.EMP WHERE SEX <> 'F';
```

Recommendation: To make your queries as efficient as possible, use indexable predicates in your queries and create suitable indexes on your tables. Indexable predicates allow the possible use of a matching index scan, which is often a very efficient access path.

Stage 1 and stage 2 predicates

Definition: Rows retrieved for a query go through two stages of processing.

1. *Stage 1* predicates (sometimes called *sargable*) can be applied at the first stage.
2. *Stage 2* predicates (sometimes called *nonsargable* or *residual*) cannot be applied until the second stage.

The following items determine whether a predicate is stage 1:

- Predicate syntax

See Table 93 on page 740 for a list of simple predicates and their types. See Examples of predicate properties for information on compound predicate types.

- Type and length of constants or columns in the predicate

A simple predicate whose syntax classifies it as indexable and stage 1 might not be indexable or stage 1 because it contains constants and columns whose lengths are too long.

Example: The following predicate is not indexable:

```
CHARCOL<'ABCDEFGF', where CHARCOL is defined as CHAR(6)
```

The predicate is not indexable because the length of the column is shorter than the length of the constant.

Example: The following predicate is not stage 1:

```
DECCOL>34.5, where DECCOL is defined as DECIMAL(18,2)
```

The predicate is not stage 1 because the precision of the decimal column is greater than 15.

- Whether DB2 evaluates the predicate before or after a join operation. A predicate that is evaluated after a join operation is always a stage 2 predicate.
- Join sequence

The same predicate might be stage 1 or stage 2, depending on the join sequence. *Join sequence* is the order in which DB2 joins tables when it evaluates a query. The join sequence is not necessarily the same as the order in which the tables appear in the predicate.

Example: This predicate might be stage 1 or stage 2:

$T1.C1 = T2.C1 + 1$

If T2 is the first table in the join sequence, the predicate is stage 1, but if T1 is the first table in the join sequence, the predicate is stage 2.

You can determine the join sequence by executing EXPLAIN on the query and examining the resulting plan table. See Chapter 27, “Using EXPLAIN to improve SQL performance,” on page 789 for details.

All indexable predicates are stage 1. The predicate $C1 \text{ LIKE } \%BC$ is stage 1, but is not indexable.

Recommendation: Use stage 1 predicates whenever possible.

Boolean term (BT) predicates

Definition: A *Boolean term predicate*, or *BT predicate*, is a simple or compound predicate that, when it is evaluated false for a particular row, makes the entire WHERE clause false for that particular row.

Examples: In the following query P1, P2 and P3 are simple predicates:

```
SELECT * FROM T1 WHERE P1 AND (P2 OR P3);
```

- P1 is a simple BT predicate.
- P2 and P3 are simple non-BT predicates.
- P2 OR P3 is a compound BT predicate.
- P1 AND (P2 OR P3) is a compound BT predicate.

Effect on access paths: In single-index processing, only Boolean term predicates are chosen for matching predicates. Hence, only indexable Boolean term predicates are candidates for matching index scans. To match index columns by predicates that are not Boolean terms, DB2 considers multiple-index access.

In join operations, Boolean term predicates can reject rows at an earlier stage than can non-Boolean term predicates.

Recommendation: For join operations, choose Boolean term predicates over non-Boolean term predicates whenever possible.

Predicates in the ON clause

The ON clause supplies the join condition in an outer join. For a full outer join, the clause can use only equal predicates. For other outer joins, the clause can use any predicates except predicates that contain subqueries.

For left and right outer joins, and for inner joins, join predicates in the ON clause are treated the same as other stage 1 and stage 2 predicates. A stage 2 predicate in the ON clause is treated as a stage 2 predicate of the inner table.

For full outer join, the ON clause is evaluated during the join operation like a stage 2 predicate.

In an outer join, predicates that are evaluated after the join are stage 2 predicates. Predicates in a table expression can be evaluated before the join and can therefore be stage 1 predicates.

Example: In the following statement, the predicate “EDLEVEL > 100” is evaluated before the full join and is a stage 1 predicate:

```
SELECT * FROM (SELECT * FROM DSN8810.EMP
                WHERE EDLEVEL > 100) AS X FULL JOIN DSN8810.DEPT
                ON X.WORKDEPT = DSN8810.DEPT.DEPTNO;
```

For more information about join methods, see “Interpreting access to two or more tables (join)” on page 815.

General rules about predicate evaluation

Recommendations:

1. In terms of resource usage, the earlier a predicate is evaluated, the better.
2. Stage 1 predicates are better than stage 2 predicates because they disqualify rows earlier and reduce the amount of processing needed at stage 2.
3. When possible, try to write queries that evaluate the most restrictive predicates first. When predicates with a high filter factor are processed first, unnecessary rows are screened as early as possible, which can reduce processing cost at a later stage. However, a predicate’s restrictiveness is only effective among predicates of the same type and the same evaluation stage. For information about filter factors, see “Predicate filter factors” on page 746.

This section contains the following topics:

- “Order of evaluating predicates”
- “Examples of predicate properties” on page 745
- “Predicate filter factors” on page 746
- “Column correlation” on page 752
- “DB2 predicate manipulation” on page 755
- “Predicates with encrypted data” on page 759

Order of evaluating predicates

Two sets of rules determine the order of predicate evaluation.

The first set:

1. Indexable predicates are applied first. All matching predicates on index key columns are applied first and evaluated when the index is accessed.
Next, stage 1 predicates that have not been picked as matching predicates but still refer to index columns are applied to the index. This is called *index screening*.
2. Other stage 1 predicates are applied next.
After data page access, stage 1 predicates are applied to the data.
3. Finally, the stage 2 predicates are applied on the returned data rows.

The second set of rules describes the order of predicate evaluation within each of the stages:

1. All equal predicates (including column IN *list*, where *list* has only one element, or column BETWEEN *value1* AND *value1*) are evaluated.

2. All range predicates and predicates of the form *column* IS NOT NULL are evaluated.
3. All other predicate types are evaluated.

After both sets of rules are applied, predicates are evaluated in the order in which they appear in the query. Because you specify that order, you have some control over the order of evaluation.

Exception: Regardless of coding order, non-correlated subqueries are evaluated before correlated subqueries, unless DB2 transforms the subquery into a join.

Summary of predicate processing

Table 93 lists many of the simple predicates and tells whether those predicates are indexable or stage 1. The following terms are used:

- *subq* means a correlated or noncorrelated subquery.
- *noncor subq* means a noncorrelated subquery.
- *cor subq* means a correlated subquery.
- *op* is any of the operators >, >=, <, <=, >, <.
- *value* is a constant, host variable, or special register.
- *pattern* is any character string that does *not* start with the special characters for percent (%) or underscore (_).
- *char* is any character string that does *not* include the special characters for percent (%) or underscore (_).
- *expression* is any expression that contains arithmetic operators, scalar functions, aggregate functions, concatenation operators, columns, constants, host variables, special registers, or date or time expressions.
- *noncol expr* is a noncolumn expression, which is any expression that does not contain a column. That expression can contain arithmetic operators, scalar functions, concatenation operators, constants, host variables, special registers, or date or time expressions.

An example of a noncolumn expression is

CURRENT DATE - 50 DAYS

- *Tn col expr* is an expression that contains a column in table *Tn*. The expression might be only that column.
- *predicate* is a predicate of any type.

In general, if you form a compound predicate by combining several simple predicates with OR operators, the result of the operation has the same characteristics as the simple predicate that is evaluated latest. For example, if two indexable predicates are combined with an OR operator, the result is indexable. If a stage 1 predicate and a stage 2 predicate are combined with an OR operator, the result is stage 2.

Table 93. Predicate types and processing

Predicate Type	Indexable?	Stage 1?	Notes
COL = <i>value</i>	Y	Y	16
COL = <i>noncol expr</i>	Y	Y	9, 11, 12, 15
COL IS NULL	Y	Y	20, 21
COL <i>op value</i>	Y	Y	13

Table 93. Predicate types and processing (continued)

	Predicate Type	Index- able?	Stage 1?	Notes
	COL <i>op</i> <i>noncol expr</i>	Y	Y	9, 11, 12, 13
	COL BETWEEN <i>value1</i> AND <i>value2</i>	Y	Y	13
#	COL BETWEEN <i>noncol expr1</i> AND <i>noncol expr2</i>	Y	Y	9, 11, 12, 13, 15,23
#	<i>value</i> BETWEEN COL1 AND COL2	N	N	
	COL BETWEEN COL1 AND COL2	N	N	10
	COL BETWEEN <i>expression1</i> AND <i>expression2</i>	Y	Y	6, 7, 11, 12, 13, 14
	COL LIKE ' <i>pattern</i> '	Y	Y	5
	COL IN (<i>list</i>)	Y	Y	17, 18
	COL <> <i>value</i>	N	Y	8, 11
	COL <> <i>noncol expr</i>	N	Y	8, 11
#	COL IS NOT NULL	Y	Y	21
	COL NOT BETWEEN <i>value1</i> AND <i>value2</i>	N	Y	
	COL NOT BETWEEN <i>noncol expr1</i> AND <i>noncol expr2</i>	N	Y	
	<i>value</i> NOT BETWEEN COL1 AND COL2	N	N	
	COL NOT IN (<i>list</i>)	N	Y	
	COL NOT LIKE ' <i>char</i> '	N	Y	5
	COL LIKE '% <i>char</i> '	N	Y	1, 5
	COL LIKE ' <i>_char</i> '	N	Y	1, 5
	COL LIKE <i>host variable</i>	Y	Y	2, 5
	T1.COL = T2 <i>col expr</i>	Y	Y	6, 9, 11, 12, 14, 15, 25
	T1.COL <i>op</i> T2 <i>col expr</i>	Y	Y	6, 9, 11, 12, 13, 14, 15
	T1.COL <> T2 <i>col expr</i>	N	Y	8, 11
	T1.COL1 = T1.COL2	N	N	3, 25
	T1.COL1 <i>op</i> T1.COL2	N	N	3
	T1.COL1 <> T1.COL2	N	N	3
	COL=(<i>noncor subq</i>)	Y	Y	
#	COL = ANY (<i>noncor subq</i>)	N	N	22
	COL = ALL (<i>noncor subq</i>)	N	N	
	COL <i>op</i> (<i>noncor subq</i>)	Y	Y	26 on page 745
#	COL <i>op</i> ANY (<i>noncor subq</i>)	Y	Y	22
	COL <i>op</i> ALL (<i>noncor subq</i>)	Y	Y	
	COL <> (<i>noncor subq</i>)	N	Y	

Table 93. Predicate types and processing (continued)

	Predicate Type	Index- able?	Stage 1?	Notes
#	COL <> ANY (<i>noncor subq</i>)	N	N	22
	COL <> ALL (<i>noncor subq</i>)	N	N	
#	COL IN (<i>noncor subq</i>)	Y	Y	24
	(COL1,...COLn) IN (<i>noncor subq</i>)	Y	Y	
	COL NOT IN (<i>noncor subq</i>)	N	N	
	(COL1,...COLn) NOT IN (<i>noncor subq</i>)	N	N	
	COL = (<i>cor subq</i>)	N	N	4
#	COL = ANY (<i>cor subq</i>)	N	N	22
	COL = ALL (<i>cor subq</i>)	N	N	
	COL op (<i>cor subq</i>)	N	N	4
#	COL op ANY (<i>cor subq</i>)	N	N	22
	COL op ALL (<i>cor subq</i>)	N	N	
	COL <> (<i>cor subq</i>)	N	N	4
#	COL <> ANY (<i>cor subq</i>)	N	N	22
	COL <> ALL (<i>cor subq</i>)	N	N	
	COL IN (<i>cor subq</i>)	N	N	19
	(COL1,...COLn) IN (<i>cor subq</i>)	N	N	
	COL NOT IN (<i>cor subq</i>)	N	N	
	(COL1,...COLn) NOT IN (<i>cor subq</i>)	N	N	
	COL IS DISTINCT FROM <i>value</i>	N	Y	8, 11
	COL IS NOT DISTINCT FROM <i>value</i>	Y	Y	16
	COL IS DISTINCT FROM <i>noncol expr</i>	N	Y	8, 11
	COL IS NOT DISTINCT FROM <i>noncol expr</i>	Y	Y	9, 11, 12, 15
	T1.COL1 IS DISTINCT FROM T2.COL2	N	N	3
	T1.COL1 IS NOT DISTINCT FROM T2.COL2	N	N	3
	T1.COL1 IS DISTINCT FROM T2 <i>col expr</i>	N	Y	8, 11
	T1.COL1 IS NOT DISTINCT FROM T2 <i>col expr</i>	Y	Y	6, 9, 11, 12, 14, 15
	COL IS DISTINCT FROM (<i>noncor subq</i>)	N	Y	
	COL IS NOT DISTINCT FROM (<i>noncor subq</i>)	Y	Y	
#	COL IS DISTINCT FROM ANY (<i>noncor subq</i>)	N	N	22
#	COL IS NOT DISTINCT FROM ANY (<i>noncor subq</i>)	N	N	22
	COL IS DISTINCT FROM ALL (<i>noncor subq</i>)	N	N	
	COL IS NOT DISTINCT FROM ALL (<i>noncor subq</i>)	N	N	4
	COL IS NOT DISTINCT FROM (<i>cor subq</i>)	N	N	4
#	COL IS DISTINCT FROM ANY (<i>cor subq</i>)	N	N	22
#	COL IS DISTINCT FROM ANY (<i>cor subq</i>)	N	N	22
#	COL IS NOT DISTINCT FROM ANY (<i>cor subq</i>)	N	N	22
	COL IS DISTINCT FROM ALL (<i>cor subq</i>)	N	N	

Table 93. Predicate types and processing (continued)

Predicate Type	Index-able?	Stage 1?	Notes
COL IS NOT DISTINCT FROM ALL (<i>cor subq</i>)	N	N	
EXISTS (<i>subq</i>)	N	N	19
NOT EXISTS (<i>subq</i>)	N	N	
<i>expression</i> = <i>value</i>	N	N	
<i>expression</i> <> <i>value</i>	N	N	
<i>expression</i> <i>op</i> <i>value</i>	N	N	
<i>expression</i> <i>op</i> (<i>subq</i>)	N	N	

Notes to Table 93 on page 740:

1. Indexable only if an ESCAPE character is specified and used in the LIKE predicate. For example, COL LIKE '+%char' ESCAPE '+' is indexable.
2. Indexable only if the pattern in the host variable is an indexable constant (for example, host variable='char%').
3. If both COL1 and COL2 are from the same table, access through an index on either one is not considered for these predicates. However, the following query is an exception:

```
SELECT * FROM T1 A, T1 B WHERE A.C1 = B.C2;
```

By using correlation names, the query treats one table as if it were two separate tables. Therefore, indexes on columns C1 and C2 are considered for access.

4. If the subquery has already been evaluated for a given correlation value, then the subquery might not have to be reevaluated.
5. Not indexable or stage 1 if a field procedure exists on that column.
6. The column on the left side of the join sequence must be in a different table from any columns on the right side of the join sequence.
7. The tables that contain the columns in *expression1* or *expression2* must already have been accessed.
8. The processing for WHERE NOT COL = *value* is like that for WHERE COL <> *value*, and so on.
9. If *noncol expr*, *noncol expr1*, or *noncol expr2* is a noncolumn expression of one of these forms, then the predicate is not indexable:
 - *noncol expr* + 0
 - *noncol expr* - 0
 - *noncol expr* * 1
 - *noncol expr* / 1
 - *noncol expr* CONCAT *empty string*
10. COL, COL1, and COL2 can be the same column or different columns. The columns are in the same table.
11. Any of the following sets of conditions make the predicate stage 2:
 - The first value obtained before the predicate is evaluated is DECIMAL(*p,s*), where *p*>15, and the second value obtained before the predicate is evaluated is REAL or FLOAT.
 - The first value obtained before the predicate is evaluated is CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC, and the second value obtained before the predicate is evaluated is DATE, TIME, or TIMESTAMP.

- For string predicates, the coded character set identifier is the same as the identifier for the left side column.
 - For DATE, TIME, and TIMESTAMP predicates, the left side column must be DATE, TIME, or TIMESTAMP.
19. COL IN (*corr subq*) and EXISTS (*corr subq*) predicates might become indexable and stage 1 if they are transformed to a join during processing.
 20. The predicate types COL IS NULL and COL IS NOT NULL are stage 2 predicates when they query a column that is defined as NOT NULL.
 21. If the predicate type is COL IS NULL and the column is defined as NOT NULL, the table is not accessed because C1 cannot be NULL.
 22. The ANY and SOME keywords behave similarly. If a predicate with the ANY keyword is not indexable and not stage 1, a similar predicate with the SOME keyword is not indexable and not stage 1.
 23. Under either of these circumstances, the predicate is stage 2:
 - *noncol expr* is a case expression.
 - *noncol expr* is the product or the quotient of two noncolumn expressions, that product or quotient is an integer value, and COL is a FLOAT or a DECIMAL column.
 24. COL IN (*noncor subq*) is stage 1 for type N access only. Otherwise, it is stage 2.
 25. If the inner table is as EBCDIC or ASCII column and the outer table is a Unicode column, the predicate is stage 1 and indexable.
 26. This type of predicate is not stage 1 when a nullability mismatch is possible.

Examples of predicate properties

Assume that predicate P1 and P2 are simple, stage 1, indexable predicates:

P1 AND P2 is a compound, stage 1, indexable predicate.

P1 OR P2 is a compound, stage 1 predicate, not indexable except by a union of RID lists from two indexes.

The following examples of predicates illustrate the general rules shown in Table 93 on page 740. In each case, assume that there is an index on columns (C1,C2,C3,C4) of the table and that 0 is the lowest value in each column.

- WHERE C1=5 AND C2=7

Both predicates are stage 1 and the compound predicate is indexable. A matching index scan could be used with C1 and C2 as matching columns.

- WHERE C1=5 AND C2>7

Both predicates are stage 1 and the compound predicate is indexable. A matching index scan could be used with C1 and C2 as matching columns.

- WHERE C1>5 AND C2=7

Both predicates are stage 1, but only the first matches the index. A matching index scan could be used with C1 as a matching column.

- WHERE C1=5 OR C2=7

Both predicates are stage 1 but not Boolean terms. The compound is indexable. Multiple-index access for the compound predicate is not possible because there is no index that has C2 as the leading column. For single-index access, C1 and C2 can be only index screening columns.

- WHERE C1=5 OR C2<>7

The first predicate is indexable and stage 1, and the second predicate is stage 1 but not indexable. The compound predicate is stage 1 and not indexable.

- WHERE C1>5 OR C2=7

Both predicates are stage 1 but not Boolean terms. The compound is indexable. Multiple-index access for the compound predicate is not possible because no index has C2 as the leading column. For single-index access, C1 and C2 can be only index screening columns.

- WHERE C1 IN (*cor subq*) AND C2=C1

Both predicates are stage 2 and not indexable. The index is not considered for matching-index access, and both predicates are evaluated at stage 2.

- WHERE C1=5 AND C2=7 AND (C3 + 5) IN (7,8)

The first two predicates only are stage 1 and indexable. The index is considered for matching-index access, and all rows satisfying those two predicates are passed to stage 2 to evaluate the third predicate.

- WHERE C1=5 OR C2=7 OR (C3 + 5) IN (7,8)

The third predicate is stage 2. The compound predicate is stage 2 and all three predicates are evaluated at stage 2. The simple predicates are not Boolean terms and the compound predicate is not indexable.

- WHERE C1=5 OR (C2=7 AND C3=C4)

The third predicate is stage 2. The two compound predicates (C2=7 AND C3=C4) and (C1=5 OR (C2=7 AND C3=C4)) are stage 2. All predicates are evaluated at stage 2.

- WHERE (C1>5 OR C2=7) AND C3 = C4

The compound predicate (C1>5 OR C2=7) is indexable and stage 1. The simple predicate C3=C4 is not stage1; so the index is not considered for matching-index access. Rows that satisfy the compound predicate (C1>5 OR C2=7) are passed to stage 2 for evaluation of the predicate C3=C4.

Predicate filter factors

Definition: The *filter factor* of a predicate is a number between 0 and 1 that estimates the proportion of rows in a table for which the predicate is true. Those rows are said to *qualify* by that predicate.

Example: Suppose that DB2 can determine that column C1 of table T contains only five distinct values: A, D, Q, W and X. In the absence of other information, DB2 estimates that one-fifth of the rows have the value D in column C1. Then the predicate C1='D' has the filter factor 0.2 for table T.

How DB2 uses filter factors: Filter factors affect the choice of access paths by estimating the number of rows qualified by a set of predicates.

For simple predicates, the filter factor is a function of three variables:

1. The literal value in the predicate; for instance, 'D' in the previous example.
2. The operator in the predicate; for instance, '=' in the previous example and '<>' in the negation of the predicate.
3. Statistics on the column in the predicate. In the previous example, those include the information that column T.C1 contains only five values.

Recommendation: Control the first two of those variables when you write a predicate. Your understanding of how DB2 uses filter factors should help you write more efficient predicates.

Values of the third variable, statistics on the column, are kept in the DB2 catalog. You can update many of those values, either by running the utility RUNSTATS or by executing UPDATE for a catalog table. For information about using RUNSTATS,

see . see the discussion of maintaining statistics in the catalog in Part 4 (Volume 1) of *DB2 Administration Guide* For information on updating the catalog manually, see “Updating catalog statistics” on page 785.

If you intend to update the catalog with statistics of your own choice, you should understand how DB2 uses:

- “Default filter factors for simple predicates”
- “Filter factors for uniform distributions”
- “Interpolation formulas” on page 748
- “Filter factors for all distributions” on page 749

Default filter factors for simple predicates

Table 94 lists default filter factors for different types of predicates. DB2 uses those values when no other statistics exist.

Example: The default filter factor for the predicate C1 = 'D' is 1/25 (0.04). If D is actually not close to 0.04, the default probably does not lead to an optimal access path.

Table 94. DB2 default filter factors by predicate type

Predicate Type	Filter Factor
Col = literal	1/25
Col <> literal	1 – (1/25)
Col IS NULL	1/25
Col IS NOT DISTINCT FROM	1/25
Col IS DISTINCT FROM	1 – (1/25)
Col IN (literal list)	(number of literals)/25
Col Op literal	1/3
Col LIKE literal	1/10
Col BETWEEN literal1 and literal2	1/10

Note:

Op is one of these operators: <, <=, >, >=.

Literal is any constant value that is known at bind time.

Filter factors for uniform distributions

DB2 uses the filter factors in Table 95 if:

- There is a positive value in column COLCARDF of catalog table SYSIBM.SYSCOLUMNS for the column “Col”.
- There are no additional statistics for “Col” in SYSIBM.SYSCOLDIST.

Example: If D is one of only five values in column C1, using RUNSTATS puts the value 5 in column COLCARDF of SYSCOLUMNS. If there are no additional statistics available, the filter factor for the predicate C1 = 'D' is 1/5 (0.2).

Table 95. DB2 uniform filter factors by predicate type

Predicate type	Filter factor
Col = literal	1/COLCARDF
Col <> literal	1 – (1/COLCARDF)
Col IS NULL	1/COLCARDF
Col IS NOT DISTINCT FROM	1/COLCARDF

Table 95. DB2 uniform filter factors by predicate type (continued)

Predicate type	Filter factor
Col IS DISTINCT FROM	$1 - (1/\text{COLCARD})$
Col IN (literal list)	number of literals /COLCARD
Col Op1 literal	interpolation formula
Col Op2 literal	interpolation formula
Col LIKE literal	interpolation formula
Col BETWEEN literal1 and literal2	interpolation formula

Note:

- Op1 is < or <=, and the literal is not a host variable.
- Op2 is > or >=, and the literal is not a host variable.
- Literal is any constant value that is known at bind time.

Filter factors for other predicate types: The examples selected in Table 94 on page 747 and Table 95 on page 747 represent only the most common types of predicates. If P1 is a predicate and F is its filter factor, then the filter factor of the predicate NOT P1 is (1 - F). But, filter factor calculation is dependent on many things, so a specific filter factor cannot be given for all predicate types.

Interpolation formulas

Definition: For a predicate that uses a range of values, DB2 calculates the filter factor by an *interpolation formula*. The formula is based on an estimate of the ratio of the number of values in the range to the number of values in the entire column of the table.

The formulas: The formulas that follow are rough estimates, subject to further modification by DB2. They apply to a predicate of the form *col op. literal*. The value of (Total Entries) in each formula is estimated from the values in columns HIGH2KEY and LOW2KEY in catalog table SYSIBM.SYSCOLUMNS for column *col*: Total Entries = (HIGH2KEY value - LOW2KEY value).

- For the operators < and <=, where the literal is not a host variable:
(Literal value - LOW2KEY value) / (Total Entries)
- For the operators > and >=, where the literal is not a host variable:
(HIGH2KEY value - Literal value) / (Total Entries)
- For LIKE or BETWEEN:
(High literal value - Low literal value) / (Total Entries)

Example: For column C2 in a predicate, suppose that the value of HIGH2KEY is 1400 and the value of LOW2KEY is 200. For C2, DB2 calculates (Total Entries) = 1200.

For the predicate C1 BETWEEN 800 AND 1100, DB2 calculates the filter factor F as:
 $F = (1100 - 800)/1200 = 1/4 = 0.25$

Interpolation for LIKE: DB2 treats a LIKE predicate as a type of BETWEEN predicate. Two values that bound the range qualified by the predicate are generated from the literal string in the predicate. Only the leading characters found before the escape character ('%' or '_') are used to generate the bounds. So if the escape character is the first character of the string, the filter factor is estimated as 1, and the predicate is estimated to reject no rows.

Defaults for interpolation: DB2 might not interpolate in some cases; instead, it can use a default filter factor. Defaults for interpolation are:

- Relevant only for ranges, including LIKE and BETWEEN predicates
- Used only when interpolation is not adequate
- Based on the value of COLCARDF
- Used whether uniform or additional distribution statistics exist on the column if either of the following conditions is met:
 - The predicate does not contain constants
 - COLCARDF < 4.

Table 96 shows interpolation defaults for the operators <, <=, >, >= and for LIKE and BETWEEN.

Table 96. Default filter factors for interpolation

COLCARDF	Factor for Op	Factor for LIKE or BETWEEN
>=100000000	1/10,000	3/100000
>=10000000	1/3,000	1/10000
>=1000000	1/1,000	3/10000
>=100000	1/300	1/1000
>=10000	1/100	3/1000
>=1000	1/30	1/100
>=100	1/10	3/100
>=2	1/3	1/10
=1	1/1	1/1
<=0	1/3	1/10

Note: Op is one of these operators: <, <=, >, >=.

Filter factors for all distributions

RUNSTATS can generate additional statistics for a column or set of columns. DB2 can use that information to calculate filter factors. DB2 collects two kinds of distribution statistics:

Frequency

The percentage of rows in the table that contain a value for a column or set of columns

Cardinality

The number of distinct values in a set of columns

When they are used: Table 97 on page 750 lists the types of predicates on which these statistics are used.

Table 97. Predicates for which distribution statistics are used

Type of statistic	Single column or concatenated columns	Predicates
Frequency	Single	COL= <i>literal</i> COL IS NULL COL IN (<i>literal-list</i>) COL <i>op</i> <i>literal</i> COL BETWEEN <i>literal</i> AND <i>literal</i> COL= <i>host-variable</i> COL1=COL2 T1.COL=T2.COL COL IS NOT DISTINCT FROM
Frequency	Concatenated	COL= <i>literal</i> COL IS NOT DISTINCT FROM
Cardinality	Single	COL= <i>literal</i> COL IS NULL COL IN (<i>literal-list</i>) COL <i>op</i> <i>literal</i> COL BETWEEN <i>literal</i> AND <i>literal</i> COL= <i>host-variable</i> COL1=COL2 T1.COL=T2.COL COL IS NOT DISTINCT FROM
Cardinality	Concatenated	COL= <i>literal</i> COL= <i>host-variable</i> COL1=COL2 COL IS NOT DISTINCT FROM

Note: *op* is one of these operators: <, <=, >, >=.

How they are used: Columns COLVALUE and FREQUENCYF in table SYSCOLDIST contain distribution statistics. Regardless of the number of values in those columns, running RUNSTATS deletes the existing values and inserts rows for frequent values.

You can run RUNSTATS without the FREQVAL option, with the FREQVAL option in the *correl-spec*, with the FREQVAL option in the *colgroup-spec*, or in both, with the following effects:

- If you run RUNSTATS without the FREQVAL option, RUNSTATS inserts rows for the 10 most frequent values for the first column of the specified index.
- If you run RUNSTATS with the FREQVAL option in the *correl-spec*, RUNSTATS inserts rows for concatenated columns of an index. The NUMCOLS option specifies the number of concatenated index columns. The COUNT option specifies the number of frequent values. You can collect most-frequent values, least-frequent values, or both.
- If you run RUNSTATS with the FREQVAL option in the *colgroup-spec*, RUNSTATS inserts rows for the columns in the column group that you specify. The COUNT option specifies the number of frequent values. You can collect most-frequent values, least-frequent values, or both.
- If you specify the FREQVAL option, RUNSTATS inserts rows for columns of the specified index and for columns in a column group.

See Part 2 of *DB2 Utility Guide and Reference* for more information about RUNSTATS. DB2 uses the frequencies in column FREQUENCYF for predicates that use the values in column COLVALUE and assumes that the remaining data are uniformly distributed.

Example: Filter factor for a single column

Suppose that the predicate is C1 IN ('3','5') and that SYSCOLDIST contains these values for column C1:

COLVALUE	FREQUENCYF
'3'	.0153
'5'	.0859
'8'	.0627

The filter factor is $.0153 + .0859 = .1012$.

Example: Filter factor for correlated columns

Suppose that columns C1 and C2 are correlated. Suppose also that the predicate is C1='3' AND C2='5' and that SYSCOLDIST contains these values for columns C1 and C2:

COLVALUE	FREQUENCYF
'1' '1'	.1176
'2' '2'	.0588
'3' '3'	.0588
'3' '5'	.1176
'4' '4'	.0588
'5' '3'	.1764
'5' '5'	.3529
'6' '6'	.0588

The filter factor is .1176.

Using multiple filter factors to determine the cost of a query

When DB2 estimates the cost of a query, it determines the filter factor repeatedly and at various levels. For example, suppose that you execute the following query:

```
SELECT COLS FROM T1
WHERE C1 = 'A'
AND C3 = 'B'
AND C4 = 'C';
```

Table T1 consists of columns C1, C2, C3, and C4. Index I1 is defined on table T1 and contains columns C1, C2, and C3.

Suppose that the simple predicates in the compound predicate have the following characteristics:

C1='A'	Matching predicate
C3='B'	Screening predicate
C4='C'	Stage 1, nonindexable predicate

To determine the cost of accessing table T1 through index I1, DB2 performs these steps:

1. Estimates the matching index cost. DB2 determines the index matching filter factor by using single-column cardinality and single-column frequency statistics because only one column can be a matching column.

2. Estimates the total index filtering. This includes matching and screening filtering. If statistics exist on column group (C1,C3), DB2 uses those statistics. Otherwise DB2 uses the available single-column statistics for each of these columns.
DB2 will also use FULLKEYCARD as a bound. Therefore, it can be critical to have column group statistics on column group (C1, C3) to get an accurate estimate.
3. Estimates the table-level filtering. If statistics are available on column group (C1,C3,C4), DB2 uses them. Otherwise, DB2 uses statistics that exist on subsets of those columns.

Important: If you supply appropriate statistics at each level of filtering, DB2 is more likely to choose the most efficient access path.

You can use RUNSTATS to collect any of the needed statistics.

Column correlation

Two columns of data, A and B of a single table, are correlated if the values in column A do not vary independently of the values in column B.

Example: Table 98 is an excerpt from a large single table. Columns CITY and STATE are highly correlated, and columns DEPTNO and SEX are entirely independent.

Table 98. Data from the CREWINFO table

CITY	STATE	DEPTNO	SEX	EMPNO	ZIPCODE
Fresno	CA	A345	F	27375	93650
Fresno	CA	J123	M	12345	93710
Fresno	CA	J123	F	93875	93650
Fresno	CA	J123	F	52325	93792
New York	NY	J123	M	19823	09001
New York	NY	A345	M	15522	09530
Miami	FL	B499	M	83825	33116
Miami	FL	A345	F	35785	34099
Los Angeles	CA	X987	M	12131	90077
Los Angeles	CA	A345	M	38251	90091

In this simple example, for every value of column CITY that equals 'FRESNO', there is the same value in column STATE ('CA').

How to detect column correlation

The first indication that column correlation is a problem is because of poor response times when DB2 has chosen an inappropriate access path. If you suspect two columns in a table (CITY and STATE in table CREWINFO) are correlated, then you can issue the following SQL queries that reflect the relationships between the columns:

```
SELECT COUNT (DISTINCT CITY) AS CITYCOUNT,
       COUNT (DISTINCT STATE) AS STATECOUNT FROM CREWINFO;
```

The result of the count of each distinct column is the value of COLCARD in the DB2 catalog table SYSCOLUMNS. Multiply the previous two values together to get a preliminary result:

CITYCOUNT x STATECOUNT = ANSWER1

Then issue the following SQL statement:

```
SELECT COUNT(*) FROM
  (SELECT DISTINCT CITY, STATE
   FROM CREWINFO) AS V1;      (ANSWER2)
```

Compare the result of the previous count (ANSWER2) with ANSWER1. If ANSWER2 is less than ANSWER1, then the suspected columns are correlated.

Impacts of column correlation

DB2 might not determine the best access path, table order, or join method when your query uses columns that are highly correlated. Column correlation can make the estimated cost of operations cheaper than they actually are. Column correlation affects both single table queries and join queries.

Column correlation on the best matching columns of an index: The following query selects rows with females in department A345 from Fresno, California. Two indexes are defined on the table, Index 1 (CITY,STATE,ZIPCODE) and Index 2 (DEPTNO,SEX).

Query 1

```
SELECT ... FROM CREWINFO WHERE
  CITY = 'FRESNO' AND STATE = 'CA'      (PREDICATE1)
  AND DEPTNO = 'A345' AND SEX = 'F';    (PREDICATE2)
```

Consider the two compound predicates (labeled PREDICATE1 and PREDICATE2), their actual filtering effects (the proportion of rows they select), and their DB2 filter factors. Unless the proper catalog statistics are gathered, the filter factors are calculated as if the columns of the predicate are entirely independent (not correlated).

When the columns in a predicate correlate but the correlation is not reflected in catalog statistics, the actual filtering effect to be significantly different from the DB2 filter factor. Table 99 on page 754 shows how the actual filtering effect and the DB2 filter factor can differ, and how that difference can affect index choice and performance.

Table 99. Effects of column correlation on matching columns

	INDEX 1	INDEX 2
Matching predicates	Predicate1 CITY=FRESNO AND STATE=CA	Predicate2 DEPTNO=A345 AND SEX=F
Matching columns	2	2
DB2 estimate for matching columns (Filter Factor)	column=CITY, COLCARDF=4 Filter Factor=1/4 column=STATE, COLCARDF=3 Filter Factor=1/3	column=DEPTNO, COLCARDF=4 Filter Factor=1/4 column=SEX, COLCARDF=2 Filter Factor=1/2
Compound filter factor for matching columns	$1/4 \times 1/3 = 0.083$	$1/4 \times 1/2 = 0.125$
Qualified leaf pages based on DB2 estimations	$0.083 \times 10 = 0.83$ INDEX CHOSEN (.8 < 1.25)	$0.125 \times 10 = 1.25$
Actual filter factor based on data distribution	4/10	2/10
Actual number of qualified leaf pages based on compound predicate	$4/10 \times 10 = 4$	$2/10 \times 10 = 2$ BETTER INDEX CHOICE (2 < 4)

DB2 chooses an index that returns the fewest rows, partly determined by the smallest filter factor of the matching columns. Assume that filter factor is the only influence on the access path. The combined filtering of columns CITY and STATE seems very good, whereas the matching columns for the second index do not seem to filter as much. Based on those calculations, DB2 chooses Index 1 as an access path for Query 1.

The problem is that the filtering of columns CITY and STATE should not look good. Column STATE does almost no filtering. Since columns DEPTNO and SEX do a better job of filtering out rows, DB2 should favor Index 2 over Index 1.

Column correlation on index screening columns of an index: Correlation might also occur on nonmatching index columns, used for index screening. See “Nonmatching index scan (ACCESSTYPE=I and MATCHCOLS=0)” on page 812 for more information. Index screening predicates help reduce the number of data rows that qualify while scanning the index. However, if the index screening predicates are correlated, they do not filter as many data rows as their filter factors suggest. To illustrate this, use “Query 1” on page 753 with the following indexes on Table 98 on page 752:

Index 3 (EMPNO,CITY,STATE)
Index 4 (EMPNO,DEPTNO,SEX)

In the case of Index 3, because the columns CITY and STATE of Predicate 1 are correlated, the index access is not improved as much as estimated by the screening predicates and therefore Index 4 might be a better choice. (Note that index screening also occurs for indexes with matching columns greater than zero.)

Multiple table joins: In Query 2, Table 100 on page 755 is added to the original query (see “Query 1” on page 753) to show the impact of column correlation on join queries.

Table 100. Data from the DEPTINFO table

CITY	STATE	MANAGER	DEPT	DEPTNAME
Fresno	CA	Smith	J123	ADMIN
Los Angeles	CA	Jones	A345	LEGAL

Query 2

```
SELECT ... FROM CREWINFO T1,DEPTINFO T2
WHERE T1.CITY = 'FRESNO' AND T1.STATE='CA'          (PREDICATE 1)
AND T1.DEPTNO = T2.DEPT AND T2.DEPTNAME = 'LEGAL';
```

The order that tables are accessed in a join statement affects performance. The estimated combined filtering of Predicate1 is lower than its actual filtering. So table CREWINFO might look better as the first table accessed than it should.

Also, due to the smaller estimated size for table CREWINFO, a nested loop join might be chosen for the join method. But, if many rows are selected from table CREWINFO because Predicate1 does not filter as many rows as estimated, then another join method or join sequence might be better.

What to do about column correlation

If column correlation is causing DB2 to choose an inappropriate access path, try one of these techniques to alter the access path:

- For leading indexed columns, run the RUNSTATS utility with the KEYCARD option determine the column correlation. For all other column groups, run the RUNSTATS utility with the COLGROUP option.
- Run the RUNSTATS utility to collect column correlation information for any column group with the COLGROUP option.
- Update the catalog statistics manually.
- Use SQL that forces access through a particular index.

The last two techniques are discussed in “Special techniques to influence access path selection” on page 774.

The RUNSTATS utility collects the statistics DB2 needs to make proper choices about queries. With RUNSTATS, you can collect statistics on the concatenated key columns of an index and the number of distinct values for those concatenated columns. This gives DB2 accurate information to calculate the filter factor for the query.

Example: RUNSTATS collects statistics that benefit queries like this:

```
SELECT * FROM T1
WHERE C1 = 'a' AND C2 = 'b' AND C3 = 'c' ;
```

where:

- The first three index keys are used (MATCHCOLS = 3).
- An index exists on C1, C2, C3, C4, C5.
- Some or all of the columns in the index are correlated in some way.

See Part 5 (Volume 2) of *DB2 Administration Guide* for information on using RUNSTATS to influence access path selection.

DB2 predicate manipulation

In some specific cases, DB2 either modifies some predicates, or generates extra predicates. Although these modifications are transparent to you, they have a direct

impact on the access path selection and your PLAN_TABLE results. This is because DB2 always uses an index access path when it is cost effective. Generating extra predicates provides more indexable predicates potentially, which creates more chances for an efficient index access path.

Therefore, to understand your PLAN_TABLE results, you must understand how DB2 manipulates predicates. The information in Table 93 on page 740 is also helpful.

Predicate modifications for IN-list predicates

If an IN-list predicate has only one item in its list, the predicate becomes an EQUAL predicate.

A set of simple, Boolean term, equal predicates on the same column that are connected by OR predicates can be converted into an IN-list predicate. For example: C1=5 or C1=10 or C1=15 converts to C1 IN (5,10,15).

When DB2 simplifies join operations

Because full outer joins are less efficient than left or right joins, and left and right joins are less efficient than inner joins, you should always try to use the simplest type of join operation in your queries. However, if DB2 encounters a join operation that it can simplify, it attempts to do so. In general, DB2 can simplify a join operation when the query contains a predicate or an ON clause that eliminates the null values that are generated by the join operation.

Example: Consider this query:

```
SELECT * FROM T1 X FULL JOIN T2 Y
  ON X.C1=Y.C1
 WHERE X.C2 > 12;
```

The outer join operation gives you these result table rows:

- The rows with matching values of C1 in tables T1 and T2 (the inner join result)
- The rows from T1 where C1 has no corresponding value in T2
- The rows from T2 where C1 has no corresponding value in T1

However, when you apply the predicate, you remove all rows in the result table that came from T2 where C1 has no corresponding value in T1. DB2 transforms the full join into a left join, which is more efficient:

```
SELECT * FROM T1 X LEFT JOIN T2 Y
  ON X.C1=Y.C1
 WHERE X.C2 > 12;
```

Example: The predicate, X.C2>12, filters out all null values that result from the right join:

```
SELECT * FROM T1 X RIGHT JOIN T2 Y
  ON X.C1=Y.C1
 WHERE X.C2>12;
```

Therefore, DB2 can transform the right join into a more efficient inner join without changing the result:

```
SELECT * FROM T1 X INNER JOIN T2 Y
  ON X.C1=Y.C1
 WHERE X.C2>12;
```

The predicate that follows a join operation must have the following characteristics before DB2 transforms an outer join into a simpler outer join or into an inner join:

- The predicate is a Boolean term predicate.
- The predicate is false if one table in the join operation supplies a null value for all of its columns.

These predicates are examples of predicates that can cause DB2 to simplify join operations:

```
T1.C1 > 10
T1.C1 IS NOT NULL
T1.C1 > 10 OR T1.C2 > 15
T1.C1 > T2.C1
T1.C1 IN (1,2,4)
T1.C1 LIKE 'ABC%'
T1.C1 BETWEEN 10 AND 100
12 BETWEEN T1.C1 AND 100
```

Example: This examples shows how DB2 can simplify a join operation because the query contains an ON clause that eliminates rows with unmatched values:

```
SELECT * FROM T1 X LEFT JOIN T2 Y
  FULL JOIN T3 Z ON Y.C1=Z.C1
ON X.C1=Y.C1;
```

Because the last ON clause eliminates any rows from the result table for which column values that come from T1 or T2 are null, DB2 can replace the full join with a more efficient left join to achieve the same result:

```
SELECT * FROM T1 X LEFT JOIN T2 Y
  LEFT JOIN T3 Z ON Y.C1=Z.C1
ON X.C1=Y.C1;
```

In one case, DB2 transforms a full outer join into a left join when you cannot write code to do it. This is the case where a view specifies a full outer join, but a subsequent query on that view requires only a left outer join.

Example: Consider this view:

```
CREATE VIEW V1 (C1,T1C2,T2C2) AS
  SELECT COALESCE(T1.C1, T2.C1), T1.C2, T2.C2
  FROM T1 X FULL JOIN T2 Y
  ON T1.C1=T2.C1;
```

This view contains rows for which values of C2 that come from T1 are null. However, if you execute the following query, you eliminate the rows with null values for C2 that come from T1:

```
SELECT * FROM V1
  WHERE T1C2 > 10;
```

Therefore, for this query, a left join between T1 and T2 would have been adequate. DB2 can execute this query as if the view V1 was generated with a left outer join so that the query runs more efficiently.

Predicates generated through transitive closure

When the set of predicates that belong to a query logically imply other predicates, DB2 can generate additional predicates to provide more information for access path selection.

Rules for generating predicates: For single-table or inner join queries, DB2 generates predicates for transitive closure if:

- The query has an equal type predicate: COL1=COL2. This could be:

- A local predicate
- A join predicate
- The query also has a Boolean term predicate on one of the columns in the first predicate with one of the following formats:
 - COL1 *op* *value*
 op is =, <>, >, >=, <, or <=.
 value is a constant, host variable, or special register.
 - COL1 (NOT) BETWEEN *value1* AND *value2*
 - COL1=COL3

For outer join queries, DB2 generates predicates for transitive closure if the query has an ON clause of the form COL1=COL2 and a before join predicate that has one of the following formats:

- COL1 *op* *value*
 op is =, <>, >, >=, <, or <=
- COL1 (NOT) BETWEEN *value1* AND *value2*

DB2 generates a transitive closure predicate for an outer join query only if the generated predicate does not reference the table with unmatched rows. That is, the generated predicate cannot reference the left table for a left outer join or the right table for a right outer join.

For a multiple-CCSID query, DB2 does not generate a transitive closure predicate if the predicate that would be generated has these characteristics:

- The generated predicate is a range predicate (*op* is >, >=, <, or <=).
- Evaluation of the query with the generated predicate results in different CCSID conversion from evaluation of the query without the predicate. See Chapter 4 of *DB2 SQL Reference* for information on CCSID conversion.

When a predicate meets the transitive closure conditions, DB2 generates a new predicate, whether or not it already exists in the WHERE clause.

The generated predicates have one of the following formats:

- COL *op* *value*
 op is =, <>, >, >=, <, or <=.
 value is a constant, host variable, or special register.
- COL (NOT) BETWEEN *value1* AND *value2*
- COL1=COL2 (for single-table or inner join queries only)

Example of transitive closure for an inner join: Suppose that you have written this query, which meets the conditions for transitive closure:

```
SELECT * FROM T1, T2
  WHERE T1.C1=T2.C1 AND
        T1.C1>10;
```

DB2 generates an additional predicate to produce this query, which is more efficient:

```
SELECT * FROM T1, T2
  WHERE T1.C1=T2.C1 AND
        T1.C1>10 AND
        T2.C1>10;
```

Example of transitive closure for an outer join: Suppose that you have written this outer join query:

```
SELECT * FROM
  (SELECT T1.C1 FROM T1 WHERE T1.C1>10) X
LEFT JOIN
  (SELECT T2.C1 FROM T2) Y
ON X.C1 = Y.C1;
```

The before join predicate, T1.C1>10, meets the conditions for transitive closure, so DB2 generates a query that has the same result as this more-efficient query:

```
SELECT * FROM
  (SELECT T1.C1 FROM T1 WHERE T1.C1>10) X
LEFT JOIN
  (SELECT T2.C1 FROM T2 WHERE T2.C1>10) Y
ON X.C1 = Y.C1;
```

Predicate redundancy: A predicate is redundant if evaluation of other predicates in the query already determines the result that the predicate provides. You can specify redundant predicates or DB2 can generate them. DB2 does not determine that any of your query predicates are redundant. All predicates that you code are evaluated at execution time regardless of whether they are redundant. If DB2 generates a redundant predicate to help select access paths, that predicate is ignored at execution.

Adding extra predicates: DB2 performs predicate transitive closure only on equal and range predicates. However, you can help DB2 to choose a better access path by adding transitive closure predicates for other types of operators, such as IN or LIKE. For example, consider the following SELECT statement:

```
SELECT * FROM T1,T2
WHERE T1.C1=T2.C1
AND T1.C1 LIKE 'A%';
```

If T1.C1=T2.C1 is true, and T1.C1 LIKE 'A%' is true, then T2.C1 LIKE 'A%' must also be true. Therefore, you can give DB2 extra information for evaluating the query by adding T2.C1 LIKE 'A%':

```
SELECT * FROM T1,T2
WHERE T1.C1=T2.C1
AND T1.C1 LIKE 'A%'
AND T2.C1 LIKE 'A%';
```

Predicates with encrypted data

DB2 provides built-in functions for data encryption and decryption. These functions can secure sensitive data, but they can also degrade the performance of some statements if they are not used carefully. If a predicate contains any operator other than = and <>, encrypted data must be decrypted before comparisons can be made. Decryption makes the predicates stage 2. For advice on avoiding unnecessary encryption and decryption for predicate evaluation, see Part 3 (Volume 1) of *DB2 Administration Guide*.

Using host variables efficiently

Host variables require default filter factors: When you bind a static SQL statement that contains host variables, DB2 uses a default filter factor to determine the best access path for the SQL statement. For more information on filter factors, including default values, see “Predicate filter factors” on page 746.

DB2 often chooses an access path that performs well for a query with several host variables. However, in a new release or after maintenance has been applied, DB2 might choose a new access path that does not perform as well as the old access path. In many cases, the change in access paths is due to the default filter factors, which might lead DB2 to optimize the query in a different way.

The two ways to change the access path for a query that contains host variables are:

- Bind the package or plan that contains the query with the option REOPT(ALWAYS) or the option REOPT(ONCE).
- Rewrite the query.

Changing the access path at run time

You can use the following bind options to control how DB2 determines the access path for SQL statements with variable values:

REOPT(ALWAYS)	DB2 determines the access path for any SQL statement with variable values each time the statement is run.
REOPT(ONCE)	DB2 determines and caches the access path for any SQL statement with variable values only once at run time, using the first set of input variable values. If the statement is run multiple times, DB2 does not reoptimize each time unless the cached statement is invalidated or removed from the cache.
REOPT(NONE)	DB2 determines the access path at bind time, and does not change the access path at run time.

The REOPT(ALWAYS) bind option

Specify the REOPT(ALWAYS) bind option when you want DB2 to determine access paths at both bind time and run time for statements that contain host variables and special registers. At run time, DB2 uses the values in these variables to determine the access paths. If the statement runs multiple times, DB2 determines the access path each time that the statement runs.

Consider using the REOPT(ALWAYS) bind option in the following circumstances:

- The SQL statement does not perform well with the access path that is chosen at bind time.
- The SQL statement takes a relatively long time to execute. For long-running SQL statements, the performance gain from the better access path that is chosen based on the input variable values for each run can be greater than the performance cost of reoptimizing the access path each time that the statement runs.

To use the REOPT(ALWAYS) bind option most efficiently, first determine which SQL statements in your applications perform poorly with the REOPT(NONE) bind option and the REOPT(ONCE) bind option. Separate the code containing those statements into units that you bind into packages with the REOPT(ALWAYS) option. Bind the rest of the code into packages using the REOPT(NONE) bind option or the REOPT(ONCE) bind option, as appropriate. Then bind the plan with the REOPT(NONE) bind option. Statements in the packages bound with REOPT(ALWAYS) are candidates for repeated reoptimization at run time.

Example: To determine which queries in plans and packages that are bound with the REOPT(ALWAYS) bind option will be reoptimized at run time, execute the following SELECT statements:

```
SELECT PLNAME,
       CASE WHEN STMTNOI <> 0
         THEN STMTNOI
         ELSE STMTNO
       END AS STMTNUM,
       SEQNO, TEXT
FROM   SYSIBM.SYSSTMT
WHERE  STATUS IN ('B','F','G','J')
ORDER BY PLNAME, STMTNUM, SEQNO;

SELECT COLLID, NAME, VERSION,
       CASE WHEN STMTNOI <> 0
         THEN STMTNOI
         ELSE STMTNO
       END AS STMTNUM,
       SEQNO, STMT
FROM   SYSIBM.SYSPACKSTMT
WHERE  STATUS IN ('B','F','G','J')
ORDER BY COLLID, NAME, VERSION, STMTNUM, SEQNO;
```

If you specify the bind option VALIDATE(RUN), and a statement in the plan or package is not bound successfully, that statement is incrementally bound at run time. If you also specify the bind option REOPT(ALWAYS), DB2 reoptimizes the access path during the incremental bind.

Example: To determine which plans and packages have statements that will be incrementally bound, execute the following SELECT statements:

```
SELECT DISTINCT NAME
FROM   SYSIBM.SYSSTMT
WHERE  STATUS = 'F' OR STATUS = 'H';

SELECT DISTINCT COLLID, NAME, VERSION
FROM   SYSIBM.SYSPACKSTMT
WHERE  STATUS = 'F' OR STATUS = 'H';
```

The REOPT(ONCE) bind option

You can use the REOPT(ONCE) bind option to determine the access path for an SQL statement at run time. The REOPT(ONCE) bind option determines the access path for an SQL statement only once at run time and works only with dynamic SQL statements. The REOPT(ONCE) bind option allows DB2 to store the access path for dynamic SQL statements in the dynamic statement cache.

Consider using the REOPT(ONCE) bind option in the following circumstances:

- The SQL statement is a dynamic SQL statement.
- The SQL statement does not perform well with the access path that is chosen at bind time.
- The SQL statement is relatively simple and takes a relatively short time to execute. For simple SQL statements, reoptimizing the access path each time that the statement runs can degrade performance more than using the access path from the first run for each subsequent run.
- The same SQL statement is repeated many times in a loop, or is run by many threads. Because of the dynamic statement cache, the access path that DB2 chooses for the first set of input variables will perform well for subsequent executions of the same SQL statement, even if the input variable values are different each time.

To use the REOPT(ONCE) bind option most efficiently, first determine which dynamic SQL statements in your applications perform poorly with the REOPT(NONE) bind option and the REOPT(ALWAYS) bind option. Separate the code containing those statements into units that you bind into packages with the REOPT(ONCE) option. Bind the rest of the code into packages using the REOPT(NONE) bind option or the REOPT(ALWAYS) bind option, as appropriate. Then bind the plan with the REOPT(NONE) bind option. A dynamic statement in a package that is bound with REOPT(ONCE) is a candidate for reoptimization the first time that the statement is run.

Example: To determine which queries in plans and packages that are bound with the REOPT(ONCE) bind option will be reoptimized at run time, execute the following SELECT statements:

```
SELECT PLNAME,
       CASE WHEN STMTNOI <> 0
         THEN STMTNOI
         ELSE STMTNO
       END AS STMTNUM,
       SEQNO, TEXT
FROM   SYSIBM.SYSSTMT
WHERE  STATUS IN ('J')
ORDER BY PLNAME, STMTNUM, SEQNO;

SELECT COLLID, NAME, VERSION,
       CASE WHEN STMTNOI <> 0
         THEN STMTNOI
         ELSE STMTNO
       END AS STMTNUM,
       SEQNO, STMT
FROM   SYSIBM.SYSPACKSTMT
WHERE  STATUS IN ('J')
ORDER BY COLLID, NAME, VERSION, STMTNUM, SEQNO;
```

If you specify the bind option VALIDATE(RUN), and a statement in the plan or package is not bound successfully, that statement is incrementally bound at run time.

Example: To determine which plans and packages have statements that will be incrementally bound, execute the following SELECT statements:

```
SELECT DISTINCT NAME
FROM   SYSIBM.SYSSTMT
WHERE  STATUS = 'F' OR STATUS = 'H';

SELECT DISTINCT COLLID, NAME, VERSION
FROM   SYSIBM.SYSPACKSTMT
WHERE  STATUS = 'F' OR STATUS = 'H';
```

The REOPT(NONE) bind option

You should use the REOPT(NONE) bind option when an SQL statement with variable values performs well with the access path that is chosen at bind time. Keep in mind that an SQL statement that performs well with the REOPT(NONE) bind option might perform even better with the bind options that change the access path at run time.

Rewriting queries to influence access path selection

Important

This section describes tactics for rewriting queries to influence how DB2 selects access paths. The access path selection "tricks" that are described in the section might cause significant performance degradation if they are not carefully implemented and monitored.

Example: The selection method might change in a later release of DB2, causing your changes to degrade performance.

Before and after you make any permanent changes, take performance measurements. When you migrate to a new release, evaluate the performance again. Be prepared to back out any changes that have degraded performance.

The examples that follow identify potential performance problems and offer suggestions for tuning the queries. However, before you rewrite any query, you should consider whether the REOPT(ALWAYS) or REOPT(ONCE) bind options can solve your access path problems. See "Changing the access path at run time" on page 760 for more information about REOPT(ALWAYS) and REOPT(ONCE).

Example 1: An equal predicate

An equal predicate has a default filter factor of 1/COLCARD. The actual filter factor might be quite different.

Query:

```
SELECT * FROM DSN8810.EMP
WHERE SEX = :HV1;
```

Assumptions: Because the column SEX has only two different values, 'M' and 'F', the value COLCARD for SEX is 2. If the numbers of male and female employees are not equal, the actual filter factor of 1/2 is larger or smaller than the default, depending on whether :HV1 is set to 'M' or 'F'.

Recommendation: One of these two actions can improve the access path:

- Bind the package or plan that contains the query with the REOPT(ALWAYS) bind option. This action causes DB2 to reoptimize the query at run time, using the input values you provide. You might also consider binding the package or plan with the REOPT(ONCE) bind option.
- Write predicates to influence the DB2 selection of an access path, based on your knowledge of actual filter factors. For example, you can break the query into three different queries, two of which use constants. DB2 can then determine the exact filter factor for most cases when it binds the plan.

```
SELECT (HV1);
  WHEN ('M')
    DO;
      EXEC SQL SELECT * FROM DSN8810.EMP
        WHERE SEX = 'M';
    END;
  WHEN ('F')
    DO;
      EXEC SQL SELECT * FROM DSN8810.EMP
        WHERE SEX = 'F';
    END;
  OTHERWISE
```

```

DO:
  EXEC SQL SELECT * FROM DSN8810.EMP
    WHERE SEX = :HV1;
END;
END;

```

Example 2: Known ranges

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```

SELECT * FROM T1
  WHERE C1 BETWEEN :HV1 AND :HV2
    AND C2 BETWEEN :HV3 AND :HV4;

```

Assumptions: You know that:

- The application always provides a narrow range on C1 and a wide range on C2.
- The desired access path is through index T1X1.

Recommendation: If DB2 does not choose T1X1, rewrite the query as follows, so that DB2 does not choose index T1X2 on C2:

```

SELECT * FROM T1
  WHERE C1 BETWEEN :HV1 AND :HV2
    AND (C2 BETWEEN :HV3 AND :HV4 OR 0=1);

```

Example 3: Variable ranges

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```

SELECT * FROM T1
  WHERE C1 BETWEEN :HV1 AND :HV2
    AND C2 BETWEEN :HV3 AND :HV4;

```

Assumptions: You know that the application provides both narrow and wide ranges on C1 and C2. Hence, default filter factors do not allow DB2 to choose the best access path in all cases. For example, a small range on C1 favors index T1X1 on C1, a small range on C2 favors index T1X2 on C2, and wide ranges on both C1 and C2 favor a table space scan.

Recommendation: If DB2 does not choose the best access path, try either of the following changes to your application:

- Use a dynamic SQL statement and embed the ranges of C1 and C2 in the statement. With access to the actual range values, DB2 can estimate the actual filter factors for the query. Preparing the statement each time it is executed requires an extra step, but it can be worthwhile if the query accesses a large amount of data.
- Include some simple logic to check the ranges of C1 and C2, and then execute one of these static SQL statements, based on the ranges of C1 and C2:

```

SELECT * FROM T1 WHERE C1 BETWEEN :HV1 AND :HV2
    AND (C2 BETWEEN :HV3 AND :HV4 OR 0=1);

```

```

SELECT * FROM T1 WHERE C2 BETWEEN :HV3 AND :HV4
    AND (C1 BETWEEN :HV1 AND :HV2 OR 0=1);

```

```

SELECT * FROM T1 WHERE (C1 BETWEEN :HV1 AND :HV2 OR 0=1)
    AND (C2 BETWEEN :HV3 AND :HV4 OR 0=1);

```

Example 4: ORDER BY

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```
SELECT * FROM T1
WHERE C1 BETWEEN :HV1 AND :HV2
ORDER BY C2;
```

In this example, DB2 could choose one of the following actions:

- Scan index T1X1 and then sort the results by column C2
- Scan the table space in which T1 resides and then sort the results by column C2
- Scan index T1X2 and then apply the predicate to each row of data, thereby avoiding the sort

Which choice is best depends on the following factors:

- The number of rows that satisfy the range predicate
- The cluster ratio of the indexes

If the actual number of rows that satisfy the range predicate is significantly different from the estimate, DB2 might not choose the best access path.

Assumptions: You disagree with the DB2 choice.

Recommendation: In your application, use a dynamic SQL statement and embed the range of C1 in the statement. That allows DB2 to use the actual filter factor rather than the default, but requires extra processing for the PREPARE statement.

Example 5: A join operation

Tables A, B, and C each have indexes on columns C1, C2, C3, and C4.

Assumptions: The actual filter factors on table A are much larger than the default factors. Hence, DB2 underestimates the number of rows selected from table A and wrongly chooses that as the first table in the join.

Recommendations: You can:

- Reduce the estimated size of Table A by adding predicates
- Disfavor any index on the join column by making the join predicate on table A nonindexable

Example: The following query illustrates the second of those choices.

```
SELECT * FROM T1 A, T1 B, T1 C
WHERE (A.C1 = B.C1 OR 0=1)
AND A.C2 = C.C2
AND A.C2 BETWEEN :HV1 AND :HV2
AND A.C3 BETWEEN :HV3 AND :HV4
AND A.C4 < :HV5
AND B.C2 BETWEEN :HV6 AND :HV7
AND B.C3 < :HV8
AND C.C2 < :HV9;
```

The result of making the join predicate between A and B a nonindexable predicate (which cannot be used in single index access) disfavors the use of the index on column C1. This, in turn, might lead DB2 to access table A or B first. Or, it might lead DB2 to change the access type of table A or B, thereby influencing the join sequence of the other tables.

Writing efficient subqueries

Definitions: A *subquery* is a SELECT statement within the WHERE or HAVING clause of another SQL statement.

Decision needed: You can often write two or more SQL statements that achieve identical results, particularly if you use subqueries. The statements have different access paths, however, and probably perform differently.

Topic overview: The topics that follow describe different methods to achieve the results intended by a subquery and tell what DB2 does for each method. The information should help you estimate what method performs best for your query.

The first two methods use different types of subqueries:

- “Correlated subqueries”
- “Noncorrelated subqueries” on page 767

A subquery can sometimes be transformed into a join operation. Sometimes DB2 does that to improve the access path, and sometimes you can get better results by doing it yourself. The third method is:

- “Conditions for DB2 to transform a subquery into a join” on page 768

Finally, for a comparison of the three methods as applied to a single task, see:

- “Subquery tuning” on page 770

Correlated subqueries

Definition: A *correlated* subquery refers to at least one column of the outer query.

Any predicate that contains a correlated subquery is a stage 2 predicate unless it is transformed to a join.

Example: In the following query, the correlation name, X, illustrates the subquery’s reference to the outer query block.

```
SELECT * FROM DSN8810.EMP X
WHERE  JOB = 'DESIGNER'
      AND EXISTS (SELECT 1
                  FROM   DSN8810.PROJ
                  WHERE  DEPTNO = X.WORKDEPT
                  AND   MAJPROJ = 'MA2100');
```

What DB2 does: A correlated subquery is evaluated for each qualified row of the outer query that is referred to. In executing the example, DB2:

1. Reads a row from table EMP where JOB='DESIGNER'.
2. Searches for the value of WORKDEPT from that row, in a table stored in memory.

The in-memory table saves executions of the subquery. If the subquery has already been executed with the value of WORKDEPT, the result of the subquery is in the table and DB2 does not execute it again for the current row. Instead, DB2 can skip to step 5.

3. Executes the subquery, if the value of WORKDEPT is not in memory. That requires searching the PROJ table to check whether there is any project, where MAJPROJ is 'MA2100', for which the current WORKDEPT is responsible.
4. Stores the value of WORKDEPT and the result of the subquery in memory.
5. Returns the values of the current row of EMP to the application.

DB2 repeats this whole process for each qualified row of the EMP table.

Notes on the in-memory table: The in-memory table is applicable if the operator of the predicate that contains the subquery is one of the following operators:

<, <=, >, >=, =, <>, EXISTS, NOT EXISTS

The table is not used, however, if:

- There are more than 16 correlated columns in the subquery
- The sum of the lengths of the correlated columns is more than 256 bytes
- There is a unique index on a subset of the correlated columns of a table from the outer query

The in-memory table is a wrap-around table and does not guarantee saving the results of all possible duplicated executions of the subquery.

Noncorrelated subqueries

Definition: A *noncorrelated* subquery makes no reference to outer queries.

Example:

```
SELECT * FROM DSN8810.EMP
WHERE JOB = 'DESIGNER'
AND WORKDEPT IN (SELECT DEPTNO
                  FROM DSN8810.PROJ
                  WHERE MAJPROJ = 'MA2100');
```

What DB2 does: A noncorrelated subquery is executed once when the cursor is opened for the query. What DB2 does to process it depends on whether it returns a single value or more than one value. The query in the preceding example can return more than one value.

Single-value subqueries

When the subquery is contained in a predicate with a simple operator, the subquery is required to return 1 or 0 rows. The simple operator can be one of the following operators:

<, <=, >, >=, =, <>, NOT <, NOT <=, NOT >, NOT >=

The following noncorrelated subquery returns a single value:

```
SELECT *
FROM DSN8810.EMP
WHERE JOB = 'DESIGNER'
AND WORKDEPT <= (SELECT MAX(DEPTNO)
                  FROM DSN8810.PROJ);
```

What DB2 does: When the cursor is opened, the subquery executes. If it returns more than one row, DB2 issues an error. The predicate that contains the subquery is treated like a simple predicate with a constant specified, for example, WORKDEPT <= 'value'.

Stage 1 and stage 2 processing: The rules for determining whether a predicate with a noncorrelated subquery that returns a single value is stage 1 or stage 2 are generally the same as for the same predicate with a single variable.

Multiple-value subqueries

A subquery can return more than one value if the operator is one of the following:
op ANY, *op* ALL, *op* SOME, IN, EXISTS

where *op* is any of the operators >, >=, <, <=, NOT <, NOT <=, NOT >, NOT >=.

What DB2 does: If possible, DB2 reduces a subquery that returns more than one row to one that returns only a single row. That occurs when there is a range comparison along with ANY, ALL, or SOME. The following query is an example:

```
SELECT * FROM DSN8810.EMP
WHERE JOB = 'DESIGNER'
AND WORKDEPT <= ANY (SELECT DEPTNO
                      FROM DSN8810.PROJ
                      WHERE MAJPROJ = 'MA2100');
```

DB2 calculates the maximum value for DEPTNO from table DSN8810.PROJ and removes the ANY keyword from the query. After this transformation, the subquery is treated like a single-value subquery.

That transformation can be made with a *maximum value* if the range operator is:

- > or >= with the quantifier ALL
- < or <= with the quantifier ANY or SOME

The transformation can be made with a *minimum value* if the range operator is:

- < or <= with the quantifier ALL
- > or >= with the quantifier ANY or SOME

The resulting predicate is determined to be stage 1 or stage 2 by the same rules as for the same predicate with a single-valued subquery.

When a subquery is sorted: A noncorrelated subquery is sorted when the comparison operator is IN, NOT IN, = ANY, <> ANY, = ALL, or <> ALL. The sort enhances the predicate evaluation, reducing the amount of scanning on the subquery result. When the value of the subquery becomes smaller or equal to the expression on the left side, the scanning can be stopped and the predicate can be determined to be true or false.

When the subquery result is a character data type and the left side of the predicate is a datetime data type, then the result is placed in a work file without sorting. For some noncorrelated subqueries that use IN, NOT IN, = ANY, <> ANY, = ALL, or <> ALL comparison operators, DB2 can more accurately pinpoint an entry point into the work file, thus further reducing the amount of scanning that is done.

Results from EXPLAIN: For information about the result in a plan table for a subquery that is sorted, see “When are aggregate functions evaluated? (COLUMN_FN_EVAL)” on page 808.

Conditions for DB2 to transform a subquery into a join

For a SELECT, UPDATE, or DELETE statement, DB2 can sometimes transform a subquery into a join between the result table of a subquery and the result table of an outer query.

For a SELECT statement, DB2 does the transformation if the following conditions are true:

- The transformation does not introduce redundancy.
- The subquery appears in a WHERE clause.

- The subquery does not contain GROUP BY, HAVING, or aggregate functions.
- The subquery has only one table in the FROM clause.
- For a correlated subquery, the comparison operator of the predicate containing the subquery is IN, = ANY, or = SOME.
- For a noncorrelated subquery, the comparison operator of the predicate containing the subquery is IN, EXISTS, = ANY, or = SOME.
- For a noncorrelated subquery, the subquery select list has only one column, guaranteed by a unique index to have unique values.
- For a noncorrelated subquery, the left side of the predicate is a single column with the same data type and length as the subquery's column. (For a correlated subquery, the left side can be any expression.)

For an UPDATE or DELETE statement, or a SELECT statement that does not meet the previous conditions for transformation, DB2 does the transformation of a correlated subquery into a join if the following conditions are true:

- The transformation does not introduce redundancy.
- The subquery is correlated to its immediate outer query.
- The FROM clause of the subquery contains only one table, and the outer query (for SELECT), UPDATE, or DELETE references only one table.
- If the outer predicate is a quantified predicate with an operator of =ANY or an IN predicate, the following conditions are true:
 - The left side of the outer predicate is a single column.
 - The right side of the outer predicate is a subquery that references a single column.
 - The two columns have the same data type and length.
- The subquery does not contain the GROUP BY or DISTINCT clauses.
- The subquery does not contain aggregate functions.
- The SELECT clause of the subquery does not contain a user-defined function with an external action or a user-defined function that modifies data.
- The subquery predicate is a Boolean term predicate.
- The predicates in the subquery that provide correlation are stage 1 predicates.
- The subquery does not contain nested subqueries.
- The subquery does not contain a self-referencing UPDATE or DELETE.
- For a SELECT statement, the query does not contain the FOR UPDATE OF clause.
- For an UPDATE or DELETE statement, the statement is a searched UPDATE or DELETE.
- For a SELECT statement, parallelism is not enabled.

For a statement with multiple subqueries, DB2 does the transformation only on the last subquery in the statement that qualifies for transformation.

Example: The following subquery can be transformed into a join because it meets the first set of conditions for transformation:

```
SELECT * FROM EMP
WHERE DEPTNO IN
  (SELECT DEPTNO FROM DEPT
   WHERE LOCATION IN ('SAN JOSE', 'SAN FRANCISCO')
   AND DIVISION = 'MARKETING');
```

If there is a department in the marketing division which has branches in both San Jose and San Francisco, the result of the SQL statement is not the same as if a join were done. The join makes each employee in this department appear twice because it matches once for the department of location San Jose and again of location San Francisco, although it is the same department. Therefore, it is clear that to transform a subquery into a join, the uniqueness of the subquery select list must be guaranteed. For this example, a unique index on any of the following sets of columns would guarantee uniqueness:

- (DEPTNO)
- (DIVISION, DEPTNO)
- (DEPTNO, DIVISION).

The resultant query is:

```
SELECT EMP.* FROM EMP, DEPT
      WHERE EMP.DEPTNO = DEPT.DEPTNO AND
            DEPT.LOCATION IN ('SAN JOSE', 'SAN FRANCISCO') AND
            DEPT.DIVISION = 'MARKETING';
```

Example: The following subquery can be transformed into a join because it meets the second set of conditions for transformation:

```
UPDATE T1 SET T1.C1 = 1
      WHERE T1.C1 = ANY
            (SELECT T2.C1 FROM T2
             WHERE T2.C2 = T1.C2);
```

Results from EXPLAIN: For information about the result in a plan table for a subquery that is transformed into a join operation, see “Is a subquery transformed into a join?” on page 808.

Subquery tuning

The following three queries all retrieve the same rows. All three retrieve data about all designers in departments that are responsible for projects that are part of major project MA2100. These three queries show that there are several ways to retrieve a desired result.

Query A: A join of two tables

```
SELECT DSN8810.EMP.* FROM DSN8810.EMP, DSN8810.PROJ
      WHERE JOB = 'DESIGNER'
            AND WORKDEPT = DEPTNO
            AND MAJPROJ = 'MA2100';
```

Query B: A correlated subquery

```
SELECT * FROM DSN8810.EMP X
      WHERE JOB = 'DESIGNER'
            AND EXISTS (SELECT 1 FROM DSN8810.PROJ
                       WHERE DEPTNO = X.WORKDEPT
                             AND MAJPROJ = 'MA2100');
```

Query C: A noncorrelated subquery

```
SELECT * FROM DSN8810.EMP
      WHERE JOB = 'DESIGNER'
            AND WORKDEPT IN (SELECT DEPTNO FROM DSN8810.PROJ
                             WHERE MAJPROJ = 'MA2100');
```

If you need columns from both tables EMP and PROJ in the output, you must use a join.

PROJ might contain duplicate values of DEPTNO in the subquery, so that an equivalent join cannot be written.

In general, query A might be the one that performs best. However, if there is no index on DEPTNO in table PROJ, then query C might perform best. The IN-subquery predicate in query C is indexable. Therefore, if an index on WORKDEPT exists, DB2 might do IN-list access on table EMP. If you decide that a join cannot be used and there is an available index on DEPTNO in table PROJ, then query B might perform best.

When looking at a problem subquery, see if the query can be rewritten into another format or see if there is an index that you can create to help improve the performance of the subquery.

Knowing the sequence of evaluation is important, for the different subquery predicates and for all other predicates in the query. If the subquery predicate is costly, perhaps another predicate could be evaluated before that predicate so that the rows would be rejected before even evaluating the problem subquery predicate.

Using scrollable cursors efficiently

The following recommendations help you get the best performance from your scrollable cursors:

- Determine when scrollable cursors work best for you.

Scrollable cursors are a valuable tool for writing applications such as screen-based applications, in which the result table is small and you often move back and forth through the data. However, scrollable cursors require more DB2 processing than non-scrollable cursors. If your applications require large result tables or you only need to move sequentially forward through the data, use non-scrollable cursors.

- Declare scrollable cursors as SENSITIVE only if you need to see the latest data.

If you do not need to see updates that are made by other cursors or application processes, using a cursor that you declare as INSENSITIVE requires less processing by DB2.

If you need to see only some of the latest updates, and you do not need to see the results of insert operations, declare scrollable cursors as SENSITIVE STATIC. See Chapter 5 of *DB2 SQL Reference* for information about which updates you can see with a scrollable cursor that is declared as SENSITIVE STATIC.

If you need to see all of the latest updates and inserts, declare scrollable cursors as SENSITIVE DYNAMIC.

- To ensure maximum concurrency when you use a scrollable cursor for positioned update and delete operations, specify ISOLATION(CS) and CURRENTDATA(NO) when you bind packages and plans that contain updatable scrollable cursors. See Chapter 18, “Planning for concurrency,” on page 393 for more details.
- Use the FETCH FIRST *n* ROWS ONLY clause with scrollable cursors when it is appropriate.

In a distributed environment, when you need to retrieve a limited number of rows, FETCH FIRST *n* ROWS ONLY can improve your performance for distributed queries that use DRDA access by eliminating unneeded network traffic. See “Limiting the number of rows returned to DRDA clients” on page 464 for more information.

In a local environment, if you need to scroll through a limited subset of rows in a table, you can use `FETCH FIRST n ROWS ONLY` to make the result table smaller.

- In a distributed environment, if you do not need to use your scrollable cursors to modify data, do your cursor processing in a stored procedure.

Using stored procedures can decrease the amount of network traffic that your application requires.

- In a TEMP database, create table spaces that are large enough for processing your scrollable cursors.

DB2 uses declared temporary tables for processing the following types of scrollable cursors:

- SENSITIVE STATIC SCROLL
- INSENSITIVE SCROLL
- ASENSITIVE SCROLL, if the cursor sensitivity is INSENSITIVE. A cursor that meets the criteria for a read-only cursor has an effective sensitivity of INSENSITIVE.

See the `DECLARE CURSOR` statement in *DB2 SQL Reference* for more information about cursor sensitivity. See *DB2 Installation Guide* for more information about calculating the appropriate size for declared temporary tables for cursors.

- Remember to commit changes often for the following reasons:
 - You frequently need to leave scrollable cursors open longer than non-scrollable cursors.
 - There is an increased chance of deadlocks with scrollable cursors because scrollable cursors allow rows to be accessed and updated in any order. Frequent commits can decrease the chances of deadlocks.

To prevent cursors from closing after commit operations, declare your scrollable cursors `WITH HOLD`.

- While sensitive static sensitive scrollable cursors are open against a table, DB2 will disallow reuse of space in that table space to prevent the scrollable cursor from fetching newly inserted rows that were not in the original result set. Although this is normal, it can result in a seemingly false out-of-space indication. The problem can be more noticeable in a data sharing environment with transactions that access LOBs. Consider the following preventive measures:
 - Check applications such that they commit frequently
 - Close sensitive scrollable cursors when no longer needed
 - Remove `WITH HOLD` parm for the sensitive scrollable cursor, if possible
 - Isolate LOB table spaces in a dedicated bufferpool in the data sharing environment

Writing efficient queries on tables with data-partitioned secondary indexes

The number of partitions that DB2 accesses to evaluate a query predicate can affect the performance of the query. A query that provides data retrieval through a data-partitioned secondary index (DPSI) might access some or all partitions of the DPSI. For a query that is based only on a DPSI key value or range, DB2 must examine all partitions. If the query also has predicates on the leading columns of the partitioning key, DB2 does not need to examine all partitions. The removal from consideration of inapplicable partitions is known as *page range screening* or *limited partition scan*. A limited partition scan can be determined at bind time or at run time. For example, a limited partition scan can be determined at bind time for

a predicate in which a column is compared to a constant. A limited partition scan occurs at run time if the column is compared to a host variable, parameter marker, or special register.

The following example demonstrates how you can use a partitioning index to enable a limited partition scan on a set of partitions that DB2 needs to examine to satisfy a query predicate.

Suppose that you create table Q1, with partitioning index DATE_IX and DPSI STATE_IX:

```
CREATE TABLESPACE TS1 NUMPARTS 3;

CREATE TABLE Q1 (DATE DATE,
  CUSTNO CHAR(5),
  STATE CHAR(2),
  PURCH_AMT DECIMAL(9,2))
  IN TS1
  PARTITION BY (DATE)
  (PARTITION 1 ENDING AT ('2002-1-31'),
   PARTITION 2 ENDING AT ('2002-2-28'),
   PARTITION 3 ENDING AT ('2002-3-31'));

CREATE INDEX DATE_IX ON Q1 (DATE) PARTITIONED CLUSTER;

CREATE INDEX STATE_IX ON Q1 (STATE) PARTITIONED;
```

Now suppose that you want to execute the following query against table Q1:

```
SELECT CUSTNO, PURCH_AMT
FROM Q1
WHERE STATE = 'CA';
```

Because the predicate is based only on values of a DPSI key (STATE), DB2 must examine all partitions to find the matching rows.

Now suppose that you modify the query in the following way:

```
SELECT CUSTNO, PURCH_AMT
FROM Q1
WHERE DATE BETWEEN '2002-01-01' AND '2002-01-31' AND
STATE = 'CA';
```

Because the predicate is now based on values of a partitioning index key (DATE) and on values of a DPSI key (STATE), DB2 can eliminate the scanning of data partitions 2 and 3, which do not satisfy the query for the partitioning key. This can be determined at bind time because the columns of the predicate are compared to constants.

Now suppose that you use host variables instead of constants in the same query:

```
SELECT CUSTNO, PURCH_AMT
FROM Q1
WHERE DATE BETWEEN :hv1 AND :hv2 AND
STATE = :hv3;
```

DB2 can use the predicate on the partitioning column to eliminate the scanning of unneeded partitions at run time.

Writing queries to take advantage of limited partition scan is especially useful when a correlation exists between columns that are in a partitioning index and columns that are in a DPSI.

For example, suppose that you create table Q2, with partitioning index DATE_IX and DPSI ORDERNO_IX:

```
CREATE TABLESPACE TS2 Numparts 3;

CREATE TABLE Q2 (DATE DATE,
ORDERNO CHAR(8),
STATE CHAR(2),
PURCH_AMT DECIMAL(9,2))
IN TS2
PARTITION BY (DATE)
(PARTITION 1 ENDING AT ('2000-12-31'),
PARTITION 2 ENDING AT ('2001-12-31'),
PARTITION 3 ENDING AT ('2002-12-31'));

CREATE INDEX DATE_IX ON Q2 (DATE) PARTITIONED CLUSTER;

CREATE INDEX ORDERNO_IX ON Q2 (ORDERNO) PARTITIONED;
```

Also suppose that the first 4 bytes of each ORDERNO column value represent the four-digit year in which the order is placed. This means that the DATE column and the ORDERNO column are correlated.

To take advantage of limited partition scan, when you write a query that has the ORDERNO column in the predicate, also include the DATE column in the predicate. The partitioning index on DATE lets DB2 eliminate the scanning of partitions that are not needed to satisfy the query. For example:

```
SELECT ORDERNO, PURCH_AMT
FROM Q2
WHERE ORDERNO BETWEEN '2002AAAA' AND '2002ZZZZ' AND
DATE BETWEEN '2002-01-01' AND '2002-12-31';
```

Special techniques to influence access path selection

Important

This section describes tactics for rewriting queries and modifying catalog statistics to influence how DB2 selects access paths. The access path selection "tricks" that are described in the section might cause significant performance degradation if they are not carefully implemented and monitored.

Example: The selection method might change in a later release of DB2, causing your changes to degrade performance.

Save the old catalog statistics or SQL before you consider making any changes to control the choice of access path. Before and after you make any changes, take performance measurements. When you migrate to a new release, evaluate the performance again. Be prepared to back out any changes that have degraded performance.

This section contains the following information about determining and changing access paths:

- Obtaining information about access paths
- "Minimizing overhead for retrieving few rows: OPTIMIZE FOR *n* ROWS" on page 776
- "Fetching a limited number of rows: FETCH FIRST *n* ROWS ONLY" on page 775

- “Using the CARDINALITY clause to improve the performance of queries with user-defined table function references” on page 779
- “Reducing the number of matching columns” on page 780
- “Rearranging the order of tables in a FROM clause” on page 784
- “Updating catalog statistics” on page 785
- “Using a subsystem parameter” on page 786

Obtaining information about access paths

You can obtain information about DB2 access paths by using the following methods:

- Use Visual Explain.

The DB2 Visual Explain tool, which is invoked from a workstation client, can be used to display and analyze information on access paths chosen by DB2. Visual Explain displays the access path information in both graphic and text formats. The tool provides you with an easy-to-use interface to the PLAN_TABLE output and allows you to invoke EXPLAIN for dynamic SQL statements. You can also access the catalog statistics for certain referenced objects of an access path. In addition, the tool allows you to archive EXPLAIN output from previous SQL statements to analyze changes in your SQL environment. See *DB2 Visual Explain online help* for more information.

- Run OMEGAMON DB2 Performance Monitor accounting reports.

Another way to track performance is with the OMEGAMON DB2 Performance Monitor accounting reports. The accounting report, short layout, ordered by PLANNAME, lists the primary performance figures. Check the plans that contain SQL statements whose access paths you tried to influence. If the elapsed time, TCB time, or number of getpage requests increases sharply without a corresponding increase in the SQL activity, then there could be a problem. You can use OMEGAMON Online Monitor to track events after your changes have been implemented, providing immediate feedback on the effects of your changes.

- Specify the bind option EXPLAIN.

You can also use the EXPLAIN option when you bind or rebind a plan or package. Compare the new plan or package for the statement to the old one. If the new one has a table space scan or a nonmatching index space scan, but the old one did not, the problem is probably the statement. Investigate any changes in access path in the new plan or package; they could represent performance improvements or degradations. If neither the accounting report ordered by PLANNAME or PACKAGE nor the EXPLAIN statement suggest corrective action, use the OMEGAMON SQL activity reports for additional information. For more information on using EXPLAIN, see “Obtaining PLAN_TABLE information from EXPLAIN” on page 790.

Fetching a limited number of rows: FETCH FIRST *n* ROWS ONLY

In some applications, you execute queries that can return a large number of rows, but you need only a small subset of those rows. Retrieving the entire result table from the query can be inefficient. You can specify the FETCH FIRST *n* ROWS ONLY clause in a SELECT statement to limit the number of rows in the result table of a query to *n* rows. In addition, for a distributed query that uses DRDA access, FETCH FIRST *n* ROWS ONLY, DB2 prefetches only *n* rows.

Example: Suppose that you write an application that requires information on only the 20 employees with the highest salaries. To return only the rows of the employee table for those 20 employees, you can write a query like this:

```
SELECT LASTNAME, FIRSTNAME, EMPNO, SALARY
FROM EMP
ORDER BY SALARY DESC
FETCH FIRST 20 ROWS ONLY;
```

Interaction between OPTIMIZE FOR *n* ROWS and FETCH FIRST *n* ROWS ONLY: In general, if you specify FETCH FIRST *n* ROWS ONLY but not OPTIMIZE FOR *n* ROWS in a SELECT statement, DB2 optimizes the query as if you had specified OPTIMIZE FOR *n* ROWS.

When both the FETCH FIRST *n* ROWS ONLY clause and the OPTIMIZE FOR *n* ROWS clause are specified, the value for the OPTIMIZE FOR *n* ROWS clause is used for access path selection.

Example: Suppose that you submit the following SELECT statement:

```
SELECT * FROM EMP
FETCH FIRST 5 ROWS ONLY
OPTIMIZE FOR 20 ROWS;
```

The OPTIMIZE FOR value of 20 rows is used for access path selection.

Minimizing overhead for retrieving few rows: OPTIMIZE FOR *n* ROWS

When an application executes a SELECT statement, DB2 assumes that the application will retrieve all the qualifying rows. This assumption is most appropriate for batch environments. However, for interactive SQL applications, such as SPUFI, it is common for a query to define a very large potential result set but retrieve only the first few rows. The access path that DB2 chooses might not be optimal for those interactive applications.

This section discusses the use of OPTIMIZE FOR *n* ROWS to affect the performance of interactive SQL applications. Unless otherwise noted, this information pertains to local applications. For more information on using OPTIMIZE FOR *n* ROWS in distributed applications, see “Limiting the number of DRDA network transmissions” on page 461.

What OPTIMIZE FOR *n* ROWS does: The OPTIMIZE FOR *n* ROWS clause lets an application declare its intent to do either of these things:

- Retrieve only a subset of the result set
- Give priority to the retrieval of the first few rows

DB2 uses the OPTIMIZE FOR *n* ROWS clause to choose access paths that minimize the response time for retrieving the first few rows. For distributed queries, the value of *n* determines the number of rows that DB2 sends to the client on each DRDA network transmission. See “Limiting the number of DRDA network transmissions” on page 461 for more information on using OPTIMIZE FOR *n* ROWS in the distributed environment.

Use OPTIMIZE FOR 1 ROW to avoid sorts: You can influence the access path most by using OPTIMIZE FOR 1 ROW. OPTIMIZE FOR 1 ROW tells DB2 to select an access path that returns the first qualifying row quickly. This means that

whenever possible, DB2 avoids any access path that involves a sort. If you specify a value for *n* that is anything but 1, DB2 chooses an access path based on cost, and you won't necessarily avoid sorts.

How to specify OPTIMIZE FOR *n* ROWS for a CLI application: For a Call Level Interface (CLI) application, you can specify that DB2 uses OPTIMIZE FOR *n* ROWS for all queries. To do that, specify the keyword OPTIMIZEFORNROWS in the initialization file. For more information, see Chapter 3 of *DB2 ODBC Guide and Reference*.

How many rows you can retrieve with OPTIMIZE FOR *n* ROWS: The OPTIMIZE FOR *n* ROWS clause does not prevent you from retrieving all the qualifying rows. However, if you use OPTIMIZE FOR *n* ROWS, the total elapsed time to retrieve all the qualifying rows might be significantly greater than if DB2 had optimized for the entire result set.

When OPTIMIZE FOR *n* ROWS is effective: OPTIMIZE FOR *n* ROWS is effective only on queries that can be performed incrementally. If the query causes DB2 to gather the whole result set before returning the first row, DB2 ignores the OPTIMIZE FOR *n* ROWS clause, as in the following situations:

- The query uses SELECT DISTINCT or a set function distinct, such as COUNT(DISTINCT C1).
- Either GROUP BY or ORDER BY is used, and no index can give the necessary ordering.
- A aggregate function and no GROUP BY clause is used.
- The query uses UNION.

Example: Suppose that you query the employee table regularly to determine the employees with the highest salaries. You might use a query like this:

```
SELECT LASTNAME, FIRSTNAME, EMPNO, SALARY
FROM EMP
ORDER BY SALARY DESC;
```

An index is defined on column EMPNO, so employee records are ordered by EMPNO. If you have also defined a descending index on column SALARY, that index is likely to be very poorly clustered. To avoid many random, synchronous I/O operations, DB2 would most likely use a table space scan, then sort the rows on SALARY. This technique can cause a delay before the first qualifying rows can be returned to the application.

If you add the OPTIMIZE FOR *n* ROWS clause to the statement, DB2 will probably use the SALARY index directly because you have indicated that you expect to retrieve the salaries of only the 20 most highly paid employees.

Example: The following statement uses that strategy to avoid a costly sort operation:

```
SELECT LASTNAME, FIRSTNAME, EMPNO, SALARY
FROM EMP
ORDER BY SALARY DESC
OPTIMIZE FOR 20 ROWS;
```

Effects of using OPTIMIZE FOR *n* ROWS:

- The join method could change. Nested loop join is the most likely choice, because it has low overhead cost and appears to be more efficient if you want to retrieve only one row.

- An index that matches the ORDER BY clause is more likely to be picked. This is because no sort would be needed for the ORDER BY.
- List prefetch is less likely to be picked.
- Sequential prefetch is less likely to be requested by DB2 because it infers that you only want to see a small number of rows.
- In a join query, the table with the columns in the ORDER BY clause is likely to be picked as the outer table if there is an index on that outer table that gives the ordering needed for the ORDER BY clause.

Recommendation: For a local query, specify OPTIMIZE FOR *n* ROWS only in applications that frequently fetch only a small percentage of the total rows in a query result set. For example, an application might read only enough rows to fill the end user's terminal screen. In cases like this, the application might read the remaining part of the query result set only rarely. For an application like this, OPTIMIZE FOR *n* ROWS can result in better performance by causing DB2 to favor SQL access paths that deliver the first *n* rows as fast as possible.

When you specify OPTIMIZE FOR *n* ROWS for a remote query, a small value of *n* can help limit the number of rows that flow across the network on any given transmission.

You can improve the performance for receiving a large result set through a remote query by specifying a large value of *n* in OPTIMIZE FOR *n* ROWS. When you specify a large value, DB2 attempts to send the *n* rows in multiple transmissions. For better performance when retrieving a large result set, in addition to specifying OPTIMIZE FOR *n* ROWS with a large value of *n* in your query, do not execute other SQL statements until the entire result set for the query is processed. If retrieval of data for several queries overlaps, DB2 might need to buffer result set data in the DDF address space. See "Block fetching result sets" in Part 5 (Volume 2) of *DB2 Administration Guide* for more information.

For local or remote queries, to influence the access path most, specify OPTIMIZE for 1 ROW. This value does not have a detrimental effect on distributed queries.

Favoring index access

One common database design involves tables that contain groups of rows that logically belong together. Within each group, the rows should be accessed in the same sequence every time. The sequence is determined by the primary key on the table. Lock contention can occur when DB2 chooses different access paths for different applications that operate on a table with this design.

To minimize contention among applications that access tables with this design, specify the VOLATILE keyword when you create or alter the tables. A table that is defined with the VOLATILE keyword is known as a *volatile table*. When DB2 executes queries that include volatile tables, DB2 uses index access whenever possible. One exception is for DELETE statements on a VOLATILE table in a segmented table space when no WHERE clause is specified. In this case, a table space scan is used. As well as minimizing contention, using index access preserves the access sequence that the primary key provides.

Defining a table as volatile has a similar effect on a query to setting the NPGTHRSH subsystem parameter to favor matching index access for all qualified tables. (See "Using a subsystem parameter" on page 786 for information on the settings for NPGTHRSH.) However, the effect of NPGTHRSH is subsystem-wide,

and index access might not be appropriate for many queries. Defining tables as volatile lets you limit the set of queries that favor index access to queries that involve the volatile tables.

Using a subsystem parameter to control outer join processing

Subsystem parameter OJPERFEH can improve outer join processing. In particular, when the value of OJPERFEH is YES, DB2 takes the following actions, which can improve outer join processing in most cases:

- Does not merge table expressions or views if the parent query block of a table expression or view contains an outer join, and the merge would cause a column in a predicate to become an expression.
- Does not attempt to reduce work file usage for outer joins.
- Uses transitive closure for the ON predicates in outer joins.

However, these actions might not improve performance for some outer joins.

Recommendation: If the performance of queries that contain outer joins is not adequate, set OJPERFEH to NO, restart DB2, and rerun those queries.

Using the CARDINALITY clause to improve the performance of queries with user-defined table function references

The cardinality of a user-defined table function is the number of rows that are returned when the function is invoked. DB2 uses this number to estimate the cost of executing a query that invokes a user-defined table function. The cost of executing a query is one of the factors that DB2 uses when it calculates the access path. Therefore, if you give DB2 an accurate estimate of a user-defined table function's cardinality, DB2 can better calculate the best access path.

You can specify a cardinality value for a user-defined table function by using the CARDINALITY clause of the SQL CREATE FUNCTION or ALTER FUNCTION statement. However, this value applies to all invocations of the function, whereas a user-defined table function might return different numbers of rows, depending on the query in which it is referenced.

To give DB2 a better estimate of the cardinality of a user-defined table function for a particular query, you can use the CARDINALITY or CARDINALITY MULTIPLIER clause in that query. DB2 uses those clauses at bind time when it calculates the access cost of the user-defined table function. Using this clause is recommended only for programs that run on DB2 UDB for z/OS because the clause is not supported on earlier versions of DB2.

Example of using the CARDINALITY clause to specify the cardinality of a user-defined table function invocation: Suppose that when you created user-defined table function TUDF1, you set a cardinality value of 5, but in the following query, you expect TUDF1 to return 30 rows:

```
SELECT *  
FROM TABLE(TUDF1(3)) AS X;
```

Add the CARDINALITY 30 clause to tell DB2 that, for this query, TUDF1 should return 30 rows:

```
SELECT *  
FROM TABLE(TUDF1(3) CARDINALITY 30) AS X;
```

| *Example of using the CARDINALITY MULTIPLIER clause to specify the*
| *cardinality of a user-defined table function invocation:* Suppose that when you
| created user-defined table function TUDF2, you set a cardinality value of 5, but in
| the following query, you expect TUDF2 to return 30 times that many rows:

| SELECT *
| FROM TABLE(TUDF2(10)) AS X;

| Add the CARDINALITY MULTIPLIER 30 clause to tell DB2 that, for this query,
| TUDF1 should return 5*30, or 150, rows:

| SELECT *
| FROM TABLE(TUDF2(10) CARDINALITY MULTIPLIER 30) AS X;

Reducing the number of matching columns

Discourage the use of a poorer performing index by reducing the index's matching predicate on its leading column. Consider the example in Figure 227 on page 781, where the index that DB2 picks is less than optimal.

```

CREATE TABLE PART_HISTORY (
  PART_TYPE CHAR(2),      IDENTIFIES THE PART TYPE
  PART_SUFFIX CHAR(10),   IDENTIFIES THE PART
  W_NOW      INTEGER,      TELLS WHERE THE PART IS
  W_FROM     INTEGER,      TELLS WHERE THE PART CAME FROM
  DEVIATIONS INTEGER,      TELLS IF ANYTHING SPECIAL WITH THIS PART
  COMMENTS   CHAR(254),
  DESCRIPTION CHAR(254),
  DATE1      DATE,
  DATE2      DATE,
  DATE3      DATE);

CREATE UNIQUE INDEX IX1 ON PART_HISTORY
(PART_TYPE,PART_SUFFIX,W_FROM,W_NOW);
CREATE UNIQUE INDEX IX2 ON PART_HISTORY
(W_FROM,W_NOW,DATE1);

```

Table statistics		Index statistics			IX1	IX2
CARDF	100,000	FIRSTKEYCARDF	1000		50	
NPAGES	10,000	FULLKEYCARDF	100,000		100,000	
		CLUSTERRATIO	99%		99%	
		NLEAF	3000		2000	
		NLEVELS	3		3	
		column	cardinality	HIGH2KEY	LOW2KEY	
		Part_type	1000	'ZZ'	'AA'	
		w_now	50	1000	1	
		w_from	50	1000	1	

Q1:

```

SELECT * FROM PART_HISTORY -- SELECT ALL PARTS
WHERE PART_TYPE = 'BB'     P1 -- THAT ARE 'BB' TYPES
AND W_FROM = 3             P2 -- THAT WERE MADE IN CENTER 3
AND W_NOW = 3              P3 -- AND ARE STILL IN CENTER 3

```

Filter factor of these predicates.							
P1 = 1/1000 = .001							
P2 = 1/50 = .02							
P3 = 1/50 = .02							
ESTIMATED VALUES				WHAT REALLY HAPPENS			
index	matchcols	filter factor	data rows	index	matchcols	filter factor	data rows
ix2	2	.02*.02	40	ix2	2	.02*.50	1000
ix1	1	.001	100	ix1	1	.001	100

Figure 227. Reducing the number of MATCHCOLS

DB2 picks IX2 to access the data, but IX1 would be roughly 10 times quicker. The problem is that 50% of all parts from center number 3 are still in Center 3; they have not moved. Assume that there are no statistics on the correlated columns in catalog table SYSCOLDIST. Therefore, DB2 assumes that the parts from center number 3 are evenly distributed among the 50 centers.

You can get the desired access path by changing the query. To discourage the use of IX2 for this particular query, you can change the third predicate to be nonindexable.

```

SELECT * FROM PART_HISTORY
WHERE PART_TYPE = 'BB'
AND W_FROM = 3
AND (W_NOW = 3 + 0)      <-- PREDICATE IS MADE NONINDEXABLE

```

Now index I2 is not picked, because it has only one match column. The preferred index, I1, is picked. The third predicate is a nonindexable predicate, so an index is not used for the compound predicate.

You can make a predicate nonindexable in many ways. The recommended way is to add 0 to a predicate that evaluates to a numeric value or to concatenate an empty string to a predicate that evaluates to a character value.

Indexable	Nonindexable
T1.C3=T2.C4	(T1.C3=T2.C4 CONCAT '')
T1.C1=5	T1.C1=5+0

These techniques do not affect the result of the query and cause only a small amount of overhead.

The preferred technique for improving the access path when a table has correlated columns is to generate catalog statistics on the correlated columns. You can do that either by running RUNSTATS or by updating catalog table SYSCOLDIST manually.

Creating indexes for efficient star-join processing

A *star schema* is a database design that, in its simplest form, consists of a large table called a *fact table*, and two or more smaller tables, called *dimension tables*. More complex star schemas can be created by breaking one or more of the dimension tables into multiple tables.

To access the data in a star schema design, you often write SELECT statements that include join operations between the fact table and the dimension tables, but no join operations between dimension tables. These types of queries are known as *star-join queries*.

For a star-join query, DB2 uses a special join type called a *star join* if the following conditions are true:

- The tables meet the conditions that are specified in “Star join (JOIN_TYPE='S')” on page 823.
- The STARJOIN system parameter is set to ENABLE, and the number of tables in the query block is greater than or equal to the minimum number that is specified in the SJTABLES system parameter.
See “Star join (JOIN_TYPE='S')” on page 823 for detailed discussions of these system parameters.

This section gives suggestions for choosing indexes might improve star-join query performance.

Recommendations for creating indexes for star-join queries

Follow these recommendations to improve performance of star-join queries:

- Define a multi-column index on all key columns of the fact table. Key columns are fact table columns that have corresponding dimension tables.
- If you do not have information about the way that your data is used, first try a multi-column index on the fact table that is based on the correlation of the data. Put less highly correlated columns later in the index key than more highly correlated columns. See “Determining the order of columns in an index for a star schema design” on page 783 for information on deriving an index that follows this recommendation.

- As the correlation of columns in the fact table changes, reevaluate the index to determine if columns in the index should be reordered.
- Define indexes on dimension tables to improve access to those tables.
- When you have executed a number of queries and have more information about the way that the data is used, follow these recommendations:
 - Put more selective columns at the beginning of the index.
 - If a number of queries do not reference a dimension, put the column that corresponds to that dimension at the end of the index.

When a fact table has more than one multi-column index and none of those indexes contains all key columns, DB2 evaluates all of the indexes and uses the index that best exploits star join.

Determining the order of columns in an index for a star schema design

You can use the following method to determine the order of columns in a multi-column index. The description of the method uses the following terminology:

F A fact table.

D1...D_n
Dimension tables.

C1...C_n
Key columns in the fact table. C1 is joined to dimension D1, C2 is joined to dimension D2, and so on.

cardD1...cardD_n
Cardinality of columns C1...C_n in dimension tables D1...D_n.

cardC1...cardC_n
Cardinality of key columns C1...C_n in fact table F.

cardC_{ij}
Cardinality of pairs of column values from key columns C_i and C_j in fact table F.

cardC_{ijk}
Cardinality of triplets of column values from key columns C_i, C_j, and C_k in fact table F.

Density
A measure of the correlation of key columns in the fact table. The density is calculated as follows:

For a single column
 $\text{cardC}_i / \text{cardD}_i$

For pairs of columns
 $\text{cardC}_{ij} / (\text{cardD}_i * \text{cardD}_j)$

For triplets of columns
 $\text{cardC}_{ijk} / (\text{cardD}_i * \text{cardD}_j * \text{cardD}_k)$

S The current set of columns whose order in the index is not yet determined.

S-{C_m}
The current set of columns, excluding column C_m

Follow these steps to derive a fact table index for a star-join query that joins *n* columns of fact table F to *n* dimension tables D1 through D_n:

1. Define the set of columns whose index key order is to be determined as the n columns of fact table F that correspond to dimension tables. That is, $S=\{C_1,...C_n\}$ and $L=n$.
2. Calculate the density of all sets of $L-1$ columns in S .
3. Find the lowest density. Determine which column is not in the set of columns with the lowest density. That is, find column C_m in S , such that for every C_i in S , $\text{density}(S-\{C_m\}) < \text{density}(S-\{C_i\})$.
4. Make C_m the L th column of the index.
5. Remove C_m from S .
6. Decrement L by 1.
7. Repeat steps 2 through 6 $n-2$ times. The remaining column after iteration $n-2$ is the first column of the index.

Example of determining column order for a fact table index: Suppose that a star schema has three dimension tables with the following cardinalities:

```
cardD1=2000
cardD2=500
cardD3=100
```

Now suppose that the cardinalities of single columns and pairs of columns in the fact table are:

```
cardC1=2000
cardC2=433
cardC3=100
cardC12=625000
cardC13=196000
cardC23=994
```

Determine the best multi-column index for this star schema.

Step 1: Calculate the density of all pairs of columns in the fact table:

```
density(C1,C2)=625000/(2000*500)=0.625
density(C1,C3)=196000/(2000*100)=0.98
density(C2,C3)=994/(500*100)=0.01988
```

Step 2: Find the pair of columns with the lowest density. That pair is (C2,C3). Determine which column of the fact table is not in that pair. That column is C1.

Step 3: Make column C1 the third column of the index.

Step 4: Repeat steps 1 through 3 to determine the second and first columns of the index key:

```
density(C2)=433/500=0.866
density(C3)=100/100=1.0
```

The column with the lowest density is C2. Therefore, C3 is the second column of the index. The remaining column, C2, is the first column of the index. That is, the best order for the multi-column index is C2, C3, C1.

Rearranging the order of tables in a FROM clause

The order of tables or views in the FROM CLAUSE can affect the access path. If your query performs poorly, it could be because the join sequence is inefficient. You can determine the join sequence within a query block from the PLANNO column in the PLAN_TABLE. For information on using the PLAN_TABLE, see Chapter 27, "Using EXPLAIN to improve SQL performance," on page 789. If you

think that the join sequence is inefficient, try rearranging the order of the tables and views in the FROM clause to match a join sequence that might perform better.

Updating catalog statistics

If you have the proper authority, you can influence access path selection by using an SQL UPDATE or INSERT statement to change statistical values in the DB2 catalog. However, this is not generally recommended except as a last resort. Although updating catalog statistics can help a certain query, other queries can be affected adversely. Also, the UPDATE statements must be repeated after RUNSTATS resets the catalog values. You should be very careful if you attempt to update statistics. .

If you update catalog statistics for a table space or index manually, and you are using dynamic statement caching, you need to invalidate statements in the cache that involve those table spaces or indexes. To invalidate statements in the dynamic statement cache without updating catalog statistics or generating reports, you can run the RUNSTATS utility with the REPORT NO and UPDATE NONE options on the table space or the index that the query is dependent on.

The example shown in Figure 227 on page 781, involves this query:

```
SELECT * FROM PART_HISTORY -- SELECT ALL PARTS
WHERE PART_TYPE = 'BB'     P1 -- THAT ARE 'BB' TYPES
    AND W_FROM = 3         P2 -- THAT WERE MADE IN CENTER 3
    AND W_NOW = 3          P3 -- AND ARE STILL IN CENTER 3
```

This query has a problem with data correlation. DB2 does not know that 50% of the parts that were made in Center 3 are still in Center 3. The problem was circumvented by making a predicate nonindexable. But suppose that hundreds of users are writing queries similar to that query. Having all users change their queries would be impossible. In this type of situation, the best solution is to change the catalog statistics.

For the query in Figure 227 on page 781, you can update the catalog statistics in one of two ways:

- Run the RUNSTATS utility, and request statistics on the correlated columns W_FROM and W_NOW. This is the preferred method. See the discussion of maintaining statistics in the catalog in Part 5 (Volume 2) of *DB2 Administration Guide* and Part 2 of *DB2 Utility Guide and Reference* for more information.
- Update the catalog statistics manually.

Updating the catalog to adjust for correlated columns: One catalog table that you can update is SYSIBM.SYSCOLDIST, which gives information about a column or set of columns in a table. Assume that because columns W_NOW and W_FROM are correlated, only 100 distinct values exist for the combination of the two columns, rather than 2500 (50 for W_FROM * 50 for W_NOW). Insert a row like this to indicate the new cardinality:

```
INSERT INTO SYSIBM.SYSCOLDIST
(FREQUENCY, FREQUENCYF, IBMREQD,
 TBOWNER, TBNAME, NAME, COLVALUE,
 TYPE, CARDF, COLGROUPCOLNO, NUMCOLUMNS)
VALUES(0, -1, 'N',
 'USRT001','PART_HISTORY','W_FROM',' ',
 'C',100,X'00040003',2);
```

You can also use the RUNSTATS utility to put this information in SYSCOLDIST. See *DB2 Utility Guide and Reference* for more information.

You tell DB2 about the frequency of a certain combination of column values by updating SYSIBM.SYSCOLDIST. For example, you can indicate that 1% of the rows in PART_HISTORY contain the values 3 for W_FROM and 3 for W_NOW by inserting this row into SYSCOLDIST:

```
INSERT INTO SYSIBM.SYSCOLDIST
(FREQUENCY, FREQUENCYF, STATTIME, IBMREQD,
TBOWNER, TBNAME, NAME, COLVALUE,
TYPE, CARDF, COLGROUPCOLNO, NUMCOLUMNS)
VALUES(0, .0100, '1996-12-01-12.00.00.000000', 'N',
'USRT001', 'PART_HISTORY', 'W_FROM', X'00800000030080000003',
'F', -1, X'00040003', 2);
```

Updating the catalog for joins with table functions: Updating catalog statistics might cause extreme performance problems if the statistics are not updated correctly. Monitor performance, and be prepared to reset the statistics to their original values if performance problems arise.

Using a subsystem parameter

This section describes subsystem parameters that influence access path selection. To set subsystem parameters, modify and run installation job DSNTIJUZ. See Part 2 of *DB2 Installation Guide* for information about how to set subsystem parameters. This section contains the following topics:

- “Using a subsystem parameter to favor matching index access”
- “Using a subsystem parameter to optimize queries with IN-list predicates” on page 787

Using a subsystem parameter to favor matching index access

DB2 often does a table space scan or nonmatching index scan when the data access statistics indicate that a table is small, even though matching index access is possible. This is a problem if the table is small or empty when statistics are collected, but the table is large when it is queried. In that case, the statistics are not accurate and can lead DB2 to pick an inefficient access path.

The best solution to the problem is to run RUNSTATS again after the table is populated. However, if you cannot do that, you can use subsystem parameter NPGTHRSH to cause DB2 to favor matching index access over a table space scan and over nonmatching index access.

The value of NPGTHRSH is an integer that indicates the tables for which DB2 favors matching index access. Values of NPGTHRSH and their meanings are:

-1	DB2 favors matching index access for all tables.
0	DB2 selects the access path based on cost, and no tables qualify for special handling. This is the default.
$n \geq 1$	If data access statistics have been collected for all tables, DB2 favors matching index access for tables for which the total number of pages on which rows of the table appear (NPAGES) is less than n .

Tables with default statistics for NPAGES (NPAGES = -1) are presumed to have 501 pages. For such tables, DB2 will favor matching index access only when NPGTHRSH is set above 501.

Recommendation: Before you use NPGTHRSH, be aware that in some cases, matching index access can be more costly than a table space scan or nonmatching index access. Specify a small value for NPGTHRSH (10 or less), which limits the

| number of tables for which DB2 favors matching index access. If you need to use
| matching index access only for specific tables, create or alter those tables with the
| VOLATILE parameter, rather than using the system-wide NPGTHRSH parameter.
| See “Favoring index access” on page 778.

Using a subsystem parameter to optimize queries with IN-list predicates

| You can use the INLISTP parameter to control IN-list predicate optimization. If you
| set the INLISTP parameter to a number n that is between 1 and 5000, DB2
| optimizes for an IN-list predicate with up to n values. If you set the INLISTP
| predicate to zero, the optimization is disabled. The default value for the INLISTP
| parameter is 50.

| When you enable the INLISTP parameter, you enable two primary means of
| optimizing some queries that contain IN-list predicates:

- The IN-list predicate is pushed down from the parent query block into the materialized table expression.
- A correlated IN-list predicate in a subquery that is generated by transitive closure is moved up to the parent query block.

Chapter 27. Using EXPLAIN to improve SQL performance

The information under this heading, up to the end of this chapter, is Product-sensitive Programming Interface and Associated Guidance Information, as defined in “Notices” on page 1183.

Definitions and purpose: EXPLAIN is a monitoring tool that produces information about the following:

- A plan, package, or SQL statement when it is bound. The output appears in a table that you create called PLAN_TABLE, which is also called a *plan table*. For experienced users, you can use PLAN_TABLE to give optimization hints to DB2.
- An estimated cost of executing an SQL SELECT, INSERT, UPDATE, or DELETE statement. The output appears in a table that you create called DSN_STATEMENT_TABLE, which is also called a *statement table*. For more information about statement tables, see “Estimating a statement’s cost” on page 842.
- User-defined functions referred to in the statement, including the specific name and schema. The output appears in a table that you create called DSN_FUNCTION_TABLE, which is also called a *function table*. For more information about function tables, see “Ensuring that DB2 executes the intended user-defined function” on page 356.

Other tools: The following tools can help you tune SQL queries:

- **DB2 Visual Explain**

Visual Explain is a graphical workstation feature of DB2 that provides:

- An easy-to-understand display of a selected access path
- Suggestions for changing an SQL statement
- An ability to invoke EXPLAIN for dynamic SQL statements
- An ability to provide DB2 catalog statistics for referenced objects of an access path
- A subsystem parameter browser with keyword 'Find' capabilities

For information about using DB2 Visual Explain, which is a separately packaged CD-ROM provided with your DB2 UDB for z/OS Version 8 license, see *DB2 Visual Explain online help*.

- **OMEGAMON Performance Expert**

OMEGAMON is a performance monitoring tool that formats performance data. OMEGAMON combines information from EXPLAIN and from the DB2 catalog. It displays access paths, indexes, tables, table spaces, plans, packages, DBRMs, host variable definitions, ordering, table access and join sequences, and lock types. Output is presented in a dialog rather than as a table, making the information easy to read and understand. DB2 Performance Monitor (DB2 PM) performs some of the functions of OMEGAMON Performance Expert.

- **DB2 Estimator**

DB2 Estimator for Windows is an easy-to-use, stand-alone tool for estimating the performance of DB2 UDB for z/OS applications. You can use it to predict the performance and cost of running the applications, transactions, SQL statements, triggers, and utilities. For instance, you can use DB2 Estimator for estimating the impact of adding or dropping an index from a table, estimating the change in

response time from adding processor resources, and estimating the amount of time a utility job will take to run. DB2 Estimator for Windows can be downloaded from the Web.

- # • DB2-supplied EXPLAIN stored procedure. Users with authority to run EXPLAIN
- # directly can obtain access path information by calling the DB2-supplied
- # EXPLAIN stored procedure. For more information about the DB2-supplied
- # EXPLAIN stored procedure, see Appendix J, “DB2-supplied stored procedures,”
- # on page 1127.

Chapter overview: This chapter includes the following topics:

- “Obtaining PLAN_TABLE information from EXPLAIN”
- “Asking questions about data access” on page 800
- “Interpreting access to a single table” on page 809
- “Interpreting access to two or more tables (join)” on page 815
- “Interpreting data prefetch” on page 830
- “Determining sort activity” on page 834
- “Processing for views and nested table expressions” on page 836
- “Estimating a statement’s cost” on page 842

See also Chapter 28, “Parallel operations and query performance,” on page 847.

Obtaining PLAN_TABLE information from EXPLAIN

The information in PLAN_TABLE can help you to:

- Design databases, indexes, and application programs
- Determine when to rebind an application
- Determine the access path chosen for a query

For each access to a single table, EXPLAIN tells you if an index access or table space scan is used. If indexes are used, EXPLAIN tells you how many indexes and index columns are used and what I/O methods are used to read the pages. For joins of tables, EXPLAIN tells you which join method and type are used, the order in which DB2 joins the tables, and when and why it sorts any rows.

The primary use of EXPLAIN is to observe the access paths for the SELECT parts of your statements. For UPDATE and DELETE WHERE CURRENT OF, and for INSERT, you receive somewhat less information in your plan table. And some accesses EXPLAIN does not describe: for example, the access to LOB values, which are stored separately from the base table, and access to parent or dependent tables needed to enforce referential constraints.

The access paths shown for the example queries in this chapter are intended only to illustrate those examples. If you execute the queries in this chapter on your system, the access paths chosen can be different.

Steps to obtain PLAN_TABLE information: Use the following overall steps to obtain information from EXPLAIN:

1. Have appropriate access to a plan table. To create the table, see “Creating PLAN_TABLE” on page 791.
2. Populate the table with the information you want. For instructions, see “Populating and maintaining a plan table” on page 798.
3. Select the information you want from the table. For instructions, see “Reordering rows from a plan table” on page 799.

Creating PLAN_TABLE

Before you can use EXPLAIN, a PLAN_TABLE must be created to hold the results of EXPLAIN. A copy of the statements that are needed to create the table are in the DB2 sample library, under the member name DSNTESC. (Unless you need the information that they provide, you do not need to create a function table or statement table to use EXPLAIN.)

Figure 228 on page 792 shows most current format of a plan table, which consists of 58 columns. Table 101 on page 794 shows the content of each column.


```

CREATE TABLE userid.PLAN_TABLE
    (QUERYNO          INTEGER          NOT NULL,
     QBLOCKNO         SMALLINT         NOT NULL,
     APPLNAME          CHAR(8)          NOT NULL,
|     PROGNAME         VARCHAR(128)     NOT NULL,
     PLANNO           SMALLINT         NOT NULL,
     METHOD             SMALLINT         NOT NULL,
|     CREATOR          VARCHAR(128)     NOT NULL,
|     TNAME            VARCHAR(128)     NOT NULL,
     TABNO            SMALLINT         NOT NULL,
     ACCESTYPE         CHAR(2)          NOT NULL,
     MATCHCOLS         SMALLINT         NOT NULL,
|     ACCESSCREATOR    VARCHAR(128)     NOT NULL,
|     ACCESSNAME       VARCHAR(128)     NOT NULL,
     INDEXONLY        CHAR(1)          NOT NULL,
     SORTN_UNIQ        CHAR(1)          NOT NULL,
     SORTN_JOIN        CHAR(1)          NOT NULL,
     SORTN_ORDERBY     CHAR(1)          NOT NULL,
     SORTN_GROUPBY     CHAR(1)          NOT NULL,
     SORTC_UNIQ        CHAR(1)          NOT NULL,
     SORTC_JOIN        CHAR(1)          NOT NULL,
     SORTC_ORDERBY     CHAR(1)          NOT NULL,
     SORTC_GROUPBY     CHAR(1)          NOT NULL,
     TSLOCKMODE        CHAR(3)          NOT NULL,
     TIMESTAMP         CHAR(16)         NOT NULL,
|     REMARKS          VARCHAR(762)     NOT NULL,
|     PREFETCH         CHAR(1)          NOT NULL WITH DEFAULT,
#     COLUMN_FN_EVAL   CHAR(1)          NOT NULL WITH DEFAULT,
#     MIXOPSEQ         SMALLINT         NOT NULL WITH DEFAULT,
#     VERSION          VARCHAR(64)      NOT NULL WITH DEFAULT,
#     COLLID           VARCHAR(128)     NOT NULL WITH DEFAULT,
     ACCESS_DEGREE     SMALLINT         ,
     ACCESS_PGROUP_ID  SMALLINT         ,
     JOIN_DEGREE       SMALLINT         ,
     JOIN_PGROUP_ID    SMALLINT         ,
     SORTC_PGROUP_ID   SMALLINT         ,
     SORTN_PGROUP_ID   SMALLINT         ,
     PARALLELISM_MODE  CHAR(1)          ,
#     MERGE_JOIN_COLS  SMALLINT         ,
#     CORRELATION_NAME VARCHAR(128)     ,
     PAGE_RANGE        CHAR(1)          NOT NULL WITH DEFAULT,
     JOIN_TYPE         CHAR(1)          NOT NULL WITH DEFAULT,
     GROUP_MEMBER      CHAR(8)          NOT NULL WITH DEFAULT,
|     IBM_SERVICE_DATA VARCHAR(254)     FOR BIT DATA NOT NULL WITH DEFAULT,
     WHEN_OPTIMIZE     CHAR(1)          NOT NULL WITH DEFAULT,
     QBLOCK_TYPE       CHAR(6)          NOT NULL WITH DEFAULT,
     BIND_TIME         TIMESTAMP        NOT NULL WITH DEFAULT,
|     OPTHINT          VARCHAR(128)     NOT NULL WITH DEFAULT,
|     HINT_USED        VARCHAR(128)     NOT NULL WITH DEFAULT,
     PRIMARY_ACCESTYPE CHAR(1)          NOT NULL WITH DEFAULT,
     PARENT_QBLOCKNO   SMALLINT         NOT NULL WITH DEFAULT,
     TABLE_TYPE       CHAR(1)          ,
|     TABLE_ENCODE    CHAR(1)          NOT NULL WITH DEFAULT,
|     TABLE_SCCSID    SMALLINT         NOT NULL WITH DEFAULT,
|     TABLE_MCCSID    SMALLINT         NOT NULL WITH DEFAULT,
|     TABLE_DCCSID    SMALLINT         NOT NULL WITH DEFAULT,
|     ROUTINE_ID        INTEGER          NOT NULL WITH DEFAULT,
|     CTEREF           SMALLINT         NOT NULL WITH DEFAULT,
|     STMTOKEN          VARCHAR(240))
    IN database-name.table-space-name
    CCSID EBCDIC;

```

Figure 228. 58-column format of PLAN_TABLE

Your plan table can use many other formats with fewer columns, as shown in Figure 229. However, use the 58-column format because it gives you the most information. If you alter an existing plan table with fewer than 58 columns to the 58-column format:

- If they exist, change the data type of columns: PROGNAME, CREATOR, TNAME, ACESSTYPE, ACCESSNAME, REMARKS, COLLID, CORRELATION_NAME, IBM_SERVICE_DATA, OPTHINT, and HINT_USED. Use the values shown in Figure 228 on page 792.
- Add the missing columns to the table. Use the column definitions shown in Figure 228 on page 792. For most columns added, specify NOT NULL WITH DEFAULT so that default values are included for the rows in the table. However, as the figure shows, certain columns do allow nulls. Do not specify those columns as NOT NULL WITH DEFAULT.

QUERYNO	INTEGER	NOT NULL	ACCESS_DEGREE	SMALLINT	
QBLOCKNO	SMALLINT	NOT NULL	ACCESS_PGROUP_ID	SMALLINT	
APPLNAME	CHAR(8)	NOT NULL	JOIN_DEGREE	SMALLINT	
PROGNAME	CHAR(8)	NOT NULL	JOIN_PGROUP_ID	SMALLINT	
PLANNO	SMALLINT	NOT NULL	-----34 column format-----		
METHOD	SMALLINT	NOT NULL	SORTC_PGROUP_ID	SMALLINT	
CREATOR	CHAR(8)	NOT NULL	SORTN_PGROUP_ID	SMALLINT	
TNAME	CHAR(18)	NOT NULL	PARALLELISM_MODE	CHAR(1)	
TABNO	SMALLINT	NOT NULL	MERGE_JOIN_COLS	SMALLINT	
ACESSTYPE	CHAR(2)	NOT NULL	CORRELATION_NAME	CHAR(18)	
MATCHCOLS	SMALLINT	NOT NULL	PAGE_RANGE	CHAR(1)	NOT NULL WITH DEFAULT
ACCESSCREATOR	CHAR(8)	NOT NULL	JOIN_TYPE	CHAR(1)	NOT NULL WITH DEFAULT
ACCESSNAME	CHAR(18)	NOT NULL	GROUP_MEMBER	CHAR(8)	NOT NULL WITH DEFAULT
INDEXONLY	CHAR(1)	NOT NULL	IBM_SERVICE_DATA	VARCHAR(254)	NOT NULL WITH DEFAULT
SORTN_UNIQ	CHAR(1)	NOT NULL	-----43 column format-----		
SORTN_JOIN	CHAR(1)	NOT NULL	WHEN_OPTIMIZE	CHAR(1)	NOT NULL WITH DEFAULT
SORTN_ORDERBY	CHAR(1)	NOT NULL	QBLOCK_TYPE	CHAR(6)	NOT NULL WITH DEFAULT
SORTN_GROUPBY	CHAR(1)	NOT NULL	BIND_TIME	TIMESTAMP	NOT NULL WITH DEFAULT
SORTC_UNIQ	CHAR(1)	NOT NULL	-----46 column format-----		
SORTC_JOIN	CHAR(1)	NOT NULL	OPTHINT	CHAR(8)	NOT NULL WITH DEFAULT
SORTC_ORDERBY	CHAR(1)	NOT NULL	HINT_USED	CHAR(8)	NOT NULL WITH DEFAULT
SORTC_GROUPBY	CHAR(1)	NOT NULL	PRIMARY_ACESSTYPE	CHAR(1)	NOT NULL WITH DEFAULT
TSLOCKMODE	CHAR(3)	NOT NULL	-----49 column format-----		
TIMESTAMP	CHAR(16)	NOT NULL	PARENT_QBLOCKNO	SMALLINT	NOT NULL WITH DEFAULT
REMARKS	VARCHAR(254)	NOT NULL	TABLE_TYPE	CHAR(1)	
-----25 column format-----			-----51 column format-----		
PREFETCH	CHAR(1)	NOT NULL WITH DEFAULT			
COLUMN_FN_EVAL	CHAR(1)	NOT NULL WITH DEFAULT			
MIXOPSEQ	SMALLINT	NOT NULL WITH DEFAULT			
-----28 column format-----					
VERSION	VARCHAR(64)	NOT NULL WITH DEFAULT			
COLLID	CHAR(18)	NOT NULL WITH DEFAULT			
-----30 column format-----					

Figure 229. Formats of PLAN_TABLE prior to the 58-column format

Table 101 on page 794 shows the descriptions of the columns in PLAN_TABLE.

Table 101. Descriptions of columns in PLAN_TABLE

Column Name	Description
QUERYNO	A number intended to identify the statement being explained. For a row produced by an EXPLAIN statement, specify the number in the QUERYNO clause. For a row produced by non-EXPLAIN statements, specify the number using the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE and DELETE statement syntax. Otherwise, DB2 assigns a number based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values greater than 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of TIMESTAMP is unique.
QBLOCKNO	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	The name of the application plan for the row. Applies only to embedded EXPLAIN statements executed from a plan or to statements explained when binding a plan. Blank if not applicable.
PROGNAME	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. Blank if not applicable.
PLANNO	The number of the step in which the query indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.
METHOD	A number (0, 1, 2, 3, or 4) that indicates the join method used for the step: 0 First table accessed, continuation of previous table accessed, or not used. 1 <i>Nested loop</i> join. For each row of the present composite table, matching rows of a new table are found and joined. 2 <i>Merge scan</i> join. The present composite table and the new table are scanned in the order of the join columns, and matching rows are joined. 3 Sorts needed by ORDER BY, GROUP BY, SELECT DISTINCT, UNION, a quantified predicate, or an IN predicate. This step does not access a new table. 4 <i>Hybrid</i> join. The current composite table is scanned in the order of the join-column rows of the new table. The new table is accessed using list prefetch.
CREATOR	The creator of the new table accessed in this step, blank if METHOD is 3.
TNAME	The name of a table, materialized query table, created or declared temporary table, materialized view, or materialized table expression. The value is blank if METHOD is 3. The column can also contain the name of a table in the form DSNWFQB(<i>qblockno</i>). DSNWFQB(<i>qblockno</i>) is used to represent the intermediate result of a UNION ALL or an outer join that is materialized. If a view is merged, the name of the view does not appear.
TABNO	Values are for IBM use only.

Table 101. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
ACCESSTYPE	The method of accessing the new table: I By an index (identified in ACCESSCREATOR and ACCESSNAME) I1 By a one-fetch index scan M By a multiple index scan (followed by MX, MI, or MU) MI By an intersection of multiple indexes MU By a union of multiple indexes MX By an index scan on the index named in ACCESSNAME N By an index scan when the matching predicate contains the IN keyword R By a table space scan RW By a work file scan of the result of a materialized user-defined table function T By a sparse index (star join work files) V By buffers for an INSERT statement within a SELECT blank Not applicable to the current row
MATCHCOLS	For ACCESSTYPE I, I1, N, or MX, the number of index keys used in an index scan; otherwise, 0.
ACCESSCREATOR	For ACCESSTYPE I, I1, N, or MX, the creator of the index; otherwise, blank.
ACCESSNAME	For ACCESSTYPE I, I1, N, or MX, the name of the index; otherwise, blank.
INDEXONLY	Whether access to an index alone is enough to carry out the step, or whether data too must be accessed. Y=Yes; N=No.
SORTN_UNIQ	Whether the new table is sorted to remove duplicate rows. Y=Yes; N=No.
SORTN_JOIN	Whether the new table is sorted for join method 2 or 4. Y=Yes; N=No.
SORTN_ORDERBY	Whether the new table is sorted for ORDER BY. Y=Yes; N=No.
SORTN_GROUPBY	Whether the new table is sorted for GROUP BY. Y=Yes; N=No.
SORTC_UNIQ	Whether the composite table is sorted to remove duplicate rows. Y=Yes; N=No.
SORTC_JOIN	Whether the composite table is sorted for join method 1, 2 or 4. Y=Yes; N=No.
SORTC_ORDERBY	Whether the composite table is sorted for an ORDER BY clause or a quantified predicate. Y=Yes; N=No.
SORTC_GROUPBY	Whether the composite table is sorted for a GROUP BY clause. Y=Yes; N=No.
TSLOCKMODE	An indication of the mode of lock to be acquired on either the new table, or its table space or table space partitions. If the isolation can be determined at bind time, the values are: IS Intent share lock IX Intent exclusive lock S Share lock U Update lock X Exclusive lock SIX Share with intent exclusive lock N UR isolation; no lock If the isolation cannot be determined at bind time, then the lock mode determined by the isolation at run time is shown by the following values. NS For UR isolation, no lock; for CS, RS, or RR, an S lock. NIS For UR isolation, no lock; for CS, RS, or RR, an IS lock. NSS For UR isolation, no lock; for CS or RS, an IS lock; for RR, an S lock. SS For UR, CS, or RS isolation, an IS lock; for RR, an S lock. The data in this column is right justified. For example, IX appears as a blank followed by I followed by X. If the column contains a blank, then no lock is acquired.
TIMESTAMP	Usually, the time at which the row is processed, to the last .01 second. If necessary, DB2 adds .01 second to the value to ensure that rows for two successive queries have different values.
REMARKS	A field into which you can insert any character string of 762 or fewer characters.

Table 101. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
PREFETCH	Whether data pages are to be read in advance by prefetch:
#	S Pure sequential prefetch
#	L Prefetch through a page list
#	D Possible candidate for dynamic prefetch
#	blank Unknown or no prefetch
COLUMN_FN_EVAL	When an SQL aggregate function is evaluated. R = while the data is being read from the table or index; S = while performing a sort to satisfy a GROUP BY clause; blank = after data retrieval and after any sorts.
MIXOPSEQ	The sequence number of a step in a multiple index operation.
	1, 2, ... n For the steps of the multiple index procedure (ACCESSTYPE is MX, MI, or MU.)
	0 For any other rows (ACCESSTYPE is I, II, M, N, R, or blank.)
VERSION	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. Blank if not applicable.
COLLID	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. Blank if not applicable. The value DSNDYNAMICSQLCACHE indicates that the row is for a cached statement.
Note: The following nine columns, from ACCESS_DEGREE through CORRELATION_NAME, contain the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, each of them can contain null if the method it refers to does not apply.	
ACCESS_DEGREE	The number of parallel tasks or operations activated by a query. This value is determined at bind time; the actual number of parallel operations used at execution time could be different. This column contains 0 if there is a host variable.
ACCESS_PGROUP_ID	The identifier of the parallel group for accessing the new table. A parallel group is a set of consecutive operations, executed in parallel, that have the same number of parallel tasks. This value is determined at bind time; it could change at execution time.
JOIN_DEGREE	The number of parallel operations or tasks used in joining the composite table with the new table. This value is determined at bind time and can be 0 if there is a host variable. The actual number of parallel operations or tasks used at execution time could be different.
JOIN_PGROUP_ID	The identifier of the parallel group for joining the composite table with the new table. This value is determined at bind time; it could change at execution time.
SORTC_PGROUP_ID	The parallel group identifier for the parallel sort of the composite table.
SORTN_PGROUP_ID	The parallel group identifier for the parallel sort of the new table.
PARALLELISM_MODE	The kind of parallelism, if any, that is used at bind time: I Query I/O parallelism C Query CP parallelism X Sysplex query parallelism
MERGE_JOIN_COLS	The number of columns that are joined during a merge scan join (Method=2).
# CORRELATION_NAME	The correlation name of a table or view that is specified in the statement. If there is no correlation name, then the column is null.
#	
PAGE_RANGE	Whether the table qualifies for page range screening, so that plans scan only the partitions that are needed. Y = Yes; blank = No.

Table 101. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
JOIN_TYPE	The type of join: F FULL OUTER JOIN L LEFT OUTER JOIN S STAR JOIN blank INNER JOIN or no join RIGHT OUTER JOIN converts to a LEFT OUTER JOIN when you use it, so that JOIN_TYPE contains L.
GROUP_MEMBER	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
IBM_SERVICE_DATA	Values are for IBM use only.
WHEN_OPTIMIZE	When the access path was determined: blank At bind time, using a default filter factor for any host variables, parameter markers, or special registers. B At bind time, using a default filter factor for any host variables, parameter markers, or special registers; however, the statement is reoptimized at run time using input variable values for input host variables, parameter markers, or special registers. The bind option REOPT(ALWAYS) or REOPT(ONCE) must be specified for reoptimization to occur. R At run time, using input variables for any host variables, parameter markers, or special registers. The bind option REOPT(ALWAYS) or REOPT(ONCE) must be specified for this to occur.
QBLOCK_TYPE	For each query block, an indication of the type of SQL operation performed. For the outermost query, this column identifies the statement type. Possible values: SELECT SELECT INSERT INSERT UPDATE UPDATE DELETE DELETE SELUPD SELECT with FOR UPDATE OF DELCUR DELETE WHERE CURRENT OF CURSOR UPDCUR UPDATE WHERE CURRENT OF CURSOR CORSUB Correlated subselect or fullselect NCOSUB Noncorrelated subselect or fullselect TABLEX Table expression TRIGGR WHEN clause on CREATE TRIGGER UNION UNION UNIONA UNION ALL
# BIND_TIME	For static SQL statements, the time at which the plan or package for this statement or query block was bound. For cached dynamic SQL statements, the time at which the statement entered the cache. For static and cached dynamic SQL statements, this is a full-precision timestamp value. For non-cached dynamic SQL statements, this is the value contained in the TIMESTAMP column of PLAN_TABLE appended by 4 zeroes.
OPTHINT	A string that you use to identify this row as an optimization hint for DB2. DB2 uses this row as input when choosing an access path.
HINT_USED	If DB2 used one of your optimization hints, it puts the identifier for that hint (the value in OPTHINT) in this column.
PRIMARY_ACCESTYPE	Indicates whether direct row access will be attempted first: D DB2 will try to use direct row access. If DB2 cannot use direct row access at run time, it uses the access path described in the ACCESTYPE column of PLAN_TABLE. blank DB2 will not try to use direct row access.

Table 101. Descriptions of columns in PLAN_TABLE (continued)

Column Name	Description
PARENT_QBLOCKNO	A number that indicates the QBLOCKNO of the parent query block.
TABLE_TYPE	The type of new table: B Buffers for an INSERT statement within a SELECT C Common table expression F Table function M Materialized query table Q Temporary intermediate result table (not materialized). For the name of a view or nested table expression, a value of Q indicates that the materialization was virtual and not actual. Materialization can be virtual when the view or nested table expression definition contains a UNION ALL that is not distributed. R Recursive common table expression T Table W Work file The value of the column is null if the query uses GROUP BY, ORDER BY, or DISTINCT, which requires an implicit sort.
TABLE_ENCODE	The encoding scheme of the table. If the table has a single CCSID set, possible values are: A ASCII E EBCDIC U Unicode M is the value of the column when the table contains multiple CCSID set, the value of the column is M.
TABLE_SCCSID	The SBCS CCSID value of the table. If column TABLE_ENCODE is M, the value is 0.
TABLE_MCCSID	The mixed CCSID value of the table. If column TABLE_ENCODE is M, the value is 0.
TABLE_DCCSID	The DBCS CCSID value of the table. If column TABLE_ENCODE is M, the value is 0.
ROUTINE_ID	Values are for IBM use only.
CTEREF	If the referenced table is a common table expression, the value is the top-level query block number.
STMTTOKEN	User-specified statement token.

Populating and maintaining a plan table

For the two distinct ways to populate a plan table, see:

- “Executing the SQL statement EXPLAIN”
- “Binding with the option EXPLAIN(YES)” on page 799

When you populate the plan table through EXPLAIN, any INSERT triggers on the table are not activated. If you insert rows yourself, then those triggers are activated.

For tips on maintaining a growing plan table, see “Maintaining a plan table” on page 799.

Executing the SQL statement EXPLAIN

You can populate a PLAN_TABLE by executing the SQL statement EXPLAIN. In the statement, specify a single explainable SQL statement in the FOR clause.

If an alias is defined on a PLAN_TABLE that was created with a different authorization ID, and you have the appropriate SELECT and INSERT privileges, you can populate that PLAN_TABLE even if you do not own it.

You can execute EXPLAIN either statically from an application program, or dynamically, using QMF or SPUI. For instructions and for details of the authorization that you need on PLAN_TABLE, see *DB2 SQL Reference*.

Binding with the option EXPLAIN(YES)

You can populate a plan table when you bind or rebind a plan or package. Specify the option EXPLAIN(YES). EXPLAIN obtains information about the access paths for all explainable SQL statements in a package or the DBRMs of a plan. The information appears in table *package_owner.PLAN_TABLE* or *plan_owner.PLAN_TABLE*. For dynamically prepared SQL, the qualifier of PLAN_TABLE is the current SQLID.

If the plan owner or the package owner has an alias on a PLAN_TABLE that was created by another owner, *other_owner.PLAN_TABLE* is populated instead of *package_owner.PLAN_TABLE* or *plan_owner.PLAN_TABLE*.

Performance considerations: EXPLAIN as a bind option should not be a performance concern. The same processing for access path selection is performed, regardless of whether you use EXPLAIN(YES) or EXPLAIN (NO). With EXPLAIN(YES), there is only a small amount of overhead processing to put the results in a plan table.

If a plan or package that was previously bound with EXPLAIN(YES) is automatically rebound, the value of field EXPLAIN PROCESSING on installation panel DSNTIPO determines whether EXPLAIN is run again during the automatic rebind. Again, there is a small amount of overhead for inserting the results into a plan table.

EXPLAIN for remote binds: A remote requester that accesses DB2 can specify EXPLAIN(YES) when binding a package at the DB2 server. The information appears in a plan table at the server, not at the requester. If the requester does not support the propagation of the option EXPLAIN(YES), rebind the package at the requester with that option to obtain access path information. You cannot get information about access paths for SQL statements that use private protocol.

Maintaining a plan table

DB2 adds rows to PLAN_TABLE as you choose; it does not automatically delete rows. To clear the table of obsolete rows, use DELETE, just as you would for deleting rows from any table. You can also use DROP TABLE to drop a plan table completely.

Reordering rows from a plan table

Several processes can insert rows into the same plan table. To understand access paths, you must retrieve the rows for a particular query in an appropriate order.

Retrieving rows for a plan

The rows for a particular plan are identified by the value of APPLNAME. The following query to a plan table returns the rows for all the explainable statements in a plan in their logical order:

```
SELECT * FROM JOE.PLAN_TABLE
WHERE APPLNAME = 'APPL1'
ORDER BY TIMESTAMP, QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ;
```

The result of the ORDER BY clause shows whether there are:

- Multiple QBLOCKNOs within a QUERYNO
- Multiple PLANNOs within a QBLOCKNO

- Multiple MIXOPSEQs within a PLANNO

All rows with the same non-zero value for QBLOCKNO and the same value for QUERYNO relate to a step within the query. QBLOCKNOs are not necessarily executed in the order shown in PLAN_TABLE. But within a QBLOCKNO, the PLANNO column gives the substeps in the order they execute.

For each substep, the TNAME column identifies the table accessed. Sorts can be shown as part of a table access or as a separate step.

What if QUERYNO=0? For entries that contain QUERYNO=0, use the timestamp, which is guaranteed to be unique, to distinguish individual statements.

Retrieving rows for a package

The rows for a particular package are identified by the values of PROGNAME, COLLID, and VERSION. Those columns correspond to the following four-part naming convention for packages:

LOCATION.COLLECTION.PACKAGE_ID.VERSION

COLLID gives the COLLECTION name, and PROGNAME gives the PACKAGE_ID. The following query to a plan table return the rows for all the explainable statements in a package in their logical order:

```
SELECT * FROM JOE.PLAN_TABLE
WHERE PROGNAME = 'PACK1' AND COLLID = 'COLL1' AND VERSION = 'PROD1'
ORDER BY QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ;
```

Asking questions about data access

When you examine your EXPLAIN results, try to answer the following questions:

- “Is access through an index? (ACCESSTYPE is I, I1, N or MX)” on page 801
- “Is access through more than one index? (ACCESSTYPE=M)” on page 801
- “How many columns of the index are used in matching? (MATCHCOLS=n)” on page 802
- “Is the query satisfied using only the index? (INDEXONLY=Y)” on page 802
- “Is direct row access possible? (PRIMARY_ACCESSTYPE = D)” on page 803
- “Is a view or nested table expression materialized?” on page 806
- “Was a scan limited to certain partitions? (PAGE_RANGE=Y)” on page 806
- “What kind of prefetching is expected? (PREFETCH = L, S, D, or blank)” on page 807
- “Is data accessed or processed in parallel? (PARALLELISM_MODE is I, C, or X)” on page 807
- “Are sorts performed?” on page 807
- “Is a subquery transformed into a join?” on page 808
- “When are aggregate functions evaluated? (COLUMN_FN_EVAL)” on page 808
- “How many index screening columns are used?” on page 808
- “Is a complex trigger WHEN clause used? (QBLOCKTYPE=TRIGGR)” on page 809

As explained in this section, they can be answered in terms of values in columns of a plan table.

Is access through an index? (ACCESSTYPE is I, I1, N or MX)

If the column ACCESSTYPE in the plan table has one of those values, DB2 uses an index to access the table that is named in column TNAME. The columns ACCESSCREATOR and ACCESSNAME identify the index. For a description of methods of using indexes, see “Index access paths” on page 810.

Is access through more than one index? (ACCESSTYPE=M)

The value M indicates that DB2 uses a set of indexes to access a single table. A set of rows in the plan table contain information about the multiple index access. The rows are numbered in column MIXOPSEQ in the order of execution of steps in the multiple index access. (If you retrieve the rows in order by MIXOPSEQ, the result is similar to postfix arithmetic notation.)

Both of the following examples have these indexes: IX1 on T(C1) and IX2 on T(C2).

Example: Suppose that you issue the following SELECT statement:

```
SELECT * FROM T
WHERE C1 = 1 AND C2 = 1;
```

DB2 processes the query by performing the following steps:

1. DB2 retrieves all the qualifying record identifiers (RIDs) where C1=1, by using index IX1.
2. DB2 retrieves all the qualifying RIDs where C2=1, by using index IX2. The intersection of these lists is the final set of RIDs.
3. DB2 accesses the data pages that are needed to retrieve the qualified rows by using the final RID list.

The plan table for this example is shown in Table 102.

Table 102. *PLAN_TABLE* output for example with intersection (AND) operator

TNAME	ACCESS- TYPE	MATCH- COLS	ACCESS- NAME	INDEX- ONLY	PREFETCH	MIXOP- SEQ
T	M	0		N	L	0
T	MX	1	IX1	Y		1
T	MX	1	IX2	Y		2
T	MI	0		N		3

Example: Suppose that you issue the following SELECT statement:

```
SELECT * FROM T
WHERE C1 BETWEEN 100 AND 199 OR
      C1 BETWEEN 500 AND 599;
```

In this case, the same index can be used more than once in a multiple index access because more than one predicate could be matching. DB2 processes the query by performing the following steps:

1. DB2 retrieves all RIDs where C1 is between 100 and 199, using index IX1.
2. DB2 retrieves all RIDs where C1 is between 500 and 599, again using IX1. The union of those lists is the final set of RIDs.
3. DB2 retrieves the qualified rows by using the final RID list.

The plan table for this example is shown in Table 103.

Table 103. *PLAN_TABLE* output for example with union (OR) operator

TNAME	ACCESS- TYPE	MATCH- COLS	ACCESS- NAME	INDEX- ONLY	PREFETCH	MIXOP- SEQ
T	M	0		N	L	0
T	MX	1	IX1	Y		1
T	MX	1	IX1	Y		2
T	MU	0		N		3

How many columns of the index are used in matching? (MATCHCOLS=n)

If MATCHCOLS is 0, the access method is called a *nonmatching index scan*. All the index keys and their RIDs are read.

If MATCHCOLS is greater than 0, the access method is called a *matching index scan*: the query uses predicates that match the index columns.

In general, the matching predicates on the leading index columns are equal or IN predicates. The predicate that matches the final index column can be an equal, IN, NOT NULL, or range predicate (<, <=, >, >=, LIKE, or BETWEEN).

The following example illustrates matching predicates:

```
SELECT * FROM EMP
  WHERE JOBCODE = '5' AND SALARY > 60000 AND LOCATION = 'CA';

INDEX XEMP5 on (JOBCODE, LOCATION, SALARY, AGE);
```

The index XEMP5 is the chosen access path for this query, with MATCHCOLS = 3. Two equal predicates are on the first two columns and a range predicate is on the third column. Though the index has four columns in the index, only three of them can be considered matching columns.

Is the query satisfied using only the index? (INDEXONLY=Y)

In this case, the method is called *index-only access*. For a SELECT operation, all the columns needed for the query can be found in the index and DB2 does not access the table. For an UPDATE or DELETE operation, only the index is required to read the selected row.

Index-only access to data is not possible for any step that uses list prefetch, which is described under “What kind of prefetching is expected? (PREFETCH = L, S, D, or blank)” on page 807. Index-only access is not possible for padded indexes when varying-length data is returned or a VARCHAR column has a LIKE predicate, unless the VARCHAR FROM INDEX field of installation panel DSNTIP4 is set to YES and plan or packages have been rebound to pick up the change. See Part 2 of *DB2 Installation Guide* for more information. Index-only access is always possible for nonpadded indexes.

If access is by more than one index, INDEXONLY is Y for a step with access type MX, because the data pages are not actually accessed until all the steps for intersection (MI) or union (MU) take place.

When an SQL application uses index-only access for a ROWID column, the application claims the table space or table space partition. As a result, contention may occur between the SQL application and a utility that drains the table space or partition. Index-only access to a table for a ROWID column is not possible if the associated table space or partition is in an incompatible restrictive state. For example, an SQL application can make a read claim on the table space only if the restrictive state allows readers.

Is direct row access possible? (PRIMARY_ACCESTYPE = D)

If an application selects a row from a table that contains a ROWID column, the row ID value implicitly contains the location of the row. If you use that row ID value in the search condition of subsequent SELECTs, DELETEs, or UPDATEs, DB2 might be able to navigate directly to the row. This access method is called *direct row access*.

Direct row access is very fast, because DB2 does not need to use the index or a table space scan to find the row. Direct row access can be used on any table that has a ROWID column.

To use direct row access, you first select the values of a row into host variables. The value that is selected from the ROWID column contains the location of that row. Later, when you perform queries that access that row, you include the row ID value in the search condition. If DB2 determines that it can use direct row access, it uses the row ID value to navigate directly to the row. See “Example: Coding with row IDs for direct row access” on page 805 for a coding example.

Which predicates qualify for direct row access?

For a query to qualify for direct row access, the search condition must be a Boolean term *stage 1* predicate that fits one of these descriptions:

1. A simple Boolean term predicate of the form `COL=noncolumn expression`, where COL has the ROWID data type and *noncolumn expression* contains a row ID
2. A simple Boolean term predicate of the form `COL IN list`, where COL has the ROWID data type and the values in *list* are row IDs, and an index is defined on COL
3. A compound Boolean term that combines several simple predicates using the AND operator, and one of the simple predicates fits description 1 or 2

However, just because a query qualifies for direct row access does not mean that access path is always chosen. If DB2 determines that another access path is better, direct row access is not chosen.

Examples: In the following predicate example, ID is a ROWID column in table T1. A unique index exists on that ID column. The host variables are of the ROWID type.

```
WHERE ID IN (:hv_rowid1,:hv_rowid2,:hv_rowid3)
```

The following predicate also qualifies for direct row access:

```
WHERE ID = ROWID(X'F0DFD230E3C0D80D81C201AA0A280100000000000203')
```

Searching for propagated rows: If rows are propagated from one table to another, do not expect to use the same row ID value from the source table to search for the same row in the target table, or vice versa. This does not work when direct row access is the access path chosen.

Example: Assume that the host variable in the following statement contains a row ID from SOURCE:

```
SELECT * FROM TARGET
WHERE ID = :hv_rowid
```

Because the row ID location is not the same as in the source table, DB2 will probably not find that row. Search on another column to retrieve the row you want.

Reverting to ACESSTYPE

Although DB2 might plan to use direct row access, circumstances can cause DB2 to not use direct row access at run time. DB2 remembers the location of the row as of the time it is accessed. However, that row can change locations (such as after a REORG) between the first and second time it is accessed, which means that DB2 cannot use direct row access to find the row on the second access attempt. Instead of using direct row access, DB2 uses the access path that is shown in the ACESSTYPE column of PLAN_TABLE.

If the predicate you are using to do direct row access is not indexable and if DB2 is unable to use direct row access, then DB2 uses a table space scan to find the row. This can have a profound impact on the performance of applications that rely on direct row access. Write your applications to handle the possibility that direct row access might not be used. Some options are to:

- Ensure that your application does not try to remember ROWID columns across reorganizations of the table space.

When your application commits, it releases its claim on the table space; it is possible that a REORG can run and move the row, which disables direct row access. Plan your commit processing accordingly; use the returned row ID value before committing, or re-select the row ID value after a commit is issued.

If you are storing ROWID columns from another table, update those values after the table with the ROWID column is reorganized.

- Create an index on the ROWID column, so that DB2 can use the index if direct row access is disabled.
- Supplement the ROWID column predicate with another predicate that enables DB2 to use an existing index on the table. For example, after reading a row, an application might perform the following update:

```
EXEC SQL UPDATE EMP
SET SALARY = :hv_salary + 1200
WHERE EMP_ROWID = :hv_emp_rowid
AND EMPNO = :hv_empno;
```

If an index exists on EMPNO, DB2 can use index access if direct access fails. The additional predicate ensures DB2 does not revert to a table space scan.

Using direct row access and other access methods

Parallelism: Direct row access and parallelism are mutually exclusive. If a query qualifies for both direct row access and parallelism, direct row access is used. If direct row access fails, DB2 does not revert to parallelism; instead it reverts to the backup access type (as designated by column ACESSTYPE in the PLAN_TABLE). This might result in a table space scan. To avoid a table space scan in case direct row access fails, add an indexed column to the predicate.

RID list processing: Direct row access and RID list processing are mutually exclusive. If a query qualifies for both direct row access and RID list processing,

direct row access is used. If direct row access fails, DB2 does not revert to RID list processing; instead it reverts to the backup access type.

Example: Coding with row IDs for direct row access

Figure 230 is a portion of a C program that shows you how to obtain the row ID value for a row, and then to use that value to find the row efficiently when you want to modify it.

```

/*****
/* Declare host variables */
*****/
EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS BLOB_LOCATOR hv_picture;
    SQL TYPE IS CLOB_LOCATOR hv_resume;
    SQL TYPE IS ROWID hv_emp_rowid;
    short hv_dept, hv_id;
    char hv_name[30];
    decimal hv_salary[5,2];
EXEC SQL END DECLARE SECTION;

/*****
/* Retrieve the picture and resume from the PIC_RES table */
*****/
strcpy(hv_name, "Jones");
EXEC SQL SELECT PR.PICTURE, PR.RESUME INTO :hv_picture, :hv_resume
FROM PIC_RES PR
WHERE PR.Name = :hv_name;

/*****
/* Insert a row into the EMPDATA table that contains the
/* picture and resume you obtained from the PIC_RES table */
*****/
EXEC SQL INSERT INTO EMPDATA
VALUES (DEFAULT,9999,'Jones', 35000.00, 99,
:hv_picture, :hv_resume);

/*****
/* Now retrieve some information about that row,
/* including the ROWID value.
*****/
hv_dept = 99;
EXEC SQL SELECT E.SALARY, E.EMP_ROWID
INTO :hv_salary, :hv_emp_rowid
FROM EMPDATA E
WHERE E.DEPTNUM = :hv_dept AND E.NAME = :hv_name;
```

Figure 230. Example of using a row ID value for direct row access (Part 1 of 2)

```

/*****
/* Update columns SALARY, PICTURE, and RESUME. Use the
/* ROWID value you obtained in the previous statement
/* to access the row you want to update.
/* smiley_face and update_resume are
/* user-defined functions that are not shown here.
*****/
EXEC SQL UPDATE EMPDATA
    SET SALARY = :hv_salary + 1200,
    PICTURE = smiley_face(:hv_picture),
    RESUME = update_resume(:hv_resume)
    WHERE EMP_ROWID = :hv_emp_rowid;

/*****
/* Use the ROWID value to obtain the employee ID from the
/* same record.
*****/
EXEC SQL SELECT E.ID INTO :hv_id
    FROM EMPDATA E
    WHERE E.EMP_ROWID = :hv_emp_rowid;

/*****
/* Use the ROWID value to delete the employee record
/* from the table.
*****/
EXEC SQL DELETE FROM EMPDATA
    WHERE EMP_ROWID = :hv_emp_rowid;

```

Figure 230. Example of using a row ID value for direct row access (Part 2 of 2)

Is a view or nested table expression materialized?

When the column TNAME names a view or nested table expression and column TABLE_TYPE contains a W, it indicates that the view or nested table expression is materialized. *Materialization* means that the data rows that are selected by the view or nested table expression are put into a work file that is to be processed like a table. (For a more detailed description of materialization, see “Processing for views and nested table expressions” on page 836.)

Was a scan limited to certain partitions? (PAGE_RANGE=Y)

DB2 can limit a scan of data in a partitioned table space to one or more partitions. The method is called a *limited partition scan*. The query must provide a predicate on the first key column of the partitioning index. DB2 can use all leading columns of the partitioning key. The rules for using a partitioning column to limit partitions are the same as the rules for determining a matching column on an index. If a partitioning column is a candidate to be a matching column, that column can limit partitions.

A limited partition scan can be combined with other access methods. For example, consider the following query:

```

SELECT .. FROM T
    WHERE (C1 BETWEEN '2002' AND '3280'
    OR C1 BETWEEN '6000' AND '8000')
    AND C2 = '6';

```

Assume that table T has a partitioned index on column C1 and that values of C1 between 2002 and 3280 all appear in partitions 3 and 4 and the values between 6000 and 8000 appear in partitions 8 and 9. Assume also that T has another index on column C2. DB2 could choose any of these access methods:

- A matching index scan on column C1. The scan reads index values and data only from partitions 3, 4, 8, and 9. (PAGE_RANGE=N)
- A matching index scan on column C2. (DB2 might choose that if few rows have C2=6.) The matching index scan reads all RIDs for C2=6 from the index on C2 and corresponding data pages from partitions 3, 4, 8, and 9. (PAGE_RANGE=Y)
- A table space scan on T. DB2 avoids reading data pages from any partitions except 3, 4, 8 and 9. (PAGE_RANGE=Y)

What kind of prefetching is expected? (PREFETCH = L, S, D, or blank)

Prefetching is a method of determining in advance that a set of data pages is about to be used and then reading the entire set into a buffer with a single asynchronous I/O operation. If the value of PREFETCH is:

- S, the method is called *sequential prefetch*. The data pages that are read in advance are sequential. A table space scan always uses sequential prefetch. An index scan might not use it. For a more complete description, see “Sequential prefetch (PREFETCH=S)” on page 830.
- L, the method is called *list prefetch*. One or more indexes are used to select the RIDs for a list of data pages to be read in advance; the pages need not be sequential. Usually, the RIDs are sorted. The exception is the case of a hybrid join (described under “Hybrid join (METHOD=4)” on page 821) when the value of column SORTN_JOIN is N. For a more complete description, see “List prefetch (PREFETCH=L)” on page 831.
- D, the method is called *dynamic prefetch*. DB2 expects that the pages to be accessed will be sufficiently nonsequential to invoke dynamic prefetch.
- Blank, prefetching is not expected. However, depending on the pattern of the page access, data can be prefetched at execution time through a process called *sequential detection*. For a description of that process, see “Sequential detection at execution time” on page 832.

Is data accessed or processed in parallel? (PARALLELISM_MODE is I, C, or X)

Parallel processing applies only to read-only queries.

If mode is:	DB2 plans to use:
I	Parallel I/O operations
C	Parallel CP operations
X	Sysplex query parallelism

Non-null values in columns ACCESS_DEGREE and JOIN_DEGREE indicate to what degree DB2 plans to use parallel operations. At execution time, however, DB2 might not actually use parallelism, or it might use fewer operations in parallel than were originally planned. For a more complete description, see Chapter 28, “Parallel operations and query performance,” on page 847. For more information about Sysplex query parallelism, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*.

Are sorts performed?

SORTN_JOIN and SORTC_JOIN: SORTN_JOIN indicates that the new table of a join is sorted before the join. (For hybrid join, this is a sort of the RID list.) When SORTN_JOIN and SORTC_JOIN are both ‘Y’, two sorts are performed for the join. The sorts for joins are indicated on the same row as the new table access.

METHOD 3 sorts: These are used for ORDER BY, GROUP BY, SELECT DISTINCT, UNION, or a quantified predicate. A quantified predicate is 'col = ANY (fullselect)' or 'col = SOME (fullselect)'. They are indicated on a separate row. A single row of the plan table can indicate two sorts of a composite table, but only one sort is actually done.

SORTC_UNIQ and SORTC_ORDERBY: SORTC_UNIQ indicates a sort to remove duplicates, as might be needed by a SELECT statement with DISTINCT or UNION. SORTC_ORDERBY usually indicates a sort for an ORDER BY clause. But SORTC_UNIQ and SORTC_ORDERBY also indicate when the results of a noncorrelated subquery are sorted, both to remove duplicates and to order the results. One sort does both the removal and the ordering.

Is a subquery transformed into a join?

For better access paths, DB2 sometimes transforms subqueries into joins, as described in "Conditions for DB2 to transform a subquery into a join" on page 768. A plan table shows that a subquery is transformed into a join by the value in column QBLOCKNO.

- If the subquery is not transformed into a join, that means it is executed in a separate operation, and its value of QBLOCKNO is greater than the value for the outer query.
- If the subquery is transformed into a join, it and the outer query have the same value of QBLOCKNO. A join is also indicated by a value of 1, 2, or 4 in column METHOD.

When are aggregate functions evaluated? (COLUMN_FN_EVAL)

When the aggregate functions are evaluated is based on the access path chosen for the SQL statement.

- If the ACESSTYPE column is I1, then a MAX or MIN function can be evaluated by one access of the index named in ACCESSNAME.
- For other values of ACESSTYPE, the COLUMN_FN_EVAL column tells when DB2 is evaluating the aggregate functions.

Value	Functions are evaluated ...
S	While performing a sort to satisfy a GROUP BY clause
R	While the data is being read from the table or index
blank	After data retrieval and after any sorts

Generally, values of R and S are considered better for performance than a blank.

Use variance and standard deviation with care: The VARIANCE and STDDEV functions are always evaluated late (that is, COLUMN_FN_EVAL is blank). This causes other functions in the same query block to be evaluated late as well. For example, in the following query, the sum function is evaluated later than it would be if the variance function was not present:

```
SELECT SUM(C1), VARIANCE(C1) FROM T1;
```

How many index screening columns are used?

The plan table does not provide the answer to this question. However, you can use Visual Explain to determine how many index screening columns are used. For information about Visual Explain, see DB2 Visual Explain online help.

Is a complex trigger WHEN clause used? (QBLOCKTYPE=TRIGGR)

The plan table does not report simple trigger WHEN clauses, such as WHEN (N.C1 < 5). However, the plan table does report complex trigger WHEN clauses, which are clauses that involve other base tables and transition tables. The QBLOCK_TYPE column of the top level query block shows TRIGGR to indicate a complex trigger WHEN clause.

Example: Consider the following trigger:

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW TABLE AS NT OLD AS O
  FOR EACH STATEMENT MODE DB2SQL
  WHEN (O.ON_HAND < (SELECT MAX(ON_HAND) FROM NT))
  BEGIN ATOMIC
    INSERT INTO ORDER_LOG VALUES (O.PARTNO, O.ON_HAND);
  END
```

Table 104 shows the corresponding plan table for the WHEN clause.

Table 104. Plan table for the WHEN clause

QBLOCKNO	PLANNO	TABLE	ACCESSTYPE	QBLOCK_TYPE	PARENT_QBLOCKNO
1	1			TRIGGR	0
2	1	NT	R	NCOSUB	1

Interpreting access to a single table

The following sections describe different access paths that values in a plan table can indicate, along with suggestions for supplying better access paths for DB2 to choose from:

- Table space scans (ACCESSTYPE=R PREFETCH=S)
- “Index access paths” on page 810
- “UPDATE using an index” on page 815

Table space scans (ACCESSTYPE=R PREFETCH=S)

Table space scan is most often used for one of the following reasons:

- Access is through a created temporary table. (Index access is not possible for created temporary tables.)
- A matching index scan is not possible because an index is not available, or no predicates match the index columns.
- A high percentage of the rows in the table is returned. In this case, an index is not really useful because most rows need to be read anyway.
- The indexes that have matching predicates have low cluster ratios and are therefore efficient only for small amounts of data.

Assume that table T has no index on C1. The following is an example that uses a table space scan:

```
SELECT * FROM T WHERE C1 = VALUE;
```

In this case, at least every row in T must be examined to determine whether the value of C1 matches the given value.

Table space scans of nonsegmented table spaces

DB2 reads and examines every page in the table space, regardless of which table the page belongs to. It might also read pages that have been left as free space and space not yet reclaimed after deleting data.

Table space scans of segmented table spaces

If the table space is segmented, DB2 first determines which segments need to be read. It then reads only the segments in the table space that contain rows of T. If the prefetch quantity, which is determined by the size of your buffer pool, is greater than the SEGSIZE and if the segments for T are not contiguous, DB2 might read unnecessary pages. Use a SEGSIZE value that is as large as possible, consistent with the size of the data. A large SEGSIZE value is best to maintain clustering of data rows. For very small tables, specify a SEGSIZE value that is equal to the number of pages required for the table.

Recommendation for SEGSIZE value: Table 105 summarizes the recommendations for SEGSIZE, depending on how large the table is.

Table 105. Recommendations for SEGSIZE

Number of pages	SEGSIZE recommendation
≤ 28	4 to 28
> 28 < 128 pages	32
≥ 128 pages	64

Table space scans of partitioned table spaces

Partitioned table spaces are nonsegmented. A table space scan on a partitioned table space is more efficient than on a nonpartitioned table space. DB2 takes advantage of the partitions by a limited partition scan, as described under “Was a scan limited to certain partitions? (PAGE_RANGE=Y)” on page 806.

Table space scans and sequential prefetch

Regardless of the type of table space, DB2 plans to use sequential prefetch for a table space scan. For a segmented table space, DB2 might not actually use sequential prefetch at execution time if it can determine that fewer than four data pages need to be accessed. For guidance on monitoring sequential prefetch, see “Sequential prefetch (PREFETCH=S)” on page 830.

If you do not want to use sequential prefetch for a particular query, consider adding to it the clause OPTIMIZE FOR 1 ROW.

Index access paths

DB2 uses the following index access paths:

- “Matching index scan (MATCHCOLS>0)” on page 811
- “Index screening” on page 811
- “Nonmatching index scan (ACCESSTYPE=I and MATCHCOLS=0)” on page 812
- “IN-list index scan (ACCESSTYPE=N)” on page 812
- “Multiple index access (ACCESSTYPE is M, MX, MI, or MU)” on page 812
- “One-fetch access (ACCESSTYPE=I1)” on page 814
- “Index-only access (INDEXONLY=Y)” on page 814
- “Equal unique index (MATCHCOLS=number of index columns)” on page 814

Matching index scan (MATCHCOLS>0)

In a *matching index scan*, predicates are specified on either the leading or all of the index key columns. These predicates provide *filtering*; only specific index pages and data pages need to be accessed. If the degree of filtering is high, the matching index scan is efficient.

In the general case, the rules for determining the number of matching columns are simple, although there are a few exceptions.

- Look at the index columns from leading to trailing. For each index column, search for an indexable boolean term predicate on that column. (See “Properties of predicates” on page 735 for a definition of boolean term.) If such a predicate is found, then it can be used as a matching predicate.
Column MATCHCOLS in a plan table shows how many of the index columns are matched by predicates.
- If no matching predicate is found for a column, the search for matching predicates stops.
- If a matching predicate is a range predicate, then there can be no more matching columns. For example, in the matching index scan example that follows, the range predicate $C2 > 1$ prevents the search for additional matching columns.
- For star joins, a missing key predicate does not cause termination of matching columns that are to be used on the fact table index.

The exceptional cases are:

- At most one IN-list predicate can be a matching predicate on an index.
- For MX accesses and index access with list prefetch, IN-list predicates cannot be used as matching predicates.
- Join predicates cannot qualify as matching predicates when doing a merge join (METHOD=2). For example, $T1.C1 = T2.C1$ cannot be a matching predicate when doing a merge join, although any local predicates, such as $C1 = 5$ can be used.
Join predicates can be used as matching predicates on the inner table of a nested loop join or hybrid join.

Matching index scan example: Assume there is an index on $T(C1, C2, C3, C4)$:

```
SELECT * FROM T
  WHERE C1=1 AND C2>1
         AND C3=1;
```

Two matching columns occur in this example. The first one comes from the predicate $C1=1$, and the second one comes from $C2>1$. The range predicate on $C2$ prevents $C3$ from becoming a matching column.

Index screening

In *index screening*, predicates are specified on index key columns but are not part of the matching columns. Those predicates improve the index access by reducing the number of rows that qualify while searching the index. For example, with an index on $T(C1, C2, C3, C4)$ in the following SQL statement, $C3 > 0$ and $C4 = 2$ are index screening predicates.

```
SELECT * FROM T
  WHERE C1 = 1
         AND C3 > 0 AND C4 = 2
         AND C5 = 8;
```

The predicates can be applied on the index, but they are not matching predicates. C5=8 is not an index screening predicate, and it must be evaluated when data is retrieved. The value of MATCHCOLS in the plan table is 1.

EXPLAIN does not directly tell when an index is screened; however, if MATCHCOLS is less than the number of index key columns, it indicates that index screening is possible.

Nonmatching index scan (ACCESSTYPE=I and MATCHCOLS=0)

In a *nonmatching index scan* no matching columns are in the index. Hence, all the index keys must be examined.

Because a nonmatching index usually provides no filtering, only a few cases provide an efficient access path. The following situations are examples:

- When index screening predicates exist
In that case, not all of the data pages are accessed.
- When the clause OPTIMIZE FOR n ROWS is used
That clause can sometimes favor a nonmatching index, especially if the index gives the ordering of the ORDER BY clause or GROUP BY clause.
- When more than one table exists in a nonsegmented table space
In that case, a table space scan reads irrelevant rows. By accessing the rows through the nonmatching index, fewer rows are read.

IN-list index scan (ACCESSTYPE=N)

An *IN-list index scan* is a special case of the matching index scan, in which a single indexable IN predicate is used as a matching equal predicate.

You can regard the IN-list index scan as a series of matching index scans with the values in the IN predicate being used for each matching index scan. The following example has an index on (C1,C2,C3,C4) and might use an IN-list index scan:

```
SELECT * FROM T
WHERE C1=1 AND C2 IN (1,2,3)
AND C3>0 AND C4<100;
```

The plan table shows MATCHCOLS = 3 and ACCESSTYPE = N. The IN-list scan is performed as the following three matching index scans:

(C1=1,C2=1,C3>0), (C1=1,C2=2,C3>0), (C1=1,C2=3,C3>0)

Parallelism is supported for queries that involve IN-list index access. These queries used to run sequentially in previous releases of DB2, although parallelism could have been used when the IN-list access was for the inner table of a parallel group. Now, in environments in which parallelism is enabled, you can see a reduction in elapsed time for queries that involve IN-list index access for the outer table of a parallel group.

Multiple index access (ACCESSTYPE is M, MX, MI, or MU)

Multiple index access uses more than one index to access a table. It is a good access path when:

- No single index provides efficient access.
- A combination of index accesses provides efficient access.

RID lists are constructed for each of the indexes involved. The unions or intersections of the RID lists produce a final list of qualified RIDs that is used to retrieve the result rows, using list prefetch. You can consider multiple index access

as an extension to list prefetch with more complex RID retrieval operations in its first phase. The complex operators are union and intersection.

DB2 chooses multiple index access for the following query:

```
SELECT * FROM EMP
WHERE (AGE = 34) OR
      (AGE = 40 AND JOB = 'MANAGER');
```

For this query:

- EMP is a table with columns EMPNO, EMPNAME, DEPT, JOB, AGE, and SAL.
- EMPX1 is an index on EMP with key column AGE.
- EMPX2 is an index on EMP with key column JOB.

The plan table contains a sequence of rows describing the access. For this query, ACESSTYPE uses the following values:

Value Meaning

- M** Start of multiple index access processing
MX Indexes are to be scanned for later union or intersection
MI An intersection (AND) is performed
MU A union (OR) is performed

The following steps relate to the previous query and the values shown for the plan table in Table 106:

1. Index EMPX1, with matching predicate AGE= 34, provides a set of candidates for the result of the query. The value of MIXOPSEQ is 1.
2. Index EMPX1, with matching predicate AGE = 40, also provides a set of candidates for the result of the query. The value of MIXOPSEQ is 2.
3. The first intersection (AND) is done, and the value of MIXOPSEQ is 3. This MI removes the two previous candidate lists (produced by MIXOPSEQs 2 and 3) by intersecting them to form an intermediate candidate list, IR1, which is not shown in PLAN_TABLE.
4. Index EMPX2, with matching predicate JOB='MANAGER', also provides a set of candidates for the result of the query. The value of MIXOPSEQ is 4.
5. The last step, where the value MIXOPSEQ is 5, is a union (OR) of the two remaining candidate lists, which are IR1 and the candidate list produced by MIXOPSEQ 1. This final union gives the result for the query.

Table 106. Plan table output for a query that uses multiple indexes. Depending on the filter factors of the predicates, the access steps can appear in a different order.

PLAN-NO	TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	PREFETCH	MIXOP-SEQ
1	EMP	M	0		L	0
1	EMP	MX	1	EMPX1		1
1	EMP	MX	1	EMPX1		2
1	EMP	MI	0			3
1	EMP	MX	1	EMPX2		4
1	EMP	MU	0			5

In this example, the steps in the multiple index access follow the physical sequence of the predicates in the query. This is not always the case. The multiple index steps are arranged in an order that uses RID pool storage most efficiently and for the least amount of time.

One-fetch access (ACCESSTYPE=I1)

One-fetch index access requires retrieving only one row. It is the best possible access path and is chosen whenever it is available. It applies to a statement with a MIN or MAX aggregate function: the order of the index allows a single row to give the result of the function.

One-fetch index access is a possible access path when:

- The query includes only one table.
- The query includes only one aggregate function (either MIN or MAX).
- Either no predicate or all predicates are matching predicates for the index.
- The query includes no GROUP BY clause.
- Aggregate functions are on:
 - The first index column if no predicates exist
 - The last matching column of the index if the last matching predicate is a range type
 - The next index column (after the last matching column) if all matching predicates are equal type

Queries using one-fetch index access: Assuming that an index exists on T(C1,C2,C3), the following queries use one-fetch index scan:

```
SELECT MIN(C1) FROM T;  
SELECT MIN(C1) FROM T WHERE C1>5;  
SELECT MIN(C1) FROM T WHERE C1>5 AND C1<10;  
SELECT MIN(C2) FROM T WHERE C1=5;  
SELECT MAX(C1) FROM T;  
SELECT MAX(C2) FROM T WHERE C1=5 AND C2<10;  
SELECT MAX(C2) FROM T WHERE C1=5 AND C2>5 AND C2<10;  
SELECT MAX(C2) FROM T WHERE C1=5 AND C2 BETWEEN 5 AND 10;
```

Index-only access (INDEXONLY=Y)

With *index-only access*, the access path does not require any data pages because the access information is available in the index. Conversely, when an SQL statement requests a column that is not in the index, updates any column in the table, or deletes a row, DB2 has to access the associated data pages. Because the index is almost always smaller than the table itself, an index-only access path usually processes the data efficiently.

With an index on T(C1,C2), the following queries can use index-only access:

```
SELECT C1, C2 FROM T WHERE C1 > 0;  
SELECT C1, C2 FROM T;  
SELECT COUNT(*) FROM T WHERE C1 = 1;
```

Equal unique index (MATCHCOLS=number of index columns)

An index that is fully matched and unique, and in which all matching predicates are equal-predicates, is called an *equal unique index* case. This case guarantees that only one row is retrieved. If there is no one-fetch index access available, this is considered the most efficient access over all other indexes that are not equal unique. (The uniqueness of an index is determined by whether or not it was defined as unique.)

Sometimes DB2 can determine that an index that is not fully matching is actually an equal unique index case. Assume the following case:

Unique Index1: (C1, C2)
Unique Index2: (C2, C1, C3)

```
SELECT C3 FROM T
WHERE C1 = 1 AND C2 = 5;
```

Index1 is a fully matching equal unique index. However, Index2 is also an equal unique index even though it is not fully matching. Index2 is the better choice because, in addition to being equal and unique, it also provides index-only access.

UPDATE using an index

If no index key columns are updated, you can use an index while performing an UPDATE operation.

To use a matching index scan to update an index in which its key columns are being updated, the following conditions must be met:

- Each updated key column must have a corresponding predicate of the form "index_key_column = constant" or "index_key_column IS NULL".
- If a view is involved, WITH CHECK OPTION must not be specified.

For updates that do not involve dynamic scrollable cursors, DB2 can use list prefetch, multiple index access, or IN-list access. With list prefetch or multiple index access, any index or indexes can be used in an UPDATE operation. Of course, to be chosen, those access paths must provide efficient access to the data.

A positioned update that uses a dynamic scrollable cursor cannot use an access path with list prefetch, or multiple index access. This means that indexes that do not meet the preceding criteria cannot be used to locate the rows to be updated.

Interpreting access to two or more tables (join)

A *join* operation retrieves rows from more than one table and combines them. The operation specifies at least two tables, but they need not be distinct.

This section begins with "Definitions and examples of join operations" and continues with descriptions of the methods of joining that can be indicated in a plan table:

- "Nested loop join (METHOD=1)" on page 818
- "Merge scan join (METHOD=2)" on page 820
- "Hybrid join (METHOD=4)" on page 821
- "Star join (JOIN_TYPE='S')" on page 823

Definitions and examples of join operations

This section contains definitions and examples that are related to join operations.

Definitions: A *composite table* represents the result of accessing one or more tables in a query. If a query contains a single table, only one composite table exists. If one or more joins are involved, an *outer composite table* consists of the intermediate result rows from the previous join step. This intermediate result may, or may not, be materialized into a work file.

The *new table* (or *inner table*) in a join operation is the table that is newly accessed in the step.

A join operation can involve more than two tables. In these cases, the operation is carried out in a series of steps. For non-star joins, each step joins only two tables.

Example: Figure 231 shows a two-step join operation.

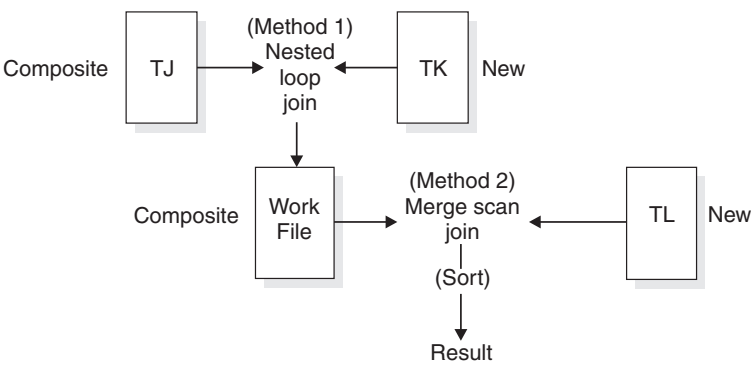


Figure 231. Two-step join operation

DB2 performs the following steps to complete the join operation:

1. Accesses the first table (METHOD=0), named TJ (TNAME), which becomes the composite table in step 2.
2. Joins the new table TK to TJ, forming a new composite table.
3. Sorts the new table TL (SORTN_JOIN=Y) and the composite table (SORTC_JOIN=Y), and then joins the two sorted tables.
4. Sorts the final composite table (TNAME is blank) into the desired order (SORTC_ORDERBY=Y).

Figure 231

Table 107 and Table 108 show a subset of columns in a plan table for this join operation.

Table 107. Subset of columns for a two-step join operation

METHOD	TNAME	ACCESS- TYPE	MATCH- COLS	ACCESS- NAME	INDEX- ONLY	TSLOCK- MODE
0	TJ	I	1	TJX1	N	IS
1	TK	I	1	TKX1	N	IS
2	TL	I	0	TLX1	Y	S
3			0		N	

Table 108. Subset of columns for a two-step join operation

SORTN UNIQ	SORTN JOIN	SORTN ORDERBY	SORTN GROUPBY	SORTC UNIQ	SORTC JOIN	SORTC ORDERBY	SORTC GROUPBY
N	N	N	N	N	N	N	N
N	N	N	N	N	N	N	N
N	Y	N	N	N	Y	N	N
N	N	N	N	N	N	Y	N

Definitions: A join operation typically matches a row of one table with a row of another on the basis of a *join condition*. For example, the condition might specify that the value in column A of one table equals the value of column X in the other table (WHERE T1.A = T2.X).

Two kinds of joins differ in what they do with rows in one table that do not match on the join condition with any row in the other table:

- An *inner join* discards rows of either table that do not match any row of the other table.
- An *outer join* keeps unmatched rows of one or the other table, or of both. A row in the composite table that results from an unmatched row is filled out with null values. As Table 109 shows, outer joins are distinguished by which unmatched rows they keep.

Table 109. Join types and kept unmatched rows

Outer join type	Included unmatched rows
Left outer join	The composite (outer) table
Right outer join	The new (inner) table
Full outer join	Both tables

Example: Suppose that you issue the following statement to explain an outer join:

```
EXPLAIN PLAN SET QUERYNO = 10 FOR
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
```

Table 110 shows a subset of the plan table for the outer join.

Table 110. Plan table output for an example with outer joins

QUERYNO	QBLOCKNO	PLANNO	TNAME	JOIN_TYPE
10	1	1	PROJECTS	
10	1	2	TEMP	L
10	2	1	PRODUCTS	
10	2	2	PARTS	F

Column JOIN_TYPE identifies the type of outer join with one of these values:

- F for FULL OUTER JOIN
- L for LEFT OUTER JOIN
- Blank for INNER JOIN or no join

At execution, DB2 converts every right outer join to a left outer join; thus JOIN_TYPE never identifies a right outer join specifically.

Materialization with outer join: Sometimes DB2 has to materialize a result table when an outer join is used in conjunction with other joins, views, or nested table expressions. You can tell when this happens by looking at the TABLE_TYPE and TNAME columns of the plan table. When materialization occurs, TABLE_TYPE

contains a W, and TNAME shows the name of the materialized table as DSNWFQB(xx), where xx is the number of the query block (QBLOCKNO) that produced the work file.

Nested loop join (METHOD=1)

This section describes nested loop join, which is common join method. Figure 232 illustrates a nested loop join.

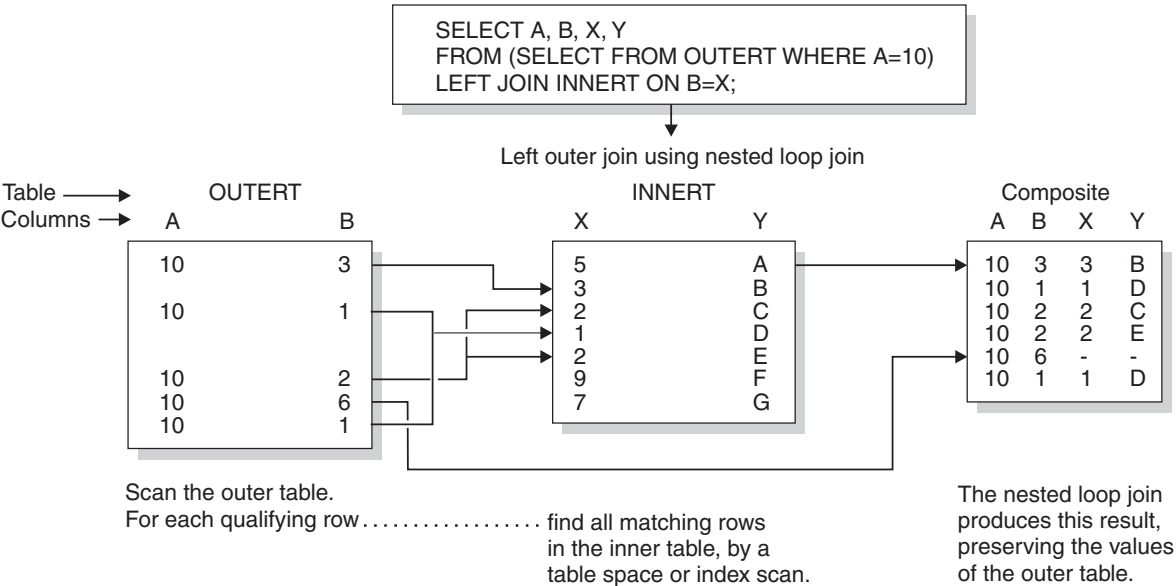


Figure 232. Nested loop join for a left outer join

Method of joining

DB2 scans the composite (outer) table. For each row in that table that qualifies (by satisfying the predicates on that table), DB2 searches for matching rows of the new (inner) table. It concatenates any it finds with the current row of the composite table. If no rows match the current row, then:

- For an inner join, DB2 discards the current row.
- For an outer join, DB2 concatenates a row of null values.

Stage 1 and stage 2 predicates eliminate unqualified rows during the join. (For an explanation of those types of predicate, see “Stage 1 and stage 2 predicates” on page 737.) DB2 can scan either table using any of the available access methods, including table space scan.

Performance considerations

The nested loop join repetitively scans the inner table. That is, DB2 scans the outer table once, and scans the inner table as many times as the number of qualifying rows in the outer table. Therefore, the nested loop join is usually the most efficient join method when the values of the join column passed to the inner table are in sequence and the index on the join column of the inner table is clustered, or the number of rows retrieved in the inner table through the index is small.

When nested loop join is used

Nested loop join is often used if:

- The outer table is small.

- Predicates with small filter factors reduce the number of qualifying rows in the outer table.
- An efficient, highly clustered index exists on the join columns of the inner table.
- The number of data pages accessed in the inner table is small.
- No join columns exist. Hybrid and sort merge joins require join columns; nested loop joins do not.

Example: left outer join: Figure 232 on page 818 illustrates a nested loop for a left outer join. The outer join preserves the unmatched row in OUTERT with values A=10 and B=6. The same join method for an inner join differs only in discarding that row.

Example: one-row table priority: For a case like the following example, with a unique index on T1.C2, DB2 detects that T1 has only one row that satisfies the search condition. DB2 makes T1 the first table in a nested loop join.

```
SELECT * FROM T1, T2
  WHERE T1.C1 = T2.C1 AND
        T1.C2 = 5;
```

Example: Cartesian join with small tables first: A *Cartesian join* is a form of nested loop join in which there are no join predicates between the two tables. DB2 usually avoids a Cartesian join, but sometimes it is the most efficient method, as in the following example. The query uses three tables: T1 has 2 rows, T2 has 3 rows, and T3 has 10 million rows.

```
SELECT * FROM T1, T2, T3
  WHERE T1.C1 = T3.C1 AND
        T2.C2 = T3.C2 AND
        T3.C3 = 5;
```

Join predicates are between T1 and T3 and between T2 and T3. There is no join predicate between T1 and T2.

Assume that 5 million rows of T3 have the value C3=5. Processing time is large if T3 is the outer table of the join and tables T1 and T2 are accessed for each of 5 million rows.

However if all rows from T1 and T2 are joined, without a join predicate, the 5 million rows are accessed only six times, once for each row in the Cartesian join of T1 and T2. It is difficult to say which access path is the most efficient. DB2 evaluates the different options and could decide to access the tables in the sequence T1, T2, T3.

Sorting the composite table: Your plan table could show a nested loop join that includes a sort on the composite table. DB2 might sort the composite table (the outer table in Figure 232) if the following conditions exist:

- The join columns in the composite table and the new table are not in the same sequence.
- The join column of the composite table has no index.
- The index is poorly clustered.

Nested loop join with a sorted composite table has the following performance advantages:

- Uses sequential detection efficiently to prefetch data pages of the new table, reducing the number of synchronous I/O operations and the elapsed time.

- Avoids repetitive full probes of the inner table index by using the index look-aside.

Merge scan join (METHOD=2)

Merge scan join is also known as *merge join* or *sort merge join*. For this method, there must be one or more predicates of the form `TABLE1.COL1=TABLE2.COL2`, where the two columns have the same data type and length attribute.

Method of joining

Figure 233 illustrates a merge scan join.

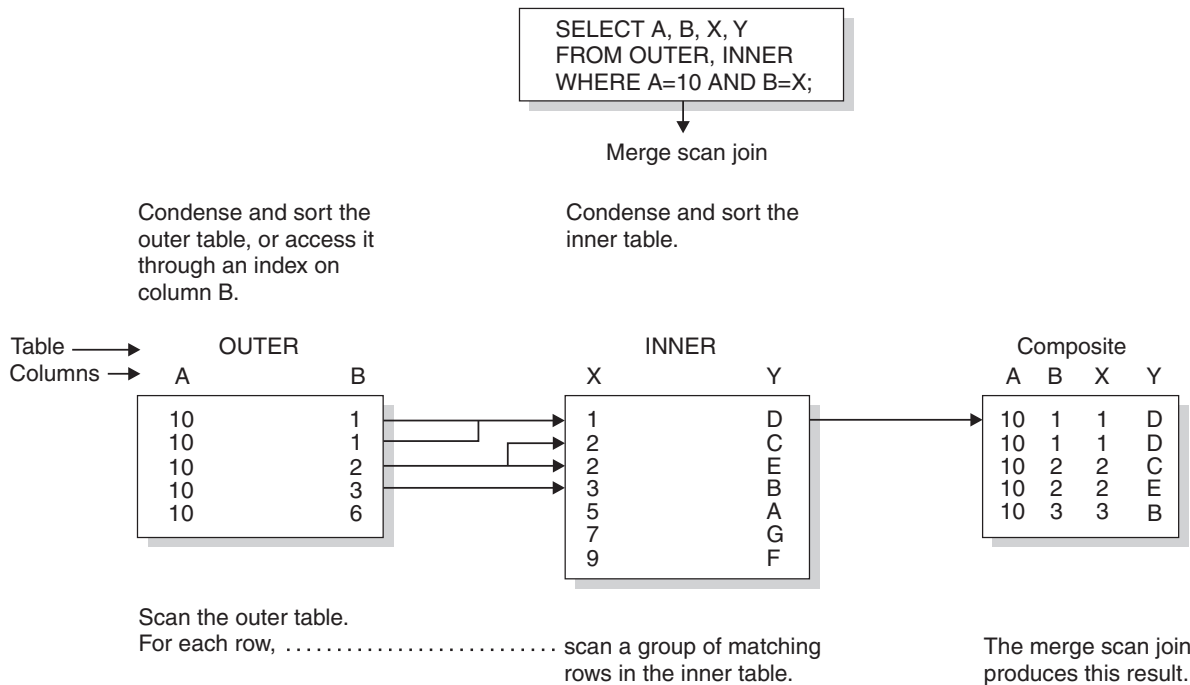


Figure 233. Merge scan join

DB2 scans both tables in the order of the join columns. If no efficient indexes on the join columns provide the order, DB2 might sort the outer table, the inner table, or both. The inner table is put into a work file; the outer table is put into a work file only if it must be sorted. When a row of the outer table matches a row of the inner table, DB2 returns the combined rows.

DB2 then reads another row of the inner table that might match the same row of the outer table and continues reading rows of the inner table as long as there is a match. When there is no longer a match, DB2 reads another row of the outer table.

- If that row has the same value in the join column, DB2 reads again the matching group of records from the inner table. Thus, a group of duplicate records in the inner table is scanned as many times as there are matching records in the outer table.
- If the outer row has a new value in the join column, DB2 searches ahead in the inner table. It can find any of the following rows:
 - Unmatched rows in the inner table, with lower values in the join column.
 - A new matching inner row. DB2 then starts the process again.

- An inner row with a higher value of the join column. Now the row of the outer table is unmatched. DB2 searches ahead in the outer table, and can find any of the following rows:
 - Unmatched rows in the outer table.
 - A new matching outer row. DB2 then starts the process again.
 - An outer row with a higher value of the join column. Now the row of the inner table is unmatched, and DB2 resumes searching the inner table.

If DB2 finds an unmatched row:

For an inner join, DB2 discards the row.

For a left outer join, DB2 discards the row if it comes from the inner table and keeps it if it comes from the outer table.

For a full outer join, DB2 keeps the row.

When DB2 keeps an unmatched row from a table, it concatenates a set of null values as if that matched from the other table. A merge scan join must be used for a full outer join.

Performance considerations

A full outer join by this method uses all predicates in the ON clause to match the two tables and reads every row at the time of the join. Inner and left outer joins use only stage 1 predicates in the ON clause to match the tables. If your tables match on more than one column, it is generally more efficient to put all the predicates for the matches in the ON clause, rather than to leave some of them in the WHERE clause.

For an inner join, DB2 can derive extra predicates for the inner table at bind time and apply them to the sorted outer table to be used at run time. The predicates can reduce the size of the work file needed for the inner table.

If DB2 has used an efficient index on the join columns, to retrieve the rows of the inner table, those rows are already in sequence. DB2 puts the data directly into the work file without sorting the inner table, which reduces the elapsed time.

When merge scan join is used

A merge scan join is often used if:

- The qualifying rows of the inner and outer table are large, and the join predicate does not provide much filtering; that is, in a many-to-many join.
- The tables are large and have no indexes with matching columns.
- Few columns are selected on inner tables. This is the case when a DB2 sort is used. The fewer the columns to be sorted, the more efficient the sort is.

Hybrid join (METHOD=4)

The method applies only to an inner join and requires an index on the join column of the inner table. Figure 234 on page 822 illustrates a hybrid join.

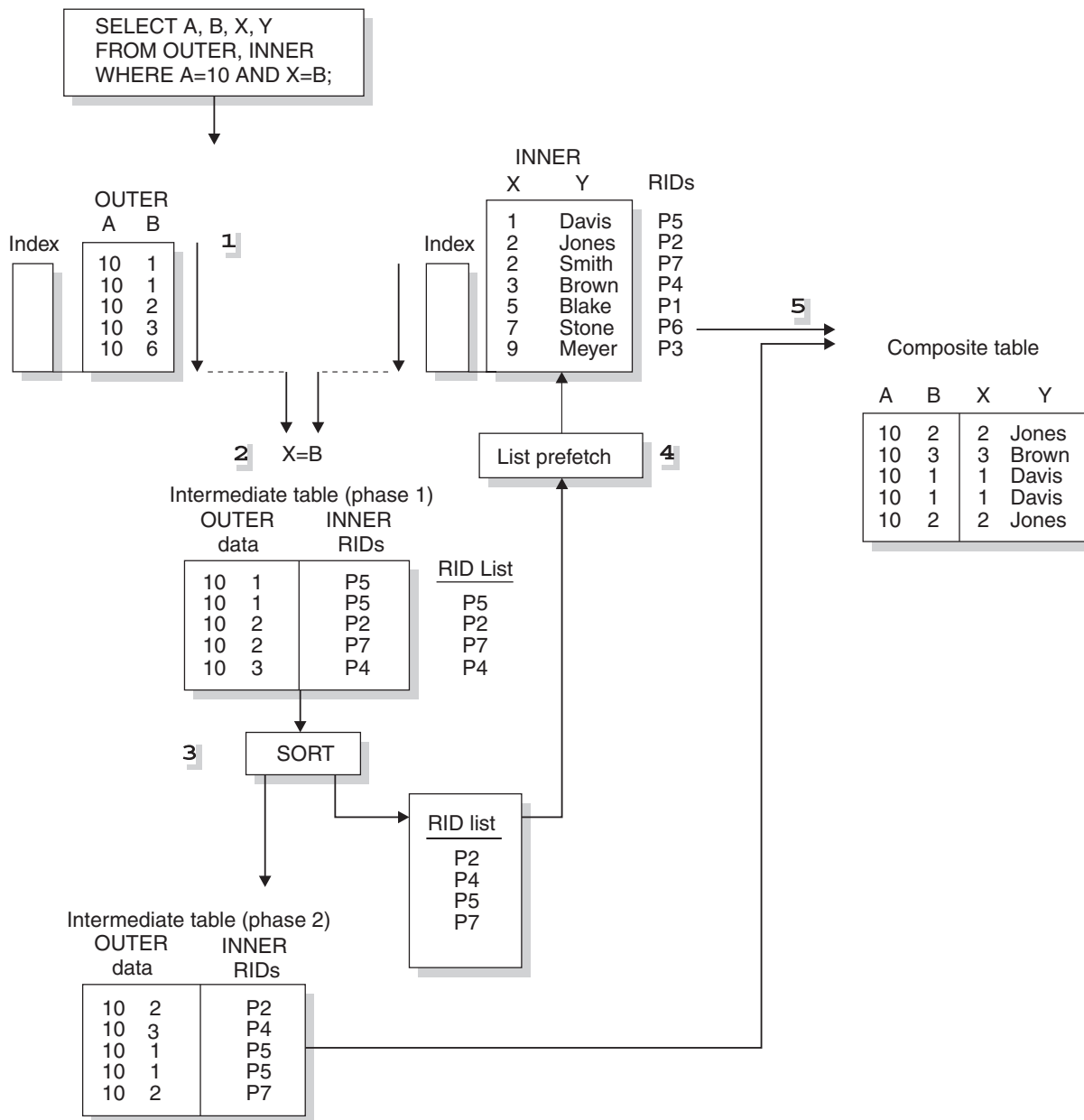


Figure 234. Hybrid join (SORTN_JOIN='Y')

Method of joining

The method requires obtaining RIDs in the order needed to use list prefetch. The steps are shown in Figure 234. In that example, both the outer table (OUTER) and the inner table (INNER) have indexes on the join columns.

DB2 performs the following steps:

- 1** Scans the outer table (OUTER).
- 2** Joins the outer table with RIDs from the index on the inner table. The result is the phase 1 intermediate table. The index of the inner table is scanned for every row of the outer table.
- 3** Sorts the data in the outer table and the RIDs, creating a sorted RID list

and the phase 2 intermediate table. The sort is indicated by a value of Y in column SORTN_JOIN of the plan table. If the index on the inner table is a well-clustered index, DB2 can skip this sort; the value in SORTN_JOIN is then N.

- 4** Retrieves the data from the inner table, using list prefetch.
- 5** Concatenates the data from the inner table and the phase 2 intermediate table to create the final composite table.

Possible results from EXPLAIN for hybrid join

Table 117 shows possible EXPLAIN results from a hybrid join and an explanation of each column value.

Table 117. Explanation of EXPLAIN results for a hybrid join

Column value	Explanation
METHOD='4'	A hybrid join was used.
SORTC_JOIN='Y'	The composite table was sorted.
SORTN_JOIN='Y'	The intermediate table was sorted in the order of inner table RIDs. A non-clustered index accessed the inner table RIDs.
SORTN_JOIN='N'	The intermediate table RIDs were not sorted. A clustered index retrieved the inner table RIDs, and the RIDs were already well ordered.
PREFETCH='L'	Pages were read using list prefetch.

Performance considerations

Hybrid join uses list prefetch more efficiently than nested loop join, especially if there are indexes on the join predicate with low cluster ratios. It also processes duplicates more efficiently because the inner table is scanned only once for each set of duplicate values in the join column of the outer table.

If the index on the inner table is highly clustered, there is no need to sort the intermediate table (SORTN_JOIN=N). The intermediate table is placed in a table in memory rather than in a work file.

When hybrid join is used

Hybrid join is often used if:

- A nonclustered index or indexes are used on the join columns of the inner table.
- The outer table has duplicate qualifying rows.

Star join (JOIN_TYPE='S')

Star join is a special join technique that DB2 uses to efficiently join tables that form a star schema. A star schema is a logical database design that is included in decision support applications. A star schema is composed of a fact table and a number of dimension tables that are connected to it. A dimension table contains several values that are given an ID, which is used in the fact table instead of all the values.

You can think of the fact table, which is much larger than the dimension tables, as being in the center surrounded by dimension tables; the result resembles a star formation. Figure 235 on page 824 illustrates the star formation.

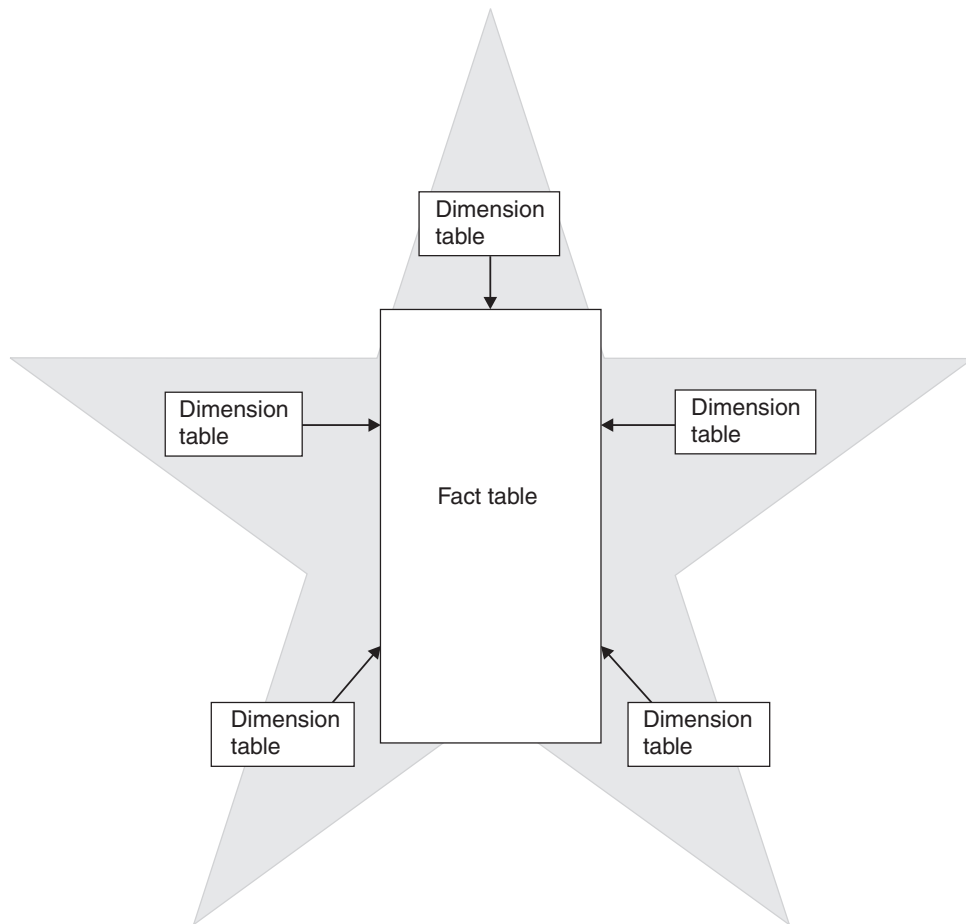


Figure 235. Star schema with a fact table and dimension tables

Unlike the steps in the other join methods (nested loop join, merge scan join, and hybrid join) in which only two tables are joined in each step, a step in the star join method can involve three or more tables. Dimension tables are joined to the fact table via a multi-column index that is defined on the fact table. Therefore, having a well-defined, multi-column index on the fact table is critical for efficient star join processing.

Example of a star schema

For an example of a star schema consider the following scenario. A star schema is composed of a fact table for sales, with dimension tables connected to it for time, products, and geographic locations. The time table has an ID for each month, its quarter, and the year. The product table has an ID for each product item and its class and its inventory. The geographic location table has an ID for each city and its country.

In this scenario, the sales table contains three columns with IDs from the dimension tables for time, product, and location instead of three columns for time, three columns for products, and two columns for location. Thus, the size of the fact table is greatly reduced. In addition, if you needed to change an item, you would do it once in a dimension table instead of several times for each instance of the item in the fact table.

You can create even more complex star schemas by normalizing a dimension table into several tables. The normalized dimension table is called a *snowflake*. Only one of the tables in the snowflake joins directly with the fact table.

When star join is used

To access the data in a star schema, you often write SELECT statements that include join operations between the fact table and the dimension tables, but no join operations between dimension tables. DB2 uses star join processing as the join type for the query if the following conditions are true:

- The query references at least two dimensions.
- All join predicates are between the fact table and the dimension tables, or within tables of the same snowflake. If a snowflake is connected to the fact table, only one table in the snowflake (the central dimension table) can be joined to the fact table.
- All join predicates between the fact table and dimension tables are equi-join predicates.
- All join predicates between the fact table and dimension tables are Boolean term predicates. For more information, see “Boolean term (BT) predicates” on page 738.
- There are no instances of predicates that consist of a local predicate on a dimension table and a local predicate on a different table that are connected with an OR logical operator.
- No correlated subqueries cross dimensions.
- No single fact table column is joined to columns of different dimension tables in join predicates. For example, fact table column F1 cannot be joined to column D1 of dimension table T1 and also joined to column D2 of dimension table T2.
- After DB2 simplifies join operations, no outer join operations exist. For more information, see “When DB2 simplifies join operations” on page 756.
- The data type and length of both sides of a join predicate are the same.
- The value of subsystem parameter STARJOIN is 1, or the cardinality of the fact table to the largest dimension table meets the requirements specified by the value of the subsystem parameter. The values of STARJOIN and cardinality requirements are:

-1 Star join is disabled. This is the default.

1 Star join is enabled. The one table with the largest cardinality is the fact table. However, if there is more than one table with this cardinality, star join is not enabled.

0 Star join is enabled if the cardinality of the fact table is at least 25 times the cardinality of the largest dimension that is a base table that is joined to the fact table.

n Star join is enabled if the cardinality of the fact table is at least *n* times the cardinality of the largest dimension that is a base table that is joined to the fact table, where $2 \leq n \leq 32768$.

You can set the subsystem parameter STARJOIN by using the STAR JOIN QUERIES field on the DSNTIP8 installation panel.

- The number of tables in the star schema query block, including the fact table, dimensions tables, and snowflake tables, meet the requirements that are specified by the value of subsystem parameter SJTABLES. The value of SJTABLES is considered only if the subsystem parameter STARJOIN qualifies the query for star join. The values of SJTABLES are:

1, 2, or 3

Star join is always considered.

4 to 255

Star join is considered if the query block has at least the specified number of tables. If star join is enabled, 10 is the default value.

256 and greater

Star join will never be considered.

Star join, which can reduce bind time significantly, does not provide optimal performance in all cases. Performance of star join depends on a number of factors such as the available indexes on the fact table, the cluster ratio of the indexes, and the selectivity of rows through local and join predicates. Follow these general guidelines for setting the value of SJTABLES:

- If you have queries that reference less than 10 tables in a star schema database and you want to make the star join method applicable to all qualified queries, set the value of SJTABLES to the minimum number of tables used in queries that you want to be considered for star join.
Example: Suppose that you query a star schema database that has one fact table and three dimension tables. You should set SJTABLES to 4.
- If you want to use star join for relatively large queries that reference a star schema database but are not necessarily suitable for star join, use the default. The star join method will be considered for all qualified queries that have 10 or more tables.
- If you have queries that reference a star schema database but, in general, do not want to use star join, consider setting SJTABLES to a higher number, such as 15, if you want to drastically cut the bind time for large queries and avoid a potential bind time SQL return code -101 for large qualified queries.

For recommendations on indexes for star schemas, see “Creating indexes for efficient star-join processing” on page 782.

Examples: query with three dimension tables: Suppose that you have a store in San Jose and want information about sales of audio equipment from that store in 2000. For this example, you want to join the following tables:

- A fact table for SALES (S)
- A dimension table for TIME (T) with columns for an ID, month, quarter, and year
- A dimension table for geographic LOCATION (L) with columns for an ID, city, region, and country
- A dimension table for PRODUCT (P) with columns for an ID, product item, class, and inventory

You could write the following query to join the tables:

```
SELECT *
FROM SALES S, TIME T, PRODUCT P, LOCATION L
WHERE S.TIME = T.ID      AND
      S.PRODUCT = P.ID    AND
      S.LOCATION = L.ID   AND
      T.YEAR = 2000       AND
      P.CLASS = 'SAN JOSE';
```

You would use the following index:

```
CREATE INDEX XSALES_TPL ON SALES (TIME, PRODUCT, LOCATION);
```

You EXPLAIN output looks like Table 118 on page 827.

Table 118. Plan table output for a star join example with *TIME*, *PRODUCT*, and *LOCATION*

QUERYNO	QBLOCKNO	METHOD	TNAME	JOIN TYPE	SORTN JOIN	ACCESS TYPE
1	1	0	TIME	S	Y	R
1	1	1	PRODUCT	S	Y	R
1	1	1	LOCATION	S	Y	R
1	1	1	SALES	S		I

All snowflakes are processed before the central part of the star join, as individual query blocks, and are materialized into work files. There is a work file for each snowflake. The EXPLAIN output identifies these work files by naming them DSN_DIM_TBLX(*nn*), where *nn* indicates the corresponding QBLOCKNO for the snowflake.

This next example shows the plan for a star join that contains two snowflakes. Suppose that two new tables MANUFACTURER (M) and COUNTRY (C) are added to the tables in the previous example to break dimension tables PRODUCT (P) and LOCATION (L) into snowflakes:

- The PRODUCT table has a new column MID that represents the manufacturer.
- Table MANUFACTURER (M) has columns for MID and name to contain manufacturer information.
- The LOCATION table has a new column CID that represents the country.
- Table COUNTRY (C) has columns for CID and name to contain country information.

You could write the following query to join all the tables:

```
SELECT *
FROM SALES S, TIME T, PRODUCT P, MANUFACTURER M,
     LOCATION L, COUNTRY C
WHERE S.TIME = T.ID      AND
      S.PRODUCT = P.ID   AND
      P.MID = M.MID      AND
      S.LOCATION = L.ID  AND
      L.CID = C.CID      AND
      T.YEAR = 2000      AND
      M.NAME = 'some_company';
```

The joined table pairs (PRODUCT, MANUFACTURER) and (LOCATION, COUNTRY) are snowflakes. The EXPLAIN output of this query looks like Table 119.

Table 119. Plan table output for a star join example with snowflakes

QUERYNO	QBLOCKNO	METHOD	TNAME	JOIN TYPE	SORTN JOIN	ACCESS TYPE
1	1	0	TIME	S	Y	R
1	1	1	DSN_DIM_TBLX(02)	S	Y	R
1	1	1	SALES	S		I
1	1	1	DSN_DIM_TBLX(03)		Y	T
1	2	0	PRODUCT			R
1	2	1	MANUFACTURER			I
1	3	0	LOCATION			R

Table 119. Plan table output for a star join example with snowflakes (continued)

QUERYNO	QBLOCKNO	METHOD	TNAME	JOIN TYPE	SORTN JOIN	ACCESS TYPE
1	3	4	COUNTRY			I
Note: This query consists of three query blocks: • QBLOCKNO=1: The main star join block • QBLOCKNO=2: A snowflake (PRODUCT, MANUFACTURER) that is materialized into work file DSN_DIM_TBLX(02) • QBLOCKNO=3: A snowflake (LOCATION, COUNTRY) that is materialized into work file DSN_DIM_TBLX(03)						

The joins in the snowflakes are processed first, and each snowflake is materialized into a work file. Therefore, when the main star join block (QBLOCKNO=1) is processed, it contains four tables: SALES (the fact table), TIME (a base dimension table), and the two snowflake work files.

In this example, in the main star join block, the star join method is used for the first three tables (as indicated by S in the JOIN TYPE column of the plan table) and the remaining work file is joined by the nested loop join with sparse index access on the work file (as indicated by T in the ACCESTYPE column for DSN_DIM_TBLX(3)).

Dedicated virtual memory pool for star join operations

You can create a dedicated virtual memory pool for star join operations. When the virtual memory pool is enabled for star joins, DB2 caches data from work files that are used by star join queries. A virtual memory pool dedicated to star join operations has the following advantages:

- Immediate data availability. During a star join operation, work files might be scanned many times. If the work-file data is cached in the dedicated virtual memory pool, that data is immediately available for join operations.
- Reduced buffer pool contention. Because the dedicated virtual memory pool caches data separately from the work-file buffer pool, contention with the buffer pool is reduced. Reduced contention improves performance particularly when sort operations are performed concurrently.

To determine the size of the virtual memory pool, perform the following steps:

1. Determine the value of A. Estimate the number of star join queries that run concurrently.
2. Determine the value of B. Estimate the average number of work files that a star join query uses. In typical cases, with highly normalized star schemas, the average number is about three to six work files.
3. Determine the value of C. Estimate the number of work-file rows, the maximum length of the key, and the total of the maximum length of the relevant columns. Multiply these three values together to find the size of the data caching space for the work file, or the value of C.
4. Multiply (A) * (B) * (C) to determine the size of the pool in MB.

The default virtual memory pool size is 20 MB. To set the pool size, use the SJMXPOOL parameter on the DSNTIP8 installation panel.

Example: The following example shows how to determine the size of the virtual memory pool. Suppose that you issue the following star join query, where SALES is the fact table:

```

SELECT C.COUNTRY, P.PRDNAM, SUM(F.SPRICE)
FROM SALES F, TIME T, PROD P, LOC L, SCOUN C
WHERE F.TID = T.TID AND
      F.PID = P.PID AND
      F.LID = L.LID AND
      L.CID = C.CID AND
      P.PCODE IN (4, 7, 21, 22, 53)
GROUP BY .COUNTRY, P.PRDNAM;

```

The EXPLAIN output of this query looks like Table 120.

Table 120. EXPLAIN output for a star join query

QBLOCKNO	PLANNO	TNAME	METHOD	JOIN_ TYPE	ACCESS TYPE	ACCESS NAME
1	1	TIME	0	S	R	
1	2	PROD	1	S	T	
1	3	SALES	1	S	I	XSALES
1	4	DSN_DIM_TBLX(02)	1		T	
1	5		3			
2	1	LOC	0		R	
2	2	SCOUN	4		I	XSCOUN

For this query, two work files can be cached in memory. These work files, PROD and DSN_DIM_TBLX(02), are indicated by ACESSTYPE=T.

To determine the size of the dedicated virtual memory pool, perform the following steps:

1. Determine the value of A. Estimate the number of star join queries that run concurrently.
In this example, based on the type of operation, up to 12 star join queries are expected run concurrently. Therefore, A = 12.
2. Determine the value of B. Estimate the average number of work files that a star join query uses.
In this example, the star join query uses two work files, PROD and DSN_DIM_TBLX(02). Therefore B = 2.
3. Determine the value of C. Estimate the number of work-file rows, the maximum length of the key, and the total of the maximum length of the relevant columns. Multiply these three values together to find the size of the data caching space for the work file, or the value of C.
Both PROD and DSN_DIM_TBLX(02) are used to determine the value of C.
Recommendation: Average the values for a representative sample of work files, and round the value up to determine an estimate for a value of C.
 - The number of work-file rows depends on the number of rows that match the predicate. For PROD, 87 rows are stored in the work file because 87 rows match the IN-list predicate. No selective predicate is used for DSN_DIM_TBLX(02), so the entire result of the join is stored in the work file. The work file for DSN_DIM_TBLX(02) holds 2800 rows.
 - The maximum length of the key depends on the data type definition of the table's key column. For PID, the key column for PROD, the maximum length is 4. DSN_DIM_TBLX(02) is a work file that results from the join of LOC and SCOUN. The key column that is used in the join is LID from the LOC table. The maximum length of LID is 4.

- The maximum data length depends on the maximum length of the key column and the maximum length of the column that is selected as part of the star join. Add to the maximum data length 1 byte for nullable columns, 2 bytes for varying length columns, and 3 bytes for nullable and varying length columns.

For the PROD work file, the maximum data length is the maximum length of PID, which is 4, plus the maximum length of PRDNAME, which is 24.

Therefore, the maximum data length for the PROD work file is 28. For the DSN_DIM_TBLX(02) workfile, the maximum data length is the maximum length of LID, which is 4, plus the maximum length of COUNTRY, which is 36. Therefore, the maximum data length for the DSN_DIM_TBLX(02) work file is 40.

For PROD, $C = (87) * (4 + 28) = 2784$ bytes. For DSN_DIM_TBLX(02), $C = (2800) * (4 + 40) = 123200$ bytes.

The average of these two estimated values for C is approximately 62 KB. Because the number of rows in each work file can vary depending on the selection criteria in the predicate, the value of C should be rounded up to the nearest multiple of 100 KB. Therefore $C = 100$ KB.

4. Multiply $(A) * (B) * (C)$ to determine the size of the pool in MB.

The size of the pool is determined by multiplying $(12) * (2) * (100\text{KB}) = 2.4$ MB.

Interpreting data prefetch

Prefetch is a mechanism for reading a set of pages, usually 32, into the buffer pool with only one asynchronous I/O operation. Prefetch can allow substantial savings in both processor cycles and I/O costs. To achieve those savings, monitor the use of prefetch.

A plan table can indicate the use of three kinds of prefetch:

- “Sequential prefetch (PREFETCH=S)”
- “Dynamic prefetch (PREFETCH=D)” on page 831
- “List prefetch (PREFETCH=L)” on page 831

Additionally, you can choose not to use prefetch.

If DB2 does not choose prefetch at bind time, it can sometimes use it at execution time nevertheless. The method is described in “Sequential detection at execution time” on page 832.

Sequential prefetch (PREFETCH=S)

Sequential prefetch reads a sequential set of pages. The maximum number of pages read by a request issued from your application program is determined by the size of the buffer pool used. For each buffer pool size (4 KB, 8 KB, 16 KB, and 32 KB), Table 121 shows the number pages read by prefetch for each asynchronous I/O.

Table 121. The number of pages read by prefetch, by buffer pool size

Buffer pool size	Number of buffers	Pages read by prefetch (for each asynchronous I/O)
4 KB	<=223 buffers	8 pages
	224-999 buffers	16 pages
	1000+ buffers	32 pages

Table 121. The number of pages read by prefetch, by buffer pool size (continued)

Buffer pool size	Number of buffers	Pages read by prefetch (for each asynchronous I/O)
8 KB	<=112 buffers	4 pages
	113-499 buffers	8 pages
	500+ buffers	16 pages
16 KB	<=56 buffers	2 pages
	57-249 buffers	4 pages
	250+ buffers	8 pages
32 KB	<=16 buffers	0 pages (prefetch disabled)
	17-99 buffers	2 pages
	100+ buffers	4 pages

For certain utilities (LOAD, REORG, RECOVER), the prefetch quantity can be twice as much.

When sequential prefetch is used: Sequential prefetch is generally used for a table space scan.

For an index scan that accesses eight or more consecutive data pages, DB2 requests sequential prefetch at bind time. The index must have a cluster ratio of 80% or higher. Both data pages and index pages are prefetched.

Dynamic prefetch (PREFETCH=D)

Dynamic prefetch can reduce paging and improve performance over sequential prefetch for some data access that involves data that is not on consecutive pages. When DB2 expects that *dynamic prefetch* will be used, DB2 sets PREFETCH=D. At runtime, dynamic prefetch might or might not actually be used. However, DB2 expects dynamic prefetch and optimizes for that behavior.

When dynamic prefetch is used: Dynamic prefetch is used in prefetch situations when the pages that DB2 will access are distributed in a nonconsecutive manner. If the pages are distributed in a sufficiently consecutive manner, sequential prefetch is used instead.

List prefetch (PREFETCH=L)

List prefetch reads a set of data pages determined by a list of RIDs taken from an index. The data pages need not be contiguous. The maximum number of pages that can be retrieved in a single list prefetch is 32 (64 for utilities).

List prefetch can be used in conjunction with either single or multiple index access.

The access method

List prefetch uses the following three steps:

1. RID retrieval: A list of RIDs for needed data pages is found by matching index scans of one or more indexes.
2. RID sort: The list of RIDs is sorted in ascending order by page number.
3. Data retrieval: The needed data pages are prefetched in order using the sorted RID list.

List prefetch does not preserve the data ordering given by the index. Because the RIDs are sorted in page number order before accessing the data, the data is not retrieved in order by any column. If the data must be ordered for an ORDER BY clause or any other reason, it requires an additional sort.

In a hybrid join, if the index is highly clustered, the page numbers might not be sorted before accessing the data.

List prefetch can be used with most matching predicates for an index scan. IN-list predicates are the exception; they cannot be the matching predicates when list prefetch is used.

When list prefetch is used

List prefetch is used:

- Usually with a single index that has a cluster ratio lower than 80%
- Sometimes on indexes with a high cluster ratio, if the estimated amount of data to be accessed is too small to make sequential prefetch efficient, but large enough to require more than one regular read
- Always to access data by multiple index access
- Always to access data from the inner table during a hybrid join
- Usually for updatable cursors when the index contains columns that might be updated.

Bind time and execution time thresholds

DB2 does not consider list prefetch if the estimated number of RIDs to be processed would take more than 50% of the RID pool when the query is executed. You can change the size of the RID pool in the field RID POOL SIZE on installation panel DSNTIPC. The maximum size of a RID pool is 10 000 MB. The maximum size of a single RID list is approximately 26 million RIDs. For information about calculating RID pool size, see Part 5 (Volume 2) of *DB2 Administration Guide*.

During execution, DB2 ends list prefetching if more than 25% of the rows in the table (with a minimum of 6524) must be accessed. Record IFCID 0125 in the performance trace, mapped by macro DSNDQW01, indicates whether list prefetch ended.

When list prefetch ends, the query continues processing by a method that depends on the current access path.

- For access through a single index or through the union of RID lists from two indexes, processing continues by a table space scan.
- For index access before forming an intersection of RID lists, processing continues with the next step of multiple index access. If no step remains and no RID list has been accumulated, processing continues by a table space scan.

When DB2 forms an intersection of RID lists, if any list has 32 or fewer RIDs, intersection stops and the list of 32 or fewer RIDs is used to access the data.

Sequential detection at execution time

If DB2 does not choose prefetch at bind time, it can sometimes use prefetch at execution time nevertheless. The method is called *sequential detection*.

When sequential detection is used

DB2 can use sequential detection for both index leaf pages and data pages. It is most commonly used on the inner table of a nested loop join, if the data is accessed sequentially.

If a table is accessed repeatedly using the same statement (for example, DELETE in a do-while loop), the data or index leaf pages of the table can be accessed sequentially. This is common in a batch processing environment. Sequential detection can then be used if access is through:

- SELECT or FETCH statements
- UPDATE and DELETE statements
- INSERT statements when existing data pages are accessed sequentially

DB2 can use sequential detection if it did not choose sequential prefetch at bind time because of an inaccurate estimate of the number of pages to be accessed.

Sequential detection is not used for an SQL statement that is subject to referential constraints.

How to tell whether sequential detection was used

A plan table does not indicate sequential detection, which is not determined until run time. You can determine whether sequential detection was used from record IFCID 0003 in the accounting trace or record IFCID 0006 in the performance trace.

How to tell if sequential detection might be used

The pattern of accessing a page is tracked when the application scans DB2 data through an index. Tracking is done to detect situations where the access pattern that develops is sequential or nearly sequential.

The most recent eight pages are tracked. A page is considered page-sequential if it is within $P/2$ advancing pages of the current page, where P is the prefetch quantity. P is usually 32.

If a page is page-sequential, DB2 determines further if data access is sequential or nearly sequential. Data access is declared sequential if more than 4 out of the last eight pages are page-sequential; this is also true for index-only access. The tracking is continuous, allowing access to slip into and out of data access sequential.

When data access is first declared sequential, which is called *initial data access* sequential, three page ranges are calculated as follows:

- Let A be the page being requested. RUN1 is defined as the page range of length $P/2$ pages starting at A .
- Let B be page $A + P/2$. RUN2 is defined as the page range of length $P/2$ pages starting at B .
- Let C be page $B + P/2$. RUN3 is defined as the page range of length P pages starting at C .

For example, assume that page A is 10. Figure 236 on page 834 illustrates the page ranges that DB2 calculates.

	A	B	C
	RUN1		RUN2
Page #	10	26	42
P=32 pages	16	16	32

Figure 236. Initial page ranges to determine when to use prefetch

For initial data access sequential, prefetch is requested starting at page A for P pages (RUN1 and RUN2). The prefetch quantity is always P pages.

For subsequent page requests where the page is 1) page sequential and 2) data access sequential is still in effect, prefetch is requested as follows:

- If the desired page is in RUN1, no prefetch is triggered because it was already triggered when data access sequential was first declared.
- If the desired page is in RUN2, prefetch for RUN3 is triggered and RUN2 becomes RUN1, RUN3 becomes RUN2, and RUN3 becomes the page range starting at C+P for a length of P pages.

If a data access pattern develops such that data access sequential is no longer in effect and, thereafter, a new pattern develops that is sequential, then initial data access sequential is declared again and handled accordingly.

Because, at bind time, the number of pages to be accessed can only be estimated, sequential detection acts as a safety net and is employed when the data is being accessed sequentially.

In extreme situations, when certain buffer pool thresholds are reached, sequential prefetch can be disabled. For a description of buffer pools and thresholds, see Part 5 (Volume 2) of *DB2 Administration Guide*.

Determining sort activity

DB2 can use two general types of sorts that DB2 can use when accessing data. One is a sort of data rows; the other is a sort of row identifiers (RIDs) in a RID list.

Sorts of data

After you run EXPLAIN, DB2 sorts are indicated in PLAN_TABLE. The sorts can be either sorts of the composite table or the new table. If a single row of PLAN_TABLE has a 'Y' in more than one of the sort composite columns, then one sort accomplishes two things. (DB2 will not perform two sorts when two 'Y's are in the same row.) For instance, if both SORTC_ORDERBY and SORTC_UNIQ are 'Y' in one row of PLAN_TABLE, then a single sort puts the rows in order and removes any duplicate rows as well.

The only reason DB2 sorts the new table is for join processing, which is indicated by SORTN_JOIN.

Sorts for group by and order by

These sorts are indicated by SORTC_ORDERBY, and SORTC_GROUPBY in PLAN_TABLE. If there is both a GROUP BY clause and an ORDER BY clause, and if every item in the ORDER-BY list is in the GROUP-BY list, then only one sort is performed, which is marked as SORTC_ORDERBY.

The performance of the sort by the GROUP BY clause is improved when the query accesses a single table and when the GROUP BY column has no index.

Sorts to remove duplicates

This type of sort is used to process a query with SELECT DISTINCT, with a set function such as COUNT(DISTINCT COL1), or to remove duplicates in UNION processing. It is indicated by SORTC_UNIQ in PLAN_TABLE.

Sorts used in join processing

Before joining two tables, it is often necessary to first sort either one or both of them. For hybrid join (METHOD 4) and nested loop join (METHOD 1), the composite table can be sorted to make the join more efficient. For merge join (METHOD 2), both the composite table and new table need to be sorted unless an index is used for accessing these tables that gives the correct order already. The sorts needed for join processing are indicated by SORTN_JOIN and SORTC_JOIN in the PLAN_TABLE.

Sorts needed for subquery processing

When a noncorrelated IN or NOT IN subquery is present in the query, the results of the subquery are sorted and put into a work file for later reference by the parent query. The results of the subquery are sorted because this allows the parent query to be more efficient when processing the IN or NOT IN predicate. Duplicates are not needed in the work file, and are removed. Noncorrelated subqueries used with =ANY or =ALL, or NOT=ANY or NOT=ALL also need the same type of sort as IN or NOT IN subqueries. When a sort for a noncorrelated subquery is performed, you see both SORTC_ORDERBY and SORTC_UNIQUE in PLAN_TABLE. This is because DB2 removes the duplicates and performs the sort.

SORTN_GROUPBY, SORTN_ORDERBY, and SORTN_UNIQ are not currently used by DB2.

Sorts of RIDs

To perform list prefetch, DB2 sorts RIDs into ascending page number order. This sort is very fast and is done totally in memory. A RID sort is usually not indicated in the PLAN_TABLE, but a RID sort normally is performed whenever list prefetch is used. The only exception to this rule is when a hybrid join is performed and a single, highly clustered index is used on the inner table. In this case SORTN_JOIN is 'N', indicating that the RID list for the inner table was not sorted.

The effect of sorts on OPEN CURSOR

The type of sort processing required by the cursor affects the amount of time it can take for DB2 to process the OPEN CURSOR statement. This section outlines the effect of sorts and parallelism on OPEN CURSOR.

Without parallelism:

- If no sorts are required, then OPEN CURSOR does not access any data. It is at the first fetch that data is returned.
- If a sort is required, then the OPEN CURSOR causes the materialized result table to be produced. Control returns to the application after the result table is materialized. If a cursor that requires a sort is closed and reopened, the sort is performed again.
- If there is a RID sort, but no data sort, then it is not until the first row is fetched that the RID list is built from the index and the first data record is returned. Subsequent fetches access the RID pool to access the next data record.

With parallelism:

- At OPEN CURSOR, parallelism is asynchronously started, regardless of whether a sort is required. Control returns to the application immediately after the parallelism work is started.
- If there is a RID sort, but no data sort, then parallelism is not started until the first fetch. This works the same way as with no parallelism.

Processing for views and nested table expressions

This section describes how DB2 processes views and nested table expressions. A nested table expression (which is called *table expression* in this description) is the specification of a subquery in the FROM clause of an SQL SELECT statement. The processing of table expressions is similar to a view. Two methods are used to satisfy your queries that reference views or table expressions:

- “Merge”
- “Materialization” on page 837

You can determine the methods that are used by executing EXPLAIN for the statement that contains the view or nested table expression. In addition, you can use EXPLAIN to determine when UNION operators are used and how DB2 might eliminate unnecessary subselects to improve the performance of a query.

Merge

The merge process is more efficient than materialization, as described in “Performance of merge versus materialization” on page 842. In the merge process, the statement that references the view or table expression is combined with the fullselect that defined the view or table expression. This combination creates a logically equivalent statement. This equivalent statement is executed against the database.

Example: Consider the following statements, one of which defines a view, the other of which references the view:

View-defining statement:

```
CREATE VIEW VIEW1 (VC1,VC21,VC32) AS
  SELECT C1,C2,C3 FROM T1
  WHERE C1 > C3;
```

View referencing statement:

```
SELECT VC1,VC21
  FROM VIEW1
  WHERE VC1 IN (A,B,C);
```

The fullselect of the view-defining statement can be merged with the view-referencing statement to yield the following logically equivalent statement:

Merged statement:

```
SELECT C1,C2 FROM T1
  WHERE C1 > C3 AND C1 IN (A,B,C);
```

Example: The following statements show another example of when a view and table expression can be merged:

```
SELECT * FROM V1 X
  LEFT JOIN
    (SELECT * FROM T2) Y ON X.C1=Y.C1
  LEFT JOIN T3 Z ON X.C1=Z.C1;
```

Merged statement:

```
SELECT * FROM V1 X
  LEFT JOIN
    T2 ON X.C1 = T2.C1
  LEFT JOIN T3 Z ON X.C1 = Z.C1;
```

Materialization

Views and table expressions cannot always be merged.

Example: Look at the following statements:

View defining statement:

```
CREATE VIEW VIEW1 (VC1,VC2) AS
  SELECT SUM(C1),C2 FROM T1
  GROUP BY C2;
```

View referencing statement:

```
SELECT MAX(VC1)
  FROM VIEW1;
```

Column VC1 occurs as the argument of a aggregate function in the view referencing statement. The values of VC1, as defined by the view-defining fullselect, are the result of applying the aggregate function SUM(C1) to groups after grouping the base table T1 by column C2. No equivalent single SQL SELECT statement can be executed against the base table T1 to achieve the intended result. There is no way to specify that aggregate functions should be applied successively.

Two steps of materialization

In the previous example, DB2 performs materialization of the view or table expression, which is a two step process.

1. The fullselect that defines the view or table expression is executed against the database, and the results are placed in a temporary copy of a result table.
2. The statement that references the view or table expression is then executed against the temporary copy of the result table to obtain the intended result.

Whether materialization is needed depends upon the attributes of the referencing statement, or logically equivalent referencing statement from a prior merge, and the attributes of the fullselect that defines the view or table expression.

When views or table expressions are materialized

In general, DB2 uses materialization to satisfy a reference to a view or table expression when there is aggregate processing (grouping, aggregate functions, distinct), indicated by the defining fullselect, in conjunction with either aggregate processing indicated by the statement referencing the view or table expression, or by the view or table expression participating in a join. For views and table expressions that are defined with UNION or UNION ALL, DB2 can often distribute aggregate processing, joins, and qualified predicates to avoid materialization. For more information, see “Using EXPLAIN to determine UNION activity and query rewrite” on page 840.

Table 122 indicates some cases in which materialization occurs. DB2 can also use materialization in statements that contain multiple outer joins, outer joins that combine with inner joins, or merges that cause a join of greater than 15 tables.

Table 122. Cases when DB2 performs view or table expression materialization. The "X" indicates a case of materialization. Notes follow the table.

SELECT FROM view or table expression uses... ¹	View definition or table expression uses... ²					
	GROUP BY	DISTINCT	Aggregate function	Aggregate function DISTINCT	UNION	UNION ALL(4)
Joins (3)	X	X	X	X	X	
GROUP BY	X	X	X	X	X	
DISTINCT		X		X	X	

Table 122. Cases when DB2 performs view or table expression materialization (continued). The "X" indicates a case of materialization. Notes follow the table.

SELECT FROM view or table expression uses... ¹	View definition or table expression uses... ²					
	GROUP BY	DISTINCT	Aggregate function	Aggregate function DISTINCT	UNION	UNION ALL(4)
Aggregate function	X	X	X	X	X	X
Aggregate function DISTINCT	X	X	X	X	X	
SELECT subset of view or table expression columns		X			X	

Notes to Table 122 on page 837:

1. If the view is referenced as the target of an INSERT, UPDATE, or DELETE, then view merge is used to satisfy the view reference. Only updatable views can be the target in these statements. See Chapter 5 of *DB2 SQL Reference* for information about which views are read-only (not updatable).

An SQL statement can reference a particular view multiple times where some of the references can be merged and some must be materialized.

2. If a SELECT list contains a host variable in a table expression, then materialization occurs. For example:

```
SELECT C1 FROM
  (SELECT :HV1 AS C1 FROM T1) X;
```

If a view or nested table expression is defined to contain a user-defined function, and if that user-defined function is defined as NOT DETERMINISTIC or EXTERNAL ACTION, then the view or nested table expression is always materialized.

3. Additional details about materialization with outer joins:

- If a WHERE clause exists in a view or table expression, and it does not contain a column, materialization occurs.

Example:

```
SELECT X.C1 FROM
  (SELECT C1 FROM T1
   WHERE 1=1) X LEFT JOIN T2 Y
    ON X.C1=Y.C1;
```

- If the outer join is a full outer join and the SELECT list of the view or nested table expression does not contain a standalone column for the column that is used in the outer join ON clause, then materialization occurs.

Example:

```
SELECT X.C1 FROM
  (SELECT C1+10 AS C2 FROM T1) X FULL JOIN T2 Y
    ON X.C2=Y.C2;
```

- If there is no column in a SELECT list of a view or nested table expression, materialization occurs.

Example:

```
SELECT X.C1 FROM
  (SELECT 1+2+:HV1. AS C1 FROM T1) X LEFT JOIN T2 Y
    ON X.C1=Y.C1;
```


- # If the SELECT list of a view or nested table expression contains a CASE expression, and the result of the CASE expression is referenced in the outer query block, then materialization occurs.
- # **Example:**
- #

```
SELECT X.C1 FROM
```
- #

```
      T1 X LEFT JOIN
```
- #

```
      (SELECT CASE C2 WHEN 5 THEN 10 ELSE 20 END AS YC1 FROM T2) Y
```
- #

```
      ON X.C1 = Y.YC1;
```
- #
- 4. DB2 cannot avoid materialization for UNION ALL in all cases. Some of the situations in which materialization occurs includes:
 - When the view is the operand in an outer join for which nulls are used for non-matching values, materialization occurs. This situation happens when the view is either operand in a full outer join, the right operand in a left outer join, or the left operand in a right outer join.
 - If the number of tables would exceed 225 after distribution, then distribution will not occur, and the result will be materialized.

Using EXPLAIN to determine when materialization occurs

For each reference to a view or table expression that is materialized, rows describing the access path for both steps of the materialization process appear in the PLAN_TABLE. These rows describe the access path used to formulate the temporary result indicated by the view's defining fullselect, and they describe the access to the temporary result as indicated by the referencing statement. The defining fullselect can also refer to views or table expressions that need to be materialized.

When DB2 chooses materialization, TNAME contains the name of the view or table expression, and TABLE_TYPE contains a W. A value of Q in TABLE_TYPE for the name of a view or nested table expression indicates that the materialization was virtual and not actual. (Materialization can be virtual when the view or nested table expression definition contains a UNION ALL that is not distributed.) When DB2 chooses merge, EXPLAIN data for the merged statement appears in PLAN_TABLE; only the names of the base tables on which the view or table expression is defined appear.

Example: Consider the following statements, which define a view and reference the view:

View defining statement:

```
CREATE VIEW V1DIS (SALARY, WORKDEPT) as
  (SELECT DISTINCT SALARY, WORKDEPT FROM DSN8810.EMP)
```

View referencing statement:

```
SELECT * FROM DSN8810.DEPT
  WHERE DEPTNO IN (SELECT WORKDEPT FROM V1DIS)
```

Table 123 shows a subset of columns in a plan table for the query.

Table 123. Plan table output for an example with view materialization

QBLOCKNO	PLANNO	QBLOCK_ TYPE	TNAME	TABLE_ TYPE	METHOD
1	1	SELECT	DEPT	T	0
2	1	NOCOSUB	V1DIS	W	0
2	2	NOCOSUB		?	3

Table 123. Plan table output for an example with view materialization (continued)

QBLOCKNO	PLANNO	QBLOCK_ TYPE	TNAME	TABLE_ TYPE	METHOD
3	1	NOCOSUB	EMP	T	0
3	2	NOCOSUB		?	3

Notice how TNAME contains the name of the view and TABLE_TYPE contains W to indicate that DB2 chooses materialization for the reference to the view because of the use of SELECT DISTINCT in the view definition.

Example: Consider the following statements, which define a view and reference the view:

View defining statement:

```
CREATE VIEW V1NODIS (SALARY, WORKDEPT) as
  (SELECT SALARY, WORKDEPT FROM DSN8810.EMP)
```

View referencing statement:

```
SELECT * FROM DSN8810.DEPT
  WHERE DEPTNO IN (SELECT WORKDEPT FROM V1NODIS)
```

If the VIEW was defined without DISTINCT, DB2 would choose merge instead of materialization. In the sample output, the name of the view does not appear in the plan table, but the table name on which the view is based does appear.

Table 124 shows a sample plan table for the query.

Table 124. Plan table output for an example with view merge

QBLOCKNO	PLANNO	QBLOCK_ TYPE	TNAME	TABLE_ TYPE	METHOD
1	1	SELECT	DEPT	T	0
2	1	NOCOSUB	EMP	T	0
2	2	NOCOSUB		?	3

For an example of when a view definition contains a UNION ALL and DB2 can distribute joins and aggregations and avoid materialization, see “Using EXPLAIN to determine UNION activity and query rewrite.” When DB2 avoids materialization in such cases, TABLE_TYPE contains a Q to indicate that DB2 uses an intermediate result that is not materialized, and TNAME shows the name of this intermediate result as DSNWFQB(xx), where xx is the number of the query block that produced the result.

Using EXPLAIN to determine UNION activity and query rewrite

For each reference to a view or table expression that is defined with UNION or UNION ALL operators, DB2 tries to rewrite the query into a logically equivalent statement with improved performance by:

- Distributing qualified predicates, joins, and aggregations across the subselects of UNION ALL. Such distribution helps to avoid materialization. No distribution is performed for UNION.
- Eliminating unnecessary subselects of the view or table expression. For DB2 to eliminate subselects, the referencing query and the view or table definition must have predicates that are based on common columns.

The QBLOCK_TYPE column in the plan table indicates union activity. For a UNION ALL, the column contains 'UNIONA'. For UNION, the column contains 'UNION'. When QBLOCK_TYPE='UNION', the METHOD column on the same row is set to 3 and the SORTC_UNIQ column is set to 'Y' to indicate that a sort is necessary to remove duplicates. As with other views and table expressions, the plan table also shows when DB2 uses materialization instead of merge.

Example: Consider the following statements, which define a view, reference the view, and show how DB2 rewrites the referencing statement:

View defining statement: View is created on three tables that contain weekly data

```
CREATE VIEW V1 (CUSTNO, CHARGES, DATE) as
  SELECT CUSTNO, CHARGES, DATE
    FROM WEEK1
   WHERE DATE BETWEEN '01/01/2000' And '01/07/2000'
 UNION ALL
  SELECT CUSTNO, CHARGES, DATE
    FROM WEEK2
   WHERE DATE BETWEEN '01/08/2000' And '01/14/2000'
 UNION ALL
  SELECT CUSTNO, CHARGES, DATE
    FROM WEEK3
   WHERE DATE BETWEEN '01/15/2000' And '01/21/2000';
```

View referencing statement: For each customer in California, find the average charges during the first and third Friday of January 2000

```
SELECT V1.CUSTNO, AVG(V1.CHARGES)
  FROM CUST, V1
 WHERE CUST.CUSTNO=V1.CUSTNO
    AND CUST.STATE='CA'
    AND DATE IN ('01/07/2000','01/21/2000')
 GROUP BY V1.CUSTNO;
```

Rewritten statement (assuming that CHARGES is defined as NOT NULL):

```
SELECT CUSTNO_U, SUM(SUM_U)/SUM(CNT_U)
  FROM
 ( SELECT WEEK1.CUSTNO, SUM(CHARGES), COUNT(CHARGES)
   FROM CUST, WEEK1
  Where CUST.CUSTNO=WEEK1.CUSTNO AND CUST.STATE='CA'
        AND DATE BETWEEN '01/01/2000' And '01/07/2000'
        AND DATE IN ('01/07/2000','01/21/2000')
   GROUP BY WEEK1.CUSTNO
 UNION ALL
  SELECT WEEK3.CUSNTO, SUM(CHARGES), COUNT(CHARGES)
   FROM CUST,WEEK3
  WHERE CUST.CUSTNO=WEEK3 AND CUST.STATE='CA'
        AND DATE BETWEEN '01/15/2000' And '01/21/2000'
        AND DATE IN ('01/07/2000','01/21/2000')
   GROUP BY WEEK3.CUSTNO
 ) AS X(CUSTNO_U,SUM_U,CNT_U)
 GROUP BY CUSNTO_U;
```

Table 125 shows a subset of columns in a plan table for the query.

Table 125. Plan table output for an example with a view with UNION ALLs

QBLOCKNO	PLANNO	TNAME	TABLE_TYPE	METHOD	QBLOCK_TYPE	PARENT_QBLOCKNO
1	1	DSNWFQB(02)	Q	0		0
1	2		?	3		0
2	1		?	0	UNIONA	1
3	1	CUST	T	0		2

Table 125. Plan table output for an example with a view with UNION ALLs (continued)

QBLOCKNO	PLANNO	TNAME	TABLE_TYPE	METHOD	QBLOCK_TYPE	PARENT_QBLOCKNO
3	2	WEEK1	T	1		2
4	1	CUST	T	0		2
4	2	WEEK3	T	2		2

Notice how DB2 eliminates the second subselect of the view definition from the rewritten query and how the plan table indicates this removal by showing a UNION ALL for only the first and third subselect in the view definition. The Q in the TABLE_TYPE column indicates that DB2 does not materialize the view.

Performance of merge versus materialization

Merge performs better than materialization. For materialization, DB2 uses a table space scan to access the materialized temporary result. DB2 materializes a view or table expression only if it cannot merge.

Materialization is a two-step process with the first step resulting in the formation of a temporary result. The smaller the temporary result, the more efficient is the second step. To reduce the size of the temporary result, DB2 attempts to evaluate certain predicates from the WHERE clause of the referencing statement at the first step of the process rather than at the second step. Only certain types of predicates qualify. First, the predicate must be a simple Boolean term predicate. Second, it must have one of the forms shown in Table 126.

Table 126. Predicate candidates for first-step evaluation

Predicate	Example
COL op literal	V1.C1 > hv1
COL IS (NOT) NULL	V1.C1 IS NOT NULL
COL (NOT) BETWEEN literal AND literal	V1.C1 BETWEEN 1 AND 10
COL (NOT) LIKE constant (ESCAPE constant)	V1.C2 LIKE 'p\%%' ESCAPE '\'
COL IN (list)	V1.C2 IN (a,b,c)

Note: Where "op" is =, <>, >, <, <=, or >=, and literal is either a host variable, constant, or special register. The literals in the BETWEEN predicate need not be identical.

Implied predicates generated through predicate transitive closure are also considered for first step evaluation.

Estimating a statement's cost

You can use EXPLAIN to populate a statement table, *owner.DSN_STATEMNT_TABLE*, at the same time as your *PLAN_TABLE* is being populated. DB2 provides cost estimates, in service units and in milliseconds, for SELECT, INSERT, UPDATE, and DELETE statements, both static and dynamic. The estimates do not take into account several factors, including cost adjustments that are caused by parallel processing, or the use of triggers or user-defined functions.

Use the information provided in the statement table to:

- Help you determine if a statement is not going to perform within range of your service-level agreements and to tune accordingly.

DB2 puts its cost estimate into one of two *cost categories*: category A or category B. Estimates that go into cost category A are the ones for which DB2 has

adequate information to make an estimate. That estimate is not likely to be 100% accurate, but is likely to be more accurate than any estimate that is in cost category B.

DB2 puts estimates into cost category B when it is forced to use default values for its estimates, such as when no statistics are available, or because host variables are used in a query. See the description of the REASON column in Table 127 on page 844 for more information about how DB2 determines into which cost category an estimate goes.

- Give a system programmer a basis for entering service-unit values by which to govern dynamic statements.

Information about using predictive governing is in Part 5 (Volume 2) of *DB2 Administration Guide*.

This section describes the following tasks to obtain and use cost estimate information from EXPLAIN:

1. “Creating a statement table”
2. “Populating and maintaining a statement table” on page 845
3. “Retrieving rows from a statement table” on page 845
4. “The implications of cost categories” on page 846

For more information about how to change applications to handle the SQLCODES that are associated with predictive governing, see “Writing an application to handle predictive governing” on page 602.

Creating a statement table

To collect information about the estimated cost of a statement, create a table called DSN_STATEMNT_TABLE to hold the results of EXPLAIN. A copy of the statements that are needed to create the table are in the DB2 sample library, under the member name DSNTEESC.

Figure 237 shows the current format of a statement table.

```
CREATE TABLE DSN_STATEMNT_TABLE
( QUERYNO          INTEGER          NOT NULL WITH DEFAULT,
  APPLNAME         CHAR(8)          NOT NULL WITH DEFAULT,
  PROGNAME         VARCHAR(128)     NOT NULL WITH DEFAULT,
  COLLID           VARCHAR(128)     NOT NULL WITH DEFAULT,
  GROUP_MEMBER     CHAR(8)          NOT NULL WITH DEFAULT,
  EXPLAIN_TIME     TIMESTAMP        NOT NULL WITH DEFAULT,
  STMT_TYPE        CHAR(6)          NOT NULL WITH DEFAULT,
  COST_CATEGORY    CHAR(1)          NOT NULL WITH DEFAULT,
  PROCMS           INTEGER          NOT NULL WITH DEFAULT,
  PROCSU           INTEGER          NOT NULL WITH DEFAULT,
  REASON           VARCHAR(254)     NOT NULL WITH DEFAULT,
  STMT_ENCODE      CHAR(1)          NOT NULL WITH DEFAULT);
```

Figure 237. Current format of DSN_STATEMNT_TABLE

Your statement table can use an older format in which the STMT_ENCODE column does not exist, PROGNAME has a data type of CHAR(8), and COLLID has a data type of CHAR(18). However, use the most current format because it gives you the most information. You can alter a statement table in the older format to a statement table in the current format.

Table 127 on page 844 shows the content of each column.

Table 127. Descriptions of columns in DSN_STATEMNT_TABLE

Column Name	Description
QUERYNO	A number that identifies the statement being explained. See the description of the QUERYNO column in Table 101 on page 794 for more information. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.
APPLNAME	The name of the application plan for the row, or blank. See the description of the APPLNAME column in Table 101 on page 794 for more information.
PROGNAME	The name of the program or package containing the statement being explained, or blank. See the description of the PROGNAME column in Table 101 on page 794 for more information.
COLLID	The collection ID for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. Blank if not applicable. The value DSNDDYNAMICSQLCACHE indicates that the row is for a cached statement.
GROUP_MEMBER	The member name of the DB2 that executed EXPLAIN, or blank. See the description of the GROUP_MEMBER column in Table 101 on page 794 for more information.
EXPLAIN_TIME	The time at which the statement is processed. This time is the same as the BIND_TIME column in PLAN_TABLE.
STMT_TYPE	The type of statement being explained. Possible values are: SELECT SELECT INSERT INSERT UPDATE UPDATE DELETE DELETE SELUPD SELECT with FOR UPDATE OF DELCUR DELETE WHERE CURRENT OF CURSOR UPDCUR UPDATE WHERE CURRENT OF CURSOR
COST_CATEGORY	Indicates if DB2 was forced to use default values when making its estimates. Possible values: A Indicates that DB2 had enough information to make a cost estimate without using default values. B Indicates that some condition exists for which DB2 was forced to use default values. See the values in REASON to determine why DB2 was unable to put this estimate in cost category A.
PROCMS	The estimated processor cost, in milliseconds, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 milliseconds, which is equivalent to approximately 24.8 days. If the estimated value exceeds this maximum, the maximum value is reported.
PROCSU	The estimated processor cost, in service units, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 service units. If the estimated value exceeds this maximum, the maximum value is reported.

Table 127. Descriptions of columns in DSN_STATEMNT_TABLE (continued)

Column Name	Description
REASON	A string that indicates the reasons for putting an estimate into cost category B.
HAVING CLAUSE	A subselect in the SQL statement contains a HAVING clause.
HOST VARIABLES	The statement uses host variables, parameter markers, or special registers.
REFERENTIAL CONSTRAINTS	Referential constraints of the type CASCADE or SET NULL exist on the target table of a DELETE statement.
TABLE CARDINALITY	The cardinality statistics are missing for one or more of the tables that are used in the statement. Or, the statement required the materialization of views or nested table expressions.
TRIGGERS	Triggers are defined on the target table of an INSERT, UPDATE, or DELETE statement.
UDF	The statement uses user-defined functions.
STMT_ENCODE	Encoding scheme of the statement. If the statement represents a single CCSID set, the possible values are: A ASCII E EBCDIC U Unicode If the statement has multiple CCSID sets, the value is M.

Populating and maintaining a statement table

You populate a statement table at the same time as you populate the corresponding plan table. For more information, see “Populating and maintaining a plan table” on page 798.

Just as with the plan table, DB2 just adds rows to the statement table; it does not automatically delete rows. INSERT triggers are not activated unless you insert rows yourself using and SQL INSERT statement.

To clear the table of obsolete rows, use DELETE, just as you would for deleting rows from any table. You can also use DROP TABLE to drop a statement table completely.

Retrieving rows from a statement table

To retrieve all rows in a statement table, you can use a query like the following statement, which retrieves all rows about the statement that is represented by query number 13:

```
SELECT * FROM JOE.DSN_STATEMNT_TABLE
WHERE QUERYNO = 13;
```

The QUERYNO, APPLNAME, PROGNAME, COLLID, and EXPLAIN_TIME columns contain the same values as corresponding columns of PLAN_TABLE for a given plan. You can use these columns to join the plan table and statement table:

```
SELECT A.*, PROCMS, COST_CATEGORY
FROM JOE.PLAN_TABLE A, JOE.DSN_STATEMNT_TABLE B
WHERE A.APPLNAME = 'APPL1' AND
A.APPLNAME = B.APPLNAME AND
```

#

```
A.QUERYNO = B.QUERYNO AND A.PROGNAME = B.PROGNAME AND  
A.COLLID = B.COLLID AND  
A.BIND_TIME = B.EXPLAIN_TIME  
ORDER BY A.QUERYNO, A.QBLOCKNO, A.PLANNO, A.MIXOPSEQ;
```

The implications of cost categories

Cost categories are DB2's way of differentiating estimates for which adequate information is available from those for which it is not. You probably wouldn't want to spend a lot of time tuning a query based on estimates that are returned in cost category B, because the actual cost could be radically different based on such things as what value is in a host variable, or how many levels of nested triggers and user-defined functions exist.

Similarly, if system administrators use these estimates as input into the resource limit specification table for governing (either predictive or reactive), they probably would want to give much greater latitude for statements in cost category B than for those in cost category A.

Because of the uncertainty involved, category B statements are also good candidates for reactive governing.

What goes into cost category B? DB2 puts a statement's estimate into cost category B when any of the following conditions exist:

- The statement has UDFs.
- Triggers are defined for the target table:
 - The statement is INSERT, and insert triggers are defined on the target table.
 - The statement is UPDATE, and update triggers are defined on the target table.
 - The statement is DELETE, and delete triggers are defined on the target table.
- The target table of a delete statement has referential constraints defined on it as the parent table, and the delete rules are either CASCADE or SET NULL.
- The WHERE clause predicate has one of the following forms:
 - COL op literal, and the literal is a host variable, parameter marker, or special register. The operator can be >, >=, <, <=, LIKE, or NOT LIKE.
 - COL BETWEEN literal AND literal where either literal is a host variable, parameter marker, or special register.
 - LIKE with an escape clause that contains a host variable.
- The cardinality statistics are missing for one or more tables that are used in the statement.
- A subselect in the SQL statement contains a HAVING clause.

What goes into cost category A? DB2 puts everything that doesn't fall into category B into category A.

Chapter 28. Parallel operations and query performance

When DB2 plans to access data from a table or index in a partitioned table space, it can initiate multiple parallel operations. The response time for data or processor-intensive queries can be significantly reduced.

Query I/O parallelism manages concurrent I/O requests for a single query, fetching pages into the buffer pool in parallel. This processing can significantly improve the performance of I/O-bound queries. I/O parallelism is used only when one of the other parallelism modes cannot be used.

Query CP parallelism enables true multitasking within a query. A large query can be broken into multiple smaller queries. These smaller queries run simultaneously on multiple processors accessing data in parallel. This reduces the elapsed time for a query.

To expand even farther the processing capacity available for processor-intensive queries, DB2 can split a large query across different DB2 members in a data sharing group. This is known as Sysplex query parallelism. For more information about Sysplex query parallelism, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*.

DB2 can use parallel operations for processing:

- Static and dynamic queries
- Local and remote data access
- Queries using single table scans and multi-table joins
- Access through an index, by table space scan or by list prefetch
- Sort operations

Parallel operations usually involve at least one table in a partitioned table space. Scans of large partitioned table spaces have the greatest performance improvements where both I/O and central processor (CP) operations can be carried out in parallel.

Parallelism for partitioned and nonpartitioned table spaces: Both partitioned and nonpartitioned table spaces can take advantage of query parallelism. Parallelism is now enabled to include non-clustering indexes. Thus, table access can be run in parallel when the application is bound with DEGREE (ANY) and the table is accessed through a non-clustering index.

This chapter contains the following topics:

- “Comparing the methods of parallelism” on page 848
- “Enabling parallel processing” on page 850
- “When parallelism is not used” on page 851
- “Interpreting EXPLAIN output” on page 852
- “Tuning parallel processing” on page 854
- “Disabling query parallelism” on page 855

Comparing the methods of parallelism

The figures in this section show how the parallel methods compare with sequential prefetch and with each other. All three techniques assume access to a table space with three partitions, P1, P2, and P3. The notations P1, P2, and P3 are partitions of a table space. R1, R2, R3, and so on, are requests for sequential prefetch. The combination P2R1, for example, means the first request from partition 2.

Figure 238 shows **sequential processing**. With sequential processing, DB2 takes the 3 partitions in order, completing partition 1 before starting to process partition 2, and completing 2 before starting 3. Sequential prefetch allows overlap of CP processing with I/O operations, but I/O operations do not overlap with each other. In the example in Figure 238, a prefetch request takes longer than the time to process it. The processor is frequently waiting for I/O.

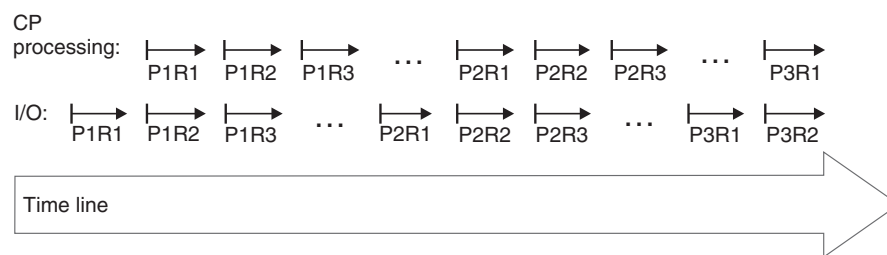


Figure 238. CP and I/O processing techniques. Sequential processing.

Figure 239 shows **parallel I/O operations**. With parallel I/O, DB2 prefetches data from the 3 partitions at one time. The processor processes the first request from each partition, then the second request from each partition, and so on. The processor is not waiting for I/O, but there is still only one processing task.

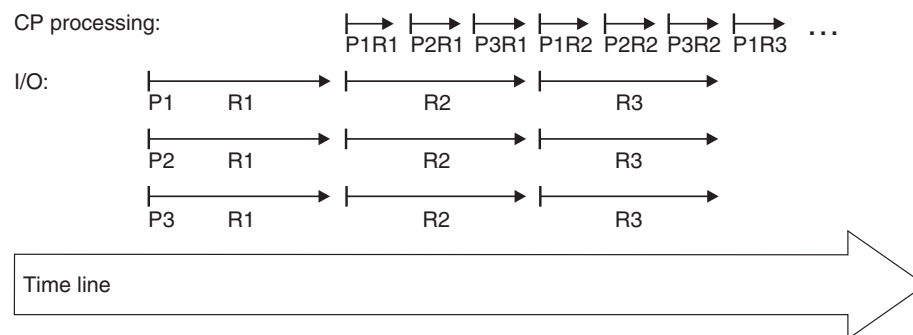


Figure 239. CP and I/O processing techniques. Parallel I/O processing.

Figure 240 on page 849 shows **parallel CP processing**. With CP parallelism, DB2 can use multiple parallel tasks to process the query. Three tasks working concurrently can greatly reduce the overall elapsed time for data-intensive and processor-intensive queries. The same principle applies for **Sysplex query parallelism**, except that the work can cross the boundaries of a single CPC.

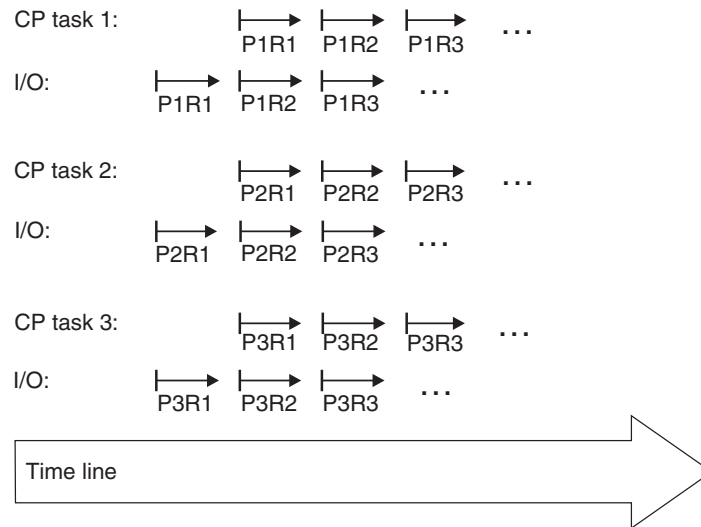


Figure 240. CP and I/O processing techniques. Query processing using CP parallelism. The tasks can be contained within a single CPC or can be spread out among the members of a data sharing group.

Queries that are most likely to take advantage of parallel operations: Queries that can take advantage of parallel processing are:

- Those in which DB2 spends most of the time fetching pages—an I/O-intensive query

A typical I/O-intensive query is something like the following query, assuming that a table space scan is used on many pages:

```
SELECT COUNT(*) FROM ACCOUNTS
WHERE BALANCE > 0 AND
      DAYS_OVERDUE > 30;
```

- Those in which DB2 spends a lot of processor time and also, perhaps, I/O time, to process rows. Those include:
 - *Queries with intensive data scans and high selectivity.* Those queries involve large volumes of data to be scanned but relatively few rows that meet the search criteria.
 - *Queries containing aggregate functions.* Column functions (such as MIN, MAX, SUM, AVG, and COUNT) usually involve large amounts of data to be scanned but return only a single aggregate result.
 - *Queries accessing long data rows.* Those queries access tables with long data rows, and the ratio of rows per page is very low (one row per page, for example).
 - *Queries requiring large amounts of central processor time.* Those queries might be read-only queries that are complex, data-intensive, or that involve a sort.

A typical processor-intensive query is something like:

```
SELECT MAX(QTY_ON_HAND) AS MAX_ON_HAND,
       AVG(PRICE) AS AVG_PRICE,
       AVG(DISCOUNTED_PRICE) AS DISC_PRICE,
       SUM(TAX) AS SUM_TAX,
       SUM(QTY_SOLD) AS SUM_QTY_SOLD,
       SUM(QTY_ON_HAND - QTY_BROKEN) AS QTY_GOOD,
       AVG(DISCOUNT) AS AVG_DISCOUNT,
       ORDERSTATUS,
       COUNT(*) AS COUNT_ORDERS
FROM   ORDER_TABLE
```

```

WHERE SHIPPER = 'OVERNIGHT' AND
      SHIP_DATE < DATE('1996-01-01')
GROUP BY ORDERSTATUS
ORDER BY ORDERSTATUS;

```

Terminology: When the term **task** is used with information about parallel processing, the context should be considered. For parallel query CP processing or Sysplex query parallelism, a task is an actual z/OS execution unit used to process a query. For parallel I/O processing, a task simply refers to the processing of one of the concurrent I/O streams.

A **parallel group** is the term used to name a particular set of parallel operations (parallel tasks or parallel I/O operations). A query can have more than one parallel group, but each parallel group within the query is identified by its own unique ID number.

The **degree of parallelism** is the number of parallel tasks or I/O operations that DB2 determines can be used for the operations on the parallel group. The maximum of parallel operations that DB2 can generate is 254. However, for most queries and DB2 environments, DB2 chooses a lower number. You might need to limit the maximum number further because more parallel operations consume processor, real storage, and I/O resources. If resource consumption is high in your parallelism environment, use the MAX DEGREE field on installation panel DSNTIP4 to explicitly limit the maximum number of parallel operations that DB2 generates, as explain in “Enabling parallel processing.”

Enabling parallel processing

Queries can only take advantage of parallelism if you enable parallel processing. Use the following actions to enable parallel processing:

- For **static SQL**, specify DEGREE(ANY) on BIND or REBIND. This bind option affects static SQL only and does not enable parallelism for dynamic statements.
- For **dynamic SQL**, set the CURRENT DEGREE special register to 'ANY'. Setting the special register affects dynamic statements only. It will have no effect on your static SQL statements. You should also make sure that parallelism is not disabled for your plan, package, or authorization ID in the RLST. You can set the special register with the following SQL statement:

```
SET CURRENT DEGREE='ANY';
```

You can also change the special register default from 1 to ANY for the entire DB2 subsystem by modifying the CURRENT DEGREE field on installation panel DSNTIP4.

- If you bind with isolation CS, choose also the option CURRENTDATA(NO), if possible. This option can improve performance in general, but it also ensures that DB2 will consider parallelism for ambiguous cursors. If you bind with CURRENTDATA(YES) and DB2 cannot tell if the cursor is read-only, DB2 does not consider parallelism. When a cursor is read-only, it is recommended that you specify the FOR FETCH ONLY or FOR READ ONLY clause on the DECLARE CURSOR statement to explicitly indicate that the cursor is read-only.
- The virtual buffer pool parallel sequential threshold (VPPSEQT) value must be large enough to provide adequate buffer pool space for parallel processing. For a description of buffer pools and thresholds, see Part 5 (Volume 2) of *DB2 Administration Guide*.

If you enable parallel processing when DB2 estimates a given query's I/O and central processor cost is high, multiple parallel tasks can be activated if DB2 estimates that elapsed time can be reduced by doing so.

Recommendation: For parallel sorts, allocate sufficient work files to maintain performance.

Special requirements for CP parallelism: DB2 must be running on a central processor complex that contains two or more tightly coupled processors (sometimes called central processors, or CPs). If only one CP is online when the query is bound, DB2 considers only parallel I/O operations.

DB2 also considers only parallel I/O operations if you declare a cursor WITH HOLD and bind with isolation RR or RS. For more restrictions on parallelism, see Table 128.

For complex queries, run the query in parallel within a member of a data sharing group. With Sysplex query parallelism, use the power of the data sharing group to process individual complex queries on many members of the data sharing group. For more information about how you can use the power of the data sharing group to run complex queries, see Chapter 6 of *DB2 Data Sharing: Planning and Administration*.

Limiting the degree of parallelism: If you want to limit the maximum number of parallel tasks that DB2 generates, you can use the MAX DEGREE field on installation panel DSNTIP4. Changing MAX DEGREE, however, is not the way to turn parallelism off. You use the DEGREE bind parameter or CURRENT DEGREE special register to turn parallelism off.

When parallelism is not used

Parallelism is not used for all queries; for some access paths, it doesn't make sense to incur parallelism overhead. For example, if you are selecting from a temporary table, parallelism is not used. Check Table 128 to determine whether your query uses any of the access paths that do not allow parallelism.

Table 128. Checklist of parallel modes and query restrictions

If query uses this...	Is parallelism allowed?			Comments
	I/O	CP	Sysplex	
Access via RID list (list prefetch and multiple index access)	Yes	Yes	No	Indicated by an "L" in the PREFETCH column of PLAN_TABLE, or an M, MX, MI, or MQ in the ACESSTYPE column of PLAN_TABLE.
Queries that return LOB values	Yes	Yes	No	
Merge scan join on more than one column	Yes	Yes	Yes	
Queries that qualify for direct row access	No	No	No	Indicated by D in the PRIMARY_ACCESS_TYPE column of PLAN_TABLE
Materialized views or materialized nested table expressions at reference time	No	No	No	
EXISTS within WHERE predicate	No	No	No	
Security label column on table	Yes	Yes	No	

DB2 avoids certain hybrid joins when parallelism is enabled: To ensure that you can take advantage of parallelism, DB2 does not pick one type of hybrid join (SORTN_JOIN=Y) when the plan or package is bound with CURRENT DEGREE=ANY or if the CURRENT DEGREE special register is set to 'ANY'.

Interpreting EXPLAIN output

To understand how DB2 plans to use parallelism, examine your PLAN_TABLE output. (Details on all columns in PLAN_TABLE are described in Table 101 on page 794.) This section describes a method for examining PLAN_TABLE columns for parallelism and gives several examples.

A method for examining PLAN_TABLE columns for parallelism

The steps for interpreting the output for parallelism are as follows:

1. Determine if DB2 plans to use parallelism:

For each query block (QBLOCKNO) in a query (QUERYNO), a non-null value in ACCESS_DEGREE or JOIN_DEGREE indicates that some degree of parallelism is planned.

2. Identify the parallel groups in the query:

All steps (PLANNO) with the same value for ACCESS_PGROUP_ID, JOIN_PGROUP_ID, SORTN_PGROUP_ID, or SORTC_PGROUP_ID indicate that a set of operations are in the same parallel group. Usually, the set of operations involves various types of join methods and sort operations. Parallel group IDs can appear in the same row of PLAN_TABLE output, or in different rows, depending on the operation being performed. The examples in "PLAN_TABLE examples showing parallelism" help clarify this concept.

3. Identify the parallelism mode:

The column PARALLELISM_MODE tells you the kind of parallelism that is planned: I for query I/O, C for query CP, and X for Sysplex query. Within a query block, you cannot have a mixture of "I" and "C" parallel modes. However, a statement that uses more than one query block, such as a UNION, can have "I" for one query block and "C" for another. You can have a mixture of "C" and "X" modes in a query block, but not in the same parallel group.

If the statement was bound while this DB2 is a member of a data sharing group, the PARALLELISM_MODE column can contain "X" even if only this one DB2 member is active. This lets DB2 take advantage of extra processing power that might be available at execution time. If other members are not available at execution time, then DB2 runs the query within the single DB2 member.

PLAN_TABLE examples showing parallelism

For these examples, the other values would not change whether the PARALLELISM_MODE is I, C, or X.

- **Example 1: single table access**

Assume that DB2 decides at bind time to initiate three concurrent requests to retrieve data from table T1. Part of PLAN_TABLE appears as shown in Table 129 on page 853. If DB2 decides not to use parallel operations for a step, ACCESS_DEGREE and ACCESS_PGROUP_ID contain null values.

Table 129. Part of PLAN_TABLE for single table access

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)

- **Example 2: nested loop join**

Consider a query that results in a series of nested loop joins for three tables, T1, T2 and T3. T1 is the outermost table, and T3 is the innermost table. DB2 decides at bind time to initiate three concurrent requests to retrieve data from each of the three tables. Each request accesses part of T1 and all of T2 and T3. For the nested loop join method with sort, all the retrievals are in the same parallel group except for star join with ACESSTYPE=T (sparse index). Part of PLAN_TABLE appears as shown in Table 130:

Table 130. Part of PLAN_TABLE for a nested loop join

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	1	3	1	3	1	(null)	(null)
T3	1	3	1	3	1	(null)	(null)

- **Example 3: merge scan join**

Consider a query that causes a merge scan join between two tables, T1 and T2. DB2 decides at bind time to initiate three concurrent requests for T1 and six concurrent requests for T2. The scan and sort of T1 occurs in one parallel group. The scan and sort of T2 occurs in another parallel group. Furthermore, the merging phase can potentially be done in parallel. Here, a third parallel group is used to initiate three concurrent requests on each intermediate sorted table. Part of PLAN_TABLE appears as shown in Table 131:

Table 131. Part of PLAN_TABLE for a merge scan join

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	d	(null)	(null)	d	(null)
T2	2	6	2	3	3	d	d

In a multi-table join, DB2 might also execute the sort for a composite that involves more than one table in a parallel task. DB2 uses a cost basis model to determine whether to use parallel sort in all cases. When DB2 decides to use parallel sort, SORTC_PGROUP_ID and SORTN_PGROUP_ID indicate the parallel group identifier. Consider a query that joins three tables, T1, T2, and T3, and uses a merge scan join between T1 and T2, and then between the composite and T3. If DB2 decides, based on the cost model, that all sorts in this query are to be performed in parallel, part of PLAN_TABLE appears as shown in Table 132 on page 854:

Table 132. Part of PLAN_TABLE for a multi-table, merge scan join

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	2	6	2	6	3	1	2
T3	2	6	4	6	5	3	4

- **Example 4: hybrid join**

Consider a query that results in a hybrid join between two tables, T1 and T2. Furthermore, T1 needs to be sorted; as a result, in PLAN_TABLE the T2 row has SORTC_JOIN=Y. DB2 decides at bind time to initiate three concurrent requests for T1 and six concurrent requests for T2. Parallel operations are used for a join through a clustered index of T2.

Because T2's RIDs can be retrieved by initiating concurrent requests on the clustered index, the joining phase is a parallel step. The retrieval of T2's RIDs and T2's rows are in the same parallel group. Part of PLAN_TABLE appears as shown in Table 133:

Table 133. Part of PLAN_TABLE for a hybrid join

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	4	6	2	6	2	1	(null)

Tuning parallel processing

Much of the information in this section applies also to Sysplex query parallelism. See Chapter 6 of *DB2 Data Sharing: Planning and Administration* for more information.

A parallel group can run at a parallel degree less than that shown in the PLAN_TABLE output. The following factors can cause a reduced degree of parallelism:

- Buffer pool availability
- Logical contention.

Consider a nested loop join. The inner table could be in a partitioned or nonpartitioned table space, but DB2 is more likely to use a parallel join operation when the outer table is partitioned.

- Physical contention
- Run-time host variables

A host variable can determine the qualifying partitions of a table for a given query. In such cases, DB2 defers the determination of the planned degree of parallelism until run time, when the host variable value is known.

- Updatable cursor

At run time, DB2 might determine that an ambiguous cursor is updatable.

- A change in the configuration of online processors

If fewer processors are online at run time, DB2 might need to reformulate the parallel degree.

Locking considerations for repeatable read applications: For CP parallelism, locks are obtained independently by each task. Be aware that this situation can possibly increase the total number of locks taken for applications that:

- Use an isolation level of repeatable read
- Use CP parallelism
- Repeatedly access the table space using a lock mode of IS without issuing COMMITs

Recommendation: As is recommended for all repeatable-read applications, issue frequent COMMITs to release the lock resources that are held. Repeatable read or read stability isolation cannot be used with Sysplex query parallelism.

Disabling query parallelism

To disable parallel operations, do any of the following actions:

- For static SQL, rebind to change the option DEGREE(ANY) to DEGREE(1). You can do this by using the DB2I panels, the DSN subcommands, or the DSNH CLIST. The default is DEGREE(1).

- For dynamic SQL, execute the following SQL statement:

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

The default value for CURRENT DEGREE is 1 unless your installation has changed the default for the CURRENT DEGREE special register.

You can use system controls to disable parallelism, as well. These are described in Part 5 (Volume 2) of *DB2 Administration Guide*.

Chapter 29. Programming for the Interactive System Productivity Facility

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). Instructions for compiling, printing, and using the application are in Part 2 of *DB2 Installation Guide*. This chapter describes how to structure applications for use with ISPF.

The following sections discuss scenarios for interaction among your program, DB2, and ISPF:

- “Using ISPF and the DSN command processor”
- “Invoking a single SQL program through ISPF and DSN” on page 858
- “Invoking multiple SQL programs through ISPF and DSN” on page 859
- “Invoking multiple SQL programs through ISPF and CAF” on page 859

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 (see “DSN command processor” on page 508) 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.

Figure 241 on page 858 shows the task control blocks that result from attaching the DSN command processor below TSO or ISPF.

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.

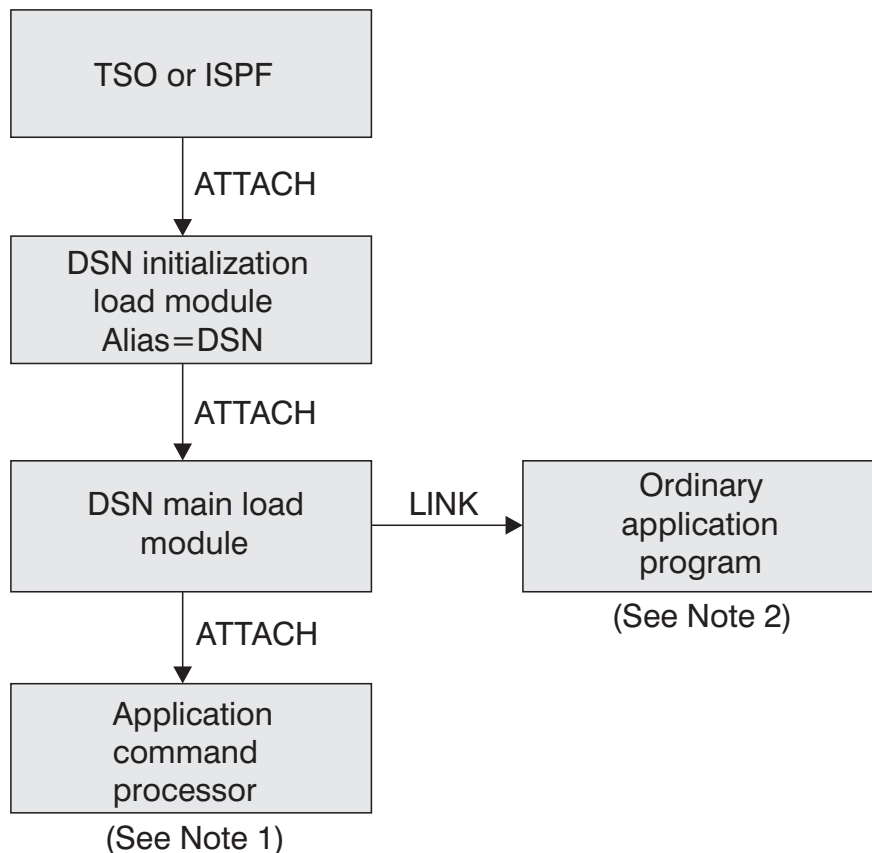


Figure 241. DSN task structure

Notes to Figure 241:

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.

Invoking a single SQL program through ISPF and DSN

With this structure, the user of your application first invokes ISPF, which displays the data and selection panels. When the user selects 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.¹ 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.

1. To achieve a more modular construction when all parts of the program use SQL, consider using packages. See Chapter 17, "Planning for DB2 program preparation," on page 381.

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 communicating through a common pool of shared variables 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.

Invoking multiple SQL programs through ISPF and CAF

You can use the call attachment facility (CAF) to call DB2; for details, see Chapter 30, “Programming for the call attachment facility,” on page 861. 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 Chapter 30, “Programming for the call attachment facility,” on page 861.

Chapter 30. Programming for the call attachment facility

An attachment facility is a part of the DB2 code that allows other programs to connect to and use DB2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls. With the call attachment facility (CAF), your application program can establish and control its own connection to DB2. Programs that run in z/OS batch, TSO foreground, and TSO background can use CAF.

It is also possible for IMS batch applications to access DB2 databases through CAF, though that method does not coordinate the commitment of work between the IMS and DB2 systems. We highly recommend that you use the DB2 DL/I batch support for IMS batch applications.

CICS application programs must use the CICS attachment facility; IMS application programs, the IMS attachment facility. Programs running in TSO foreground or TSO background can use either the DSN command processor or CAF; each has advantages and disadvantages.

Prerequisite knowledge: Analysts and programmers who consider using CAF must be familiar with z/OS concepts and facilities in the following areas:

- 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.

CAF capabilities and requirements

To decide whether to use the call attachment facility, consider the capabilities and requirements described in the following sections :

- “CAF capabilities”
- “CAF requirements” on page 863

CAF capabilities

A program using CAF can:

- 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 (though it 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 your 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 as desired.

Task capabilities

Any task in an address space can establish a connection to DB2 through CAF. There can be only one connection for each task control block (TCB). A DB2 service request issued by a program 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, nor does it 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.

Using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention. Your application design must consider that possibility.

Programming language

You can write CAF applications in assembler language, C, COBOL, Fortran, and PL/I. When choosing a language to code your application in, consider these restrictions:

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

You can find a sample assembler program (DSN8CA) and a sample COBOL program (DSN8CC) that use the call attachment facility in library *prefix.SDSNSAMP*. A PL/I application (DSN8SPM) calls DSN8CA, and a COBOL application (DSN8SCM) calls DSN8CC. For more information about the sample applications and on accessing the source code, see Appendix B, "Sample applications," on page 1013.

Tracing facility

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.

Program preparation

Preparing your application program to run in CAF is similar to preparing it 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. For examples and guidance in program preparation, see Chapter 21, “Preparing an application program to run,” on page 471.

CAF requirements

When you write programs that use CAF, be aware of the following characteristics.

Program size

The CAF code requires about 16 KB of virtual storage per address space and an additional 10 KB for each TCB using CAF.

Use of LOAD

CAF uses z/OS SVC LOAD to load two modules as part of the initialization following your first service request. Both modules are loaded into fetch-protected storage that has the job-step protection key. If your local environment intercepts and replaces the LOAD SVC, 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.

Using CAF in IMS batch

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 one in IMS, issue the SYNCPOINT command.

Run environment

Applications requesting DB2 services must adhere to several run environment characteristics. Those characteristics must be in effect regardless of the attachment facility you use. They are not unique to CAF.

- The application must be running in TCB mode. SRB mode is not supported.
- An application task cannot have any EUT 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. Therefore:
 - An application must not use CAF 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 using RRSAF.
 - An application that runs in an address space that has an RRSAF connection to DB2 cannot connect to DB2 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. This means that your CAF application cannot use DSN, and a DSN RUN subcommand cannot call your CAF application.
- The language interface module for CAF, DSNALI, is shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load CAF below the 16-MB line, you must link-edit DSNALI again.

Running DSN applications under CAF

Although doing so is not recommended, you can run existing DSN applications with CAF merely 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.

Running DSN applications with CAF is not advantageous, and the loss of DSN services can affect how well your program runs. In general, running DSN applications with CAF is not recommended unless you provide an application controller to manage the DSN application and replace any needed DSN functions. Even then, you could have to change the application to communicate connection failures to the controller correctly.

How to use CAF

To use CAF, you must first make available a load module known as the call attachment language interface, or DSNALI. For considerations for loading or link-editing this module, see “Accessing the CAF language interface” on page 867.

When the language interface is available, your program can make use of the CAF in two ways:

- Implicitly, 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 using default values for the pertinent parameters described under “Implicit connections” on page 866.
- Explicitly, by writing CALL DSNALI statements, providing the appropriate options. For the general form of the statements, see “CAF function descriptions” on page 869.

The first element of each option list is a *function*, which describes the action that you want CAF to take. For the available values of function and an approximation of their effects, see “Summary of connection functions” on page 866. The effect of any function depends in part on what functions the program has already run. Before using any function, be sure to read the description of its usage. Also read “Summary of CAF behavior” on page 881, which describes the influence of previous functions.

You might structure a CAF configuration like the one that is illustrated in Figure 242 on page 865. 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.

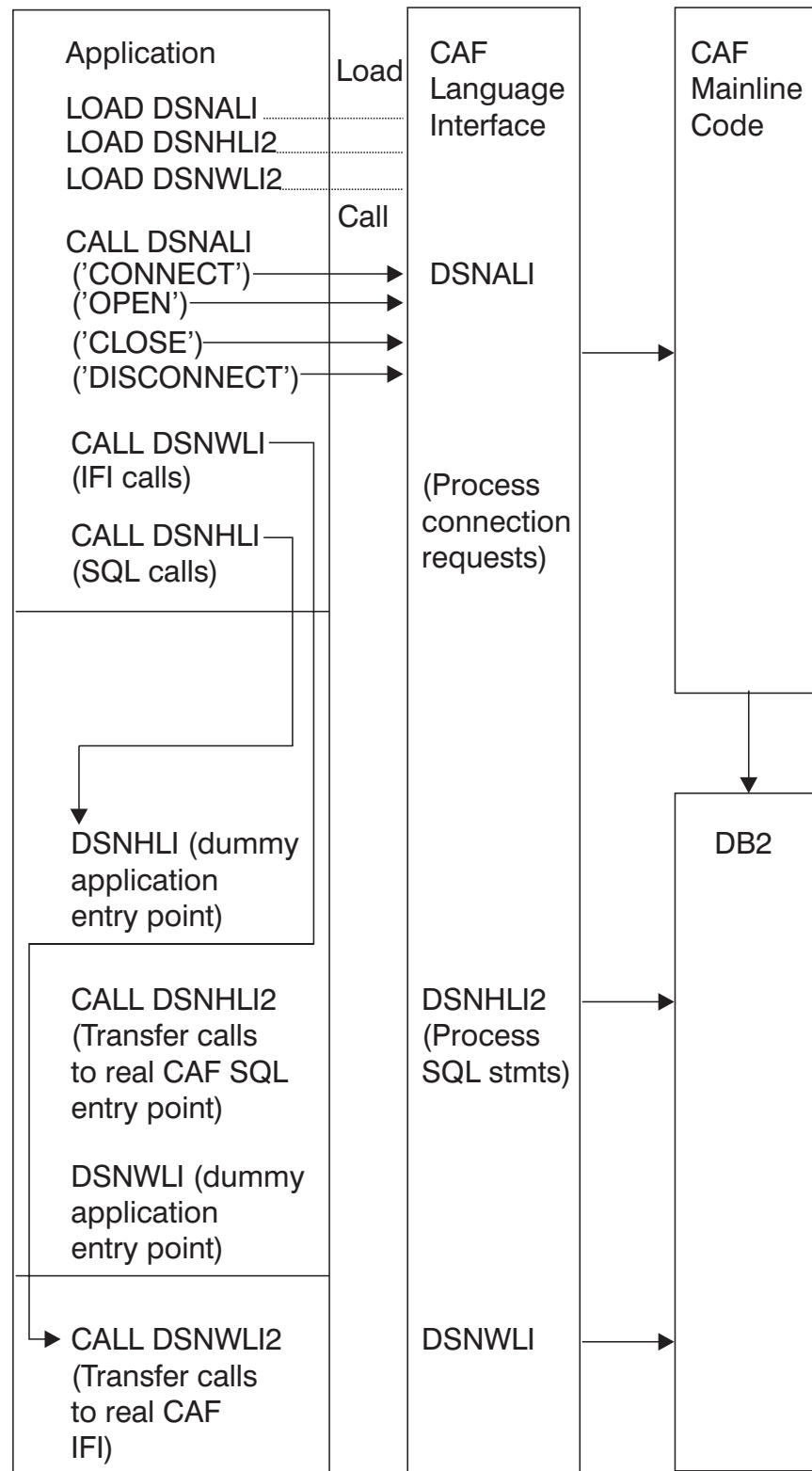


Figure 242. Sample call attachment facility configuration

The remainder of this chapter discusses:

- “Summary of connection functions” on page 866
- “Sample scenarios” on page 882
- “Exit routines from your application” on page 883

- “Error messages and dsnttrace” on page 884
- “Program examples for CAF” on page 885.

Summary of connection functions

You can use the following functions with CALL DSNALI:

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. See “CONNECT: Syntax and usage” on page 871.

OPEN

Allocates a DB2 plan. You must allocate a plan before DB2 can process SQL statements. If you did not request the CONNECT function, OPEN implicitly establishes the task, and optionally the address space, as a user of DB2. See “OPEN: Syntax and usage” on page 875.

CLOSE

Optionally commits or abends any database changes and deallocates the plan. If OPEN implicitly requests the CONNECT function, CLOSE removes the task, and possibly the address space, as a user of DB2. See “CLOSE: Syntax and usage” on page 877.

DISCONNECT

Removes the task as a user of DB2 and, if this is the last or only task in the address space with a DB2 connection, terminates the address space connection to DB2. See “DISCONNECT: Syntax and usage” on page 878.

TRANSLATE

Returns an SQLCODE and printable text in the SQLCA that describes a DB2 hexadecimal error reason code. See “TRANSLATE: Syntax and usage” on page 880. You cannot call the TRANSLATE function from the Fortran language.

Implicit connections

If you do not explicitly specify executable SQL statements in a CALL DSNALI statement of your CAF application, CAF initiates implicit CONNECT and OPEN requests to DB2. Although CAF performs these connection requests using the following default values, the 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 specified in the module DSNHDECP. CAF uses the installation default DSNHDECP, unless your own DSNHDECP is in a library in a STEPLIB or JOBLIB concatenation, or in the link list. In a data sharing group, the default subsystem name is the group attachment name.

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. If your program can make its first SQL call from different modules with different DBRMs, you cannot use a default plan name; you must use an explicit call using the OPEN function.

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

Different types of implicit connections exist. The simplest is for application to run neither CONNECT nor OPEN. You can also use CONNECT only or OPEN only. Each of these implicitly connects your application to DB2. To terminate an implicit connection, you must use the proper calls. See Table 141 on page 882 for details.

Your application program must successfully connect, either implicitly or explicitly, to DB2 before it can execute any SQL calls to the CAF DSNHLI entry point. Therefore, the application program must first determine the success or failure of all implicit connection requests.

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 -991. The application program should examine the return and reason codes immediately after the first executable SQL statement within the application program. Two ways to do this are to:

- Examine registers 0 and 15 directly.
- Examine 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.

Accessing the CAF language interface

Part of the call attachment facility is a DB2 load module, DSNALI, known as the call attachment facility language interface. DSNALI has the alias names DSNHLI2 and DSNWLI2. The module has five entry points: DSNALI, DSNHLI, DSNHLI2, DSNWLI, and DSNWLI2:

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

You can access the DSNALI module by either explicitly issuing LOAD requests when your program runs, or by including the module in your load module when you link-edit your program. There are advantages and disadvantages to each approach.

Explicit load of DSNALI

To load DSNALI, 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.

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.

You must indicate to DB2 which entry point to use. You can do this in one of two ways:

- Specify the precompiler option ATTACH(CAF).
This causes DB2 to generate calls that specify entry point DSNHLI2. 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 and is independent of the different DB2 attachment facilities. When the calls generated by the DB2 precompiler pass control to DSNHLI, your code corresponding to the dummy entry point must preserve the option list passed in R1 and call DSNHLI2 specifying the same option list. For a coding example of a dummy DSNHLI entry point, see “Using dummy entry point DSNHLI for CAF” on page 890.

Link-editing DSNALI

You can include the CAF language interface module 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 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 DB2LIB(DSNALI). By coding these options, you avoid inadvertently picking up the wrong language interface module.

If you do not need explicit calls to DSNALI for CAF functions, including DSNALI in your load module has some advantages. When you include DSNALI during the link-edit, you need not code the previously described 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.

General properties of CAF connections

Some of the basic properties of the connection the call attachment facility makes with DB2 are:

- **Connection name:** DB2CALL. You can use the DISPLAY THREAD command to list CAF applications having the connection name DB2CALL.
- **Connection type:** BATCH. BATCH connections use a single phase commit process coordinated by DB2. Application programs can also use the SQL COMMIT and ROLLBACK statements.
- **Authorization IDs:** DB2 establishes authorization identifiers for each task's connection when it processes the connection for each task. For the BATCH connection type, DB2 creates a list of authorization IDs based on the authorization ID associated with the address space and the list is the *same* for every task. A location can provide a DB2 connection authorization exit routine to change the list of IDs. For information about authorization IDs and the connection authorization exit routine, see Appendix B (Volume 2) of *DB2 Administration Guide*.
- **Scope:** The CAF processes connections as if each task is entirely isolated. When a task requests a function, the CAF passes the functions to DB2, 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.

Task termination

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.

DB2 abend

If DB2 abends 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.

CAF function descriptions

To code CAF functions in C, COBOL, Fortran, or PL/I, follow the individual language's rules for making calls to assembler routines. Specify the return code and reason code parameters in the parameter list for each CAF call.

A description of the call attach register and parameter list conventions for assembler language follow. Following it, the syntax description of specific functions describe the parameters for those particular functions.

Register conventions

If you do not specify the return code and reason code parameters in your CAF calls, CAF puts a return code in register 15 and a reason code in register 0. CAF also supports high-level languages that cannot interrogate individual registers. See Figure 243 on page 870 and the discussion following it for more information. The contents of registers 2 through 14 are preserved across calls. You must conform to the standard calling conventions listed in Table 134:

Table 134. Standard usage of registers R1 and R13-R15

Register	Usage
R1	Parameter list pointer (for details, see "Call DSNALI parameter list")
R13	Address of caller's save area
R14	Caller's return address
R15	CAF entry point address

Call DSNALI parameter list

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. Figure 243 on page 870 shows a sample parameter list structure for the CONNECT function.

When you code CALL DSNALI statements, you must specify all parameters that come before the return code parameter. You cannot omit any of those parameters by coding zeros or blanks. There are no defaults for those parameters for explicit connection service requests. Defaults are provided only for implicit connections.

All parameters starting with the return code parameter are optional.

For all languages except assembler language, code zero for a parameter in the CALL DSNALI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, suppose you are

coding a CONNECT call in a COBOL program. You want to specify all parameters except the return code parameter. Write the call in this way:

```
CALL 'DSNALI' USING FUNCTN SSID TECB SECB RIBPTR
    BY CONTENT ZERO BY REFERENCE REASCODE SRDURA EIBPTR.
```

For an assembler language call, code a comma for a parameter in the CALL DSNALI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, code a CONNECT call like this to specify all optional parameters except the return code parameter:

```
CALL DSNALI, (FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,,REASCODE,SRDURA,EIBPTR,GROUPOVERRIDE)
```

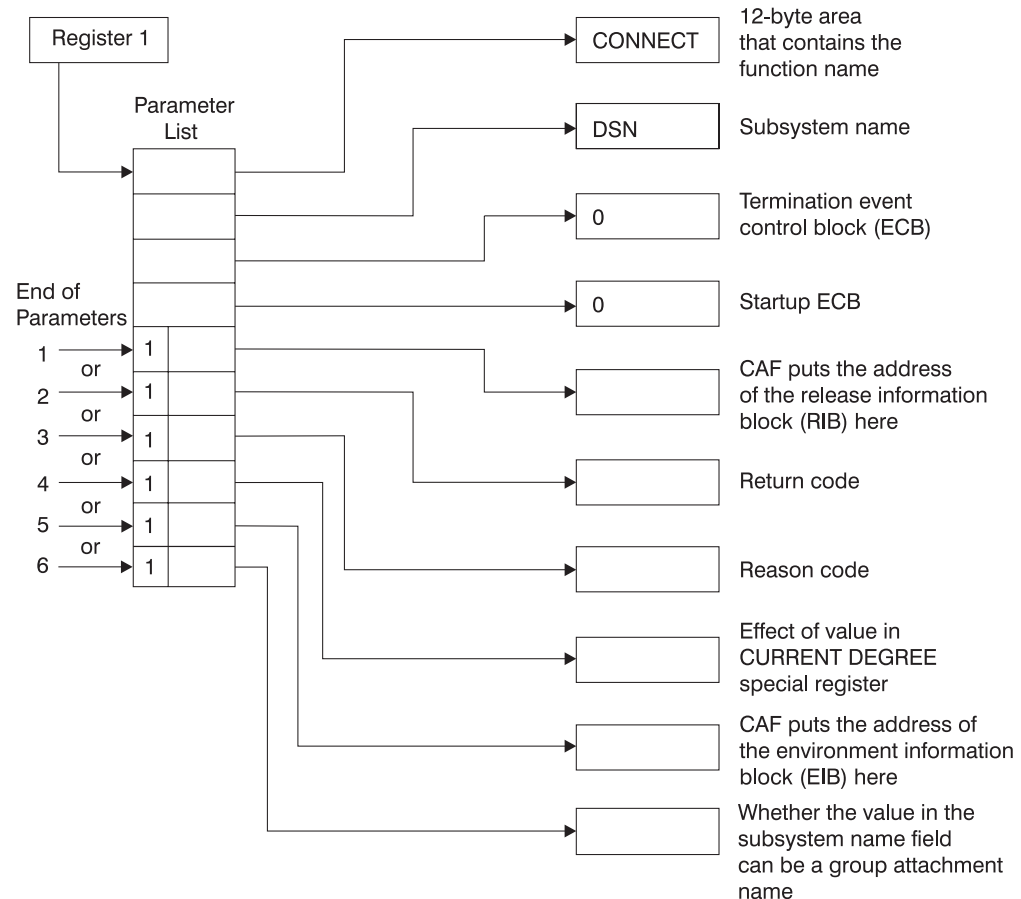


Figure 243. The parameter list for a CONNECT call

Figure 243 illustrates how you can use the indicator *end of parameter list* to control the return codes and reason code fields following a CAF CONNECT call. Each of the six illustrated termination points applies to all CAF 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 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 at this point permits 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 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. To do this, specify a comma as a place-holder for the omitted 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. To do this, specify commas as place-holders for the omitted 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. To do this, specify commas as place-holders for the omitted parameters.

6. Terminates the parameter list after the parameter *groupoverride*.

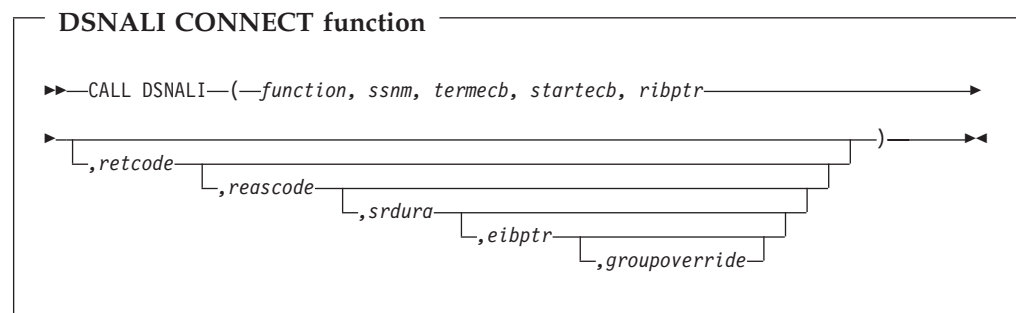
If you code your CAF application in assembler language, you can specify this parameter and omit the *retcode*, *reascode*, *srdura*, or *eibptr* parameters. To do this, specify commas as place-holders for the omitted parameters.

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.

CONNECT: Syntax and usage

CONNECT initializes a connection to DB2. You should not confuse the CONNECT function of the call attachment facility with the DB2 CONNECT statement that accesses a remote location within DB2.

“DSNALI CONNECT function” shows the syntax for the CONNECT function.



Parameters point to the following areas:

function

A 12-byte area containing *CONNECT* followed by five blanks.

ssnm

A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group) to which the connection is made.

If you specify the group attachment name, the program connects to the DB2 on the z/OS system on which the program is running. When you specify a group

attachment name and a startup ECB, DB2 ignores the startup ECB. If you need to use a startup ECB, specify a subsystem name, rather than a group attachment name. That subsystem name must be different from the group attachment name.

If your *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

termecb

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. It indicates the type of termination by a POST code, as shown in Table 135:

Table 135. POST codes and related termination types

POST code	Termination type
8	QUIESCE
12	FORCE
16	ABTERM

Before you check *termecb* in your CAF application program, first check the return code and reason code from the CONNECT call to ensure that the call completed successfully. See "Checking return codes and reason codes for CAF" on page 888 for more information.

startecb

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/DB2 reason codes before issuing a WAIT on this ECB.

If *ssnm* is a group 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 field RIBREL. You can determine the modification level within the release level by examining fields RIBCNUMB and RIBCINFO. If the value in RIBCNUMB is greater than zero, check RIBCINFO 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 not specified, CAF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

srdura

A 10-byte area containing the string 'SRDURA(CD)'. This field is optional. If it is provided, the value in the CURRENT DEGREE special register stays in effect from CONNECT until DISCONNECT. If it is not provided, the value in the CURRENT DEGREE special register stays in effect from OPEN until CLOSE. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

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, then 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 below 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 the return code, reason code, and *srdura* parameters. In assembler language, you can omit the return code, reason code, and *srdura* parameters by specifying commas as place-holders.

Macro DSNDEIB maps the EIB. It can be found in *prefix.SDSNMACS(DSNDEIB)*.

groupoverride

An 8-byte area that the application provides. This field is optional. If this field is provided, it contains the string '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 name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code, reason code, *srdura*, and *eibptr* parameters. In assembler language, you can omit the return code, reason code, *srdura*, and *eibptr* parameters by specifying commas as place-holders.

Usage: CONNECT 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 subsystem named by *ssnm*, CONNECT also initializes the address space for communication to the DB2 address spaces. CONNECT establishes the address space's cross memory authorization to DB2 and builds address space control blocks.

In a data sharing environment, use the *groupoverride* parameter on a CONNECT call 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 name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

Using a CONNECT call is optional. The first request from a task, either OPEN, 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 execute CLOSE or when the task terminates.

Establishing task and address space level connections is essentially an initialization function and involves significant overhead. If you use CONNECT to establish a task connection explicitly, it terminates when you use DISCONNECT or when the task terminates. The explicit connection minimizes the overhead by ensuring that the connection to DB2 remains after CLOSE deallocates a plan.

You can run CONNECT from any or all tasks in the address space, but the address space level is initialized only once when the first task connects.

If a task does not issue an explicit CONNECT or OPEN, the implicit connection from the first SQL or IFI call specifies a default DB2 subsystem name. A systems programmer or administrator determines the default subsystem name when installing DB2. Be certain that you know what the default name is and that it names the specific DB2 subsystem you want to use.

Practically speaking, you must 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.

Use CONNECT when:

- You need to specify a particular (non-default) subsystem name (*ssnm*).
- 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 (*termecb*), or the DB2 release level.
- Multiple tasks in the address space will be opening and closing plans.
- A single task in the address space will be opening and closing plans more than once.

The other parameters of CONNECT enable the caller to learn:

- That the operator has issued a STOP DB2 command. When this happens, 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 happens, DB2 posts the termination ECB, *termecb*.
- That DB2 is available again (after a connection attempt that failed because DB2 was down). Wait on or look at the startup ECB, *startech*. DB2 ignores this ECB if it was active at the time of the CONNECT request, or if the CONNECT request was to a group attachment name.
- The current release level of DB2. Access the RIBREL field in the release information block (RIB).

Do not issue CONNECT requests from a TCB that already has an active DB2 connection. (See “Summary of CAF behavior” on page 881 and “Error messages and dsnttrace” on page 884 for more information about CAF errors.)

Table 136 shows a CONNECT call in each language.

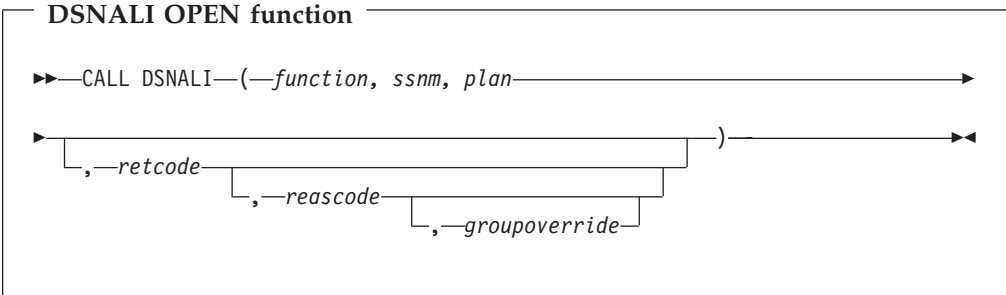
Table 136. Examples of CAF CONNECT calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA, EIBPTR, GRPOVER)
C	fnret=dsnali(&functn[0],&ssid[0], &tecb, &secb,&ribptr,&retcode, &reascodes, &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	CALL DSNALI(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA, EIBPTR,GRPOVER);
Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:	
C	#pragma linkage(dsnali, OS)
C++	extern "OS" { int DSNALI(char * functn, ...); }
PL/I	DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

OPEN: Syntax and usage

OPEN allocates resources to run the specified plan. Optionally, OPEN requests a DB2 connection for the issuing task.

“DSNALI OPEN function” shows the syntax for the OPEN function.



Parameters point to the following areas:

function

A 12-byte area containing the word OPEN followed by eight blanks.

ssnm

A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group). Optionally, OPEN establishes a connection from *ssnm* to the named DB2 subsystem. If your *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 not specified, CAF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

groupoverride

An 8-byte area that the application provides. This field is optional. If this field is provided, it contains the string '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 name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

Usage: OPEN allocates DB2 resources needed to run the plan or issue IFI requests. If the requesting task does not already have a connection to the named DB2 subsystem, then OPEN establishes it.

OPEN allocates the plan to the DB2 subsystem named in *ssnm*. The *ssnm* parameter, like the others, is required, even if the task issues a CONNECT call. If a task issues CONNECT followed by OPEN, then the subsystem names for both calls must be the same.

In a data sharing environment, use the *groupoverride* parameter on an OPEN call 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 name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

The use of OPEN is optional. If you do not use OPEN, the action of OPEN occurs on the first SQL or IFI call from the task, using the defaults listed under "Implicit connections" on page 866.

Do not use OPEN if the task already has a plan allocated.

Table 137 shows an OPEN call in each language.

Table 137. Examples of CAF OPEN calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)

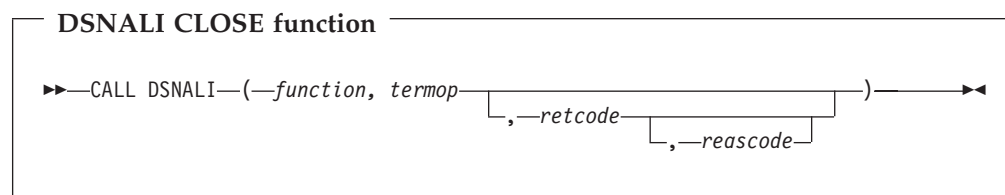
Table 137. Examples of CAF OPEN calls (continued)

Language	Call example
C	fnret=dsnali(&functn[0],&ssid[0], &planname[0],&retcode, &reascde,&grpover[0]);
COBOL	CALL 'DSNALI' USING FUNCTN SSID PLANNAME RETCODE REASCODE GRPOVER.
Fortran	CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)
PL/I	CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER);
Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:	
C	<pre>#pragma linkage(dsnali, OS)</pre>
C++	<pre>extern "OS" { int DSNALI(char * functn, ...); }</pre>
PL/I	<pre>DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);</pre>

CLOSE: Syntax and usage

CLOSE deallocates the plan and optionally disconnects the task, and possibly the address space, from DB2.

"DSNALI CLOSE function" shows the syntax for the CLOSE function.



Parameters point to the following areas:

function

A 12-byte area containing the word CLOSE followed by seven blanks.

termop

A 4-byte terminate option, with one of these values:

SYNC

Commit any modified data

ABRT

Roll back data to the previous commit point.

retcode

A 4-byte area in which CAF should place the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascodes

A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

Usage: CLOSE deallocates the created plan either explicitly using OPEN or implicitly at the first SQL call.

If you did not issue a CONNECT for the task, CLOSE also 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 created for the address space and removes the cross memory authorization.

Do not use CLOSE when your current task does not have a plan allocated.

Using CLOSE is optional. If you omit it, DB2 performs the same actions when your task terminates, using the SYNC parameter if termination is normal and the ABRT parameter if termination is abnormal. (The function is an implicit CLOSE.) If the objective is to shut down your application, you can improve shut down performance by using CLOSE explicitly before the task terminates.

If you want to use a new plan, you must issue an explicit CLOSE, followed by an OPEN, specifying the new plan name.

If DB2 terminates, a task that did not issue CONNECT should explicitly issue CLOSE, so that CAF can reset its control blocks to allow for future connections. This CLOSE returns the reset accomplished return code (+004) and reason code X'00C10824'. If you omit CLOSE, then 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 CAF error message DSNAB201I or DSNAB202I, depending on what your application tried to do. The task must then issue CLOSE before it can reconnect to DB2.

A task that issued CONNECT explicitly should issue DISCONNECT to cause CAF to reset its control blocks when DB2 terminates. In this case, CLOSE is not necessary.

Table 138 shows a CLOSE call in each language.

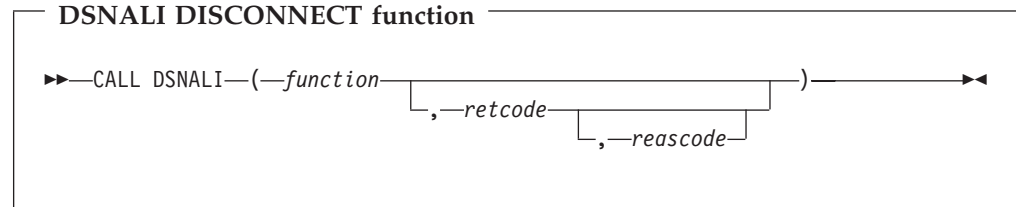
Table 138. Examples of CAF CLOSE calls

Language	Call example
Assembler	CALL DSNALI(FUNCTN,TERMOP,RETCODE, REASCODE)
C	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	CALL DSNALI(FUNCTN,TERMOP, RETCODE,REASCODE);
Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:	
C	#pragma linkage(dsnali, OS)
C++	extern "OS" { int DSNALI(char * functn, ...); }
PL/I	DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

DISCONNECT: Syntax and usage

DISCONNECT terminates a connection to DB2.

“DSNALI DISCONNECT function” shows the syntax for the DISCONNECT function.



The single parameter points to the following area:

function

A 12-byte area containing the word DISCONNECT followed by two blanks.

retcode

A 4-byte area in which CAF places the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascodes

A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

Usage: 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 created for the address space and removes the cross memory authorization.

Only those tasks that issued CONNECT explicitly can issue DISCONNECT. If CONNECT was not used, then DISCONNECT causes an error.

If an OPEN is in effect when the DISCONNECT is issued (that is, a plan is allocated), CAF issues an implicit CLOSE with the SYNC parameter.

Using DISCONNECT is optional. Without it, DB2 performs the same functions when the task terminates. (The function is an implicit DISCONNECT.) If the objective is to shut down your application, you can improve shut down performance if you request DISCONNECT explicitly before the task terminates.

If DB2 terminates, a task that issued CONNECT must issue DISCONNECT to reset the CAF control blocks. The function returns the *reset accomplished* return codes and reason codes (+004 and X'00C10824'), and ensures that future connection requests from the task work when DB2 is back on line.

A task that did not issue `CONNECT` explicitly must issue `CLOSE` to reset the CAF control blocks when DB2 terminates.

Table 139 shows a DISCONNECT call in each language.

Table 139. Examples of CAF DISCONNECT calls

Language	Call example
Assembler	CALL DSNALI(,FUNCTN,RETCODE,REASCODE)

Table 139. Examples of CAF DISCONNECT calls (continued)

Language	Call example
C	<code>fnret=dsnali(&functn[0], &retcode, &reascode);</code>
COBOL	<code>CALL 'DSNALI' USING FUNCTN RETCODE REASCODE.</code>
Fortran	<code>CALL DSNALI(FUNCTN,RETCODE,REASCODE)</code>
PL/I	<code>CALL DSNALI(FUNCTN,RETCODE,REASCODE);</code>

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

```

C      #pragma linkage(dsnali, 05)

C++
extern "OS" {
    int DSNALI(
        char * functn,
        ...); }

PL/I   DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

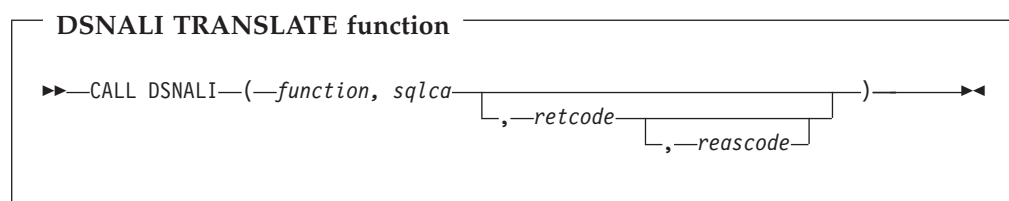
```

TRANSLATE: Syntax and usage

You can use TRANSLATE to convert a DB2 hexadecimal error reason code into a signed integer SQLCODE and a printable error message text. The SQLCODE and message text appear in the caller's SQLCA. You cannot call the TRANSLATE function from the Fortran language.

TRANSLATE is useful only after an OPEN fails, and then only if you used an explicit CONNECT before the OPEN request. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.

"DSNALI TRANSLATE function" shows the syntax for the TRANSLATE function.



Parameters point to the following areas:

function

A 12-byte area containing the word TRANSLATE followed by three blanks.

sqlca

The program's SQL communication area (SQLCA).

retcode

A 4-byte area in which CAF places the return code.

This field is optional. If not specified, 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. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

Usage: Use TRANSLATE to get a corresponding SQL error code and message text for the DB2 error reason codes that CAF returns in register 0 following an OPEN service request. DB2 places the information into the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

The TRANSLATE function can translate those codes beginning with X'00F3', but it does not translate CAF reason codes beginning with X'00C1'. If you receive error reason code X'00F30040' (*resource unavailable*) after an OPEN request, TRANSLATE returns the name of the unavailable database object in the last 44 characters of field SQLERRM. If the DB2 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 which case, register 0 is set to X'C10205' and register 15 to 200.

Table 140 shows a TRANSLATE call in each language.

Table 140. Examples of CAF TRANSLATE calls

Language	Call example
Assembler	CALL DSNALI(FUNCTN,SQLCA,RETCODE, REASCODE)
C	fnret=dsnali(&functn[0], &sqlca, &retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN SQLCA RETCODE REASCODE.
PL/I	CALL DSNALI(FUNCTN,SQLCA,RETCODE, REASCODE);
Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:	
C	<pre>#pragma linkage(dsnali, 0S)</pre>
C++	<pre>extern "OS" { int DSNALI(char * functn, ...); }</pre>
PL/I	<pre>DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);</pre>

Summary of CAF behavior

Table 141 on page 882 summarizes CAF behavior after various inputs from application programs. Use it to help plan the calls your program makes, and to help understand where CAF errors can occur. Careful use of this table can avoid major structural problems in your application.

In the table, an error shows as *Error nnn*. The corresponding reason code is X'00C10'nnn; the message number is DSNAnnnI or DSNAnnnE. For a list of reason codes, see "CAF return codes and reason codes" on page 884.

Table 141. 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	Error 204	Error 205
CONNECT	Error 201	OPEN	OPEN, followed by the SQL or IFI call	Error 203	DISCONNECT	TRANSLATE
CONNECT followed by OPEN	Error 201	Error 202	The SQL or IFI call	CLOSE ¹	DISCONNECT	TRANSLATE
CONNECT followed by SQL or IFI call	Error 201	Error 202	The SQL or IFI call	CLOSE ¹	DISCONNECT	TRANSLATE
OPEN	Error 201	Error 202	The SQL or IFI call	CLOSE ²	Error 204	TRANSLATE
SQL or IFI call	Error 201	Error 202	The SQL or IFI call	CLOSE ²	Error 204	TRANSLATE ³

Notes:

1. The task and address space connections remain active. If CLOSE fails because DB2 was down, then the CAF control blocks are reset, the function produces return code 4 and reason code XX'00C10824', and CAF is ready for more connection requests when DB2 is again on line.
2. 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.

Table 141 uses the following conventions:

- The top row lists the possible CAF functions that programs can use as their call.
- The first column lists the task's most recent history of connection requests. For example, CONNECT followed by OPEN 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 an implicit CONNECT and OPEN function is performed, followed by the SQL function.

Sample scenarios

This section shows sample scenarios for connecting tasks to DB2.

A single task with implicit connections

The simplest connection scenario is a single task making calls to DB2, using no explicit CALL DSNALI statements. The task implicitly connects to the default subsystem name, using the default plan name.

When the task terminates:

- Any database changes are committed (if termination was normal) or rolled back (if termination was abnormal).
- 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

A more complex scenario, but still with a single task, is this:

```
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 one and only one DB2 subsystem at any point in time. A CAF error occurs if the subsystem name on OPEN does not match the one on CONNECT. To switch to a different subsystem, the application must disconnect from the current subsystem, then issue a connect request specifying a new subsystem name.

Several tasks

In this scenario, multiple tasks within the address space are using DB2 services. Each task must explicitly specify the same subsystem name on either the CONNECT or 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			

Exit routines from your application

You can provide exit routines from your application for the purposes described in the following text.

Attention exit routines

An attention exit routine enables you to regain control from DB2, during long-running or erroneous requests, by detaching the TCB currently waiting on an SQL or IFI request to complete. DB2 detects the abend caused by DETACH and performs termination processing (including ROLLBACK) for that task.

The call attachment facility has no attention exit routines. You can provide your own if necessary. However, DB2 uses enabled unlocked task (EUT) functional recovery routines (FRRs), so if you request attention while DB2 code is running, your routine may not get control.

Recovery routines

The call attachment facility has no abend recovery routines.

Your program can provide an abend exit routine. It must use tracking indicators to determine if an abend occurred during DB2 processing. If an abend occurs while DB2 has control, you have these choices:

- Allow task termination to complete. Do not retry the program. DB2 detects task termination and terminates the thread with the ABRT parameter. You lose all database changes back to the last SYNC or COMMIT point.

This is the only action that you can take for abends that CANCEL or DETACH cause. 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, a return code of +256 and a reason code of X'00F30083' occurs.

- In an ESTAE routine, issue CLOSE with the ABRT parameter followed by DISCONNECT. The ESTAE exit routine can retry so that you do not need to reinstate the application task.

Standard z/OS functional recovery routines (FRRs) can cover only code running in service request block (SRB) mode. Because DB2 does not support calls from SRB mode routines, you can use only enabled unlocked task (EUT) FRRs in your routines that call DB2.

Do not have an EUT FRR active when using CAF, processing SQL requests, or calling IFI.

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

With z/OS, if you have an active EUT FRR, all DB2 requests fail, including the initial CONNECT or OPEN. 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.

Error messages and dsnttrace

CAF produces no error messages unless you allocate a DSNTRACE data set. If you allocate a DSNTRACE data set either dynamically or by including a //DSNTRACE DD statement in your JCL, CAF writes diagnostic trace message to that data set. You can refer to "Sample JCL for using CAF" on page 886 for sample JCL that allocates a DSNTRACE data set. The trace message numbers contain the last three digits of the reason codes.

CAF return codes and reason codes

CAF provides the return codes and reason codes either to the corresponding parameters named in a CAF call or, if you choose not to use those parameters, to registers 15 and 0. Detailed explanations of the reason codes appear in *DB2 Codes*.

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 more information about the codes, along with abend and subsystem termination reason codes issued by other parts of subsystem support, see Part 3 of *DB2 Codes*.

For SQL calls, CAF returns standard SQLCODEs in the SQLCA. See Part 2 of *DB2 Codes* for a list of those return codes and their meanings. CAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA).

Table 142 shows the CAF return codes and reason codes.

Table 142. CAF return codes and reason codes

Return code	Reason code	Explanation
0	X'00000000'	Successful completion.
4	X'00C10824'	CAF reset complete. Ready to make a new connection.
# 8 #	X'00C10831'	Release level mismatch between DB2 and the and the call attachment facility code.
200 ¹	X'00C10201'	Received a second CONNECT from the same TCB. The first CONNECT could have been implicit or explicit.
200 ¹	X'00C10202'	Received a second OPEN from the same TCB. The first OPEN could have been implicit or explicit.
200 ¹	X'00C10203'	CLOSE issued when there was no active OPEN.
200 ¹	X'00C10204'	DISCONNECT issued when there was no active CONNECT, or the AXSET macro was issued between CONNECT and DISCONNECT.
200 ¹	X'00C10205'	TRANSLATE issued when there was no connection to DB2.
200 ¹	X'00C10206'	Wrong number of parameters 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.

Notes:

1. A CAF error probably caused by errors in the parameter lists coming from 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. For an explanation of the abend reason codes, see Part 3 of *DB2 Codes*. If tracing is on, a descriptive message is written to the DSNTRACE data set just before the abend.

Subsystem support subcomponent codes (X'00F3'): 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 more information about the codes, along with abend and subsystem termination reason codes that are issued by other parts of subsystem support, see Part 3 of *DB2 Codes*.

Program examples for CAF

The following pages contain sample JCL and assembler programs that access the call attachment facility (CAF).

Sample JCL for using CAF

The sample JCL that follows is a model for using CAF in a batch (non-TSO) environment. The DSNTRACE statement shown 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=*
```

Sample assembler code for using CAF

The following sections show parts of a sample assembler program using the call attachment facility. It demonstrates the basic techniques for making CAF calls but does 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 code that follows.

These code segments assume the existence of a WRITE macro. Anywhere you find this macro in the code is a good place for you to 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.

Loading and deleting the CAF language interface

The following code segment shows how an application can load entry points DSNALI and DSNHLI2 for the call attachment 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
```

Connecting to DB2 for CAF

Figure 244 on page 887 shows how to issue explicit requests for certain actions (CONNECT, OPEN, CLOSE, DISCONNECT, and TRANSLATE), using 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
      WRITE 'The current DB2 release level is' RIBREL

***** 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

```

Figure 244. CHEKCODE Subroutine for connecting to DB2

The code does not show a task that waits on the DB2 termination ECB. If you like, you can code such a task and use 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.

On entry, the code assumes that certain variables are already 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 on the OPEN call.

- CONTROL** Used to shut down processing because of unsatisfactory return or reason codes. Subroutine CHEKCODE sets CONTROL.
- CAFCALL** List-form parameter area for the CALL macro.

Checking return codes and reason codes for CAF

Figure 245 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)          /* Termination ECB was not POSTed */
# *     WHEN (0) ;                /* Look at the return code */
# *     WHEN (4) ;                /* Do nothing; everything is OK */
# *     SELECT (REASCODE)         /* Warning */
# *     WHEN ('00C10824'X)        /* Look at the reason code */
# *     CONTROL = 'RESTART'      /* Ready for another CAF call */
# *     CONTROL = 'RESTART'      /* Start over, from the top */
# *   OTHERWISE
# *     WRITE 'Found unexpected R0 when R15 was 4'
# *     CONTROL = 'SHUTDOWN'
# *   END INNER-SELECT
# *   WHEN (8,12)                /* Connection failure */
# *     SELECT (REASCODE)         /* Look at the reason code */
# *     WHEN ('00C10831'X)        /* DB2 / CAF release level mismatch*/
# *     WRITE 'Found a mismatch between DB2 and CAF release levels'
# *     WHEN ('00F30002'X,        /* These mean that DB2 is down but */
# *           '00F30012'X)        /* will POST SECB when up again */
# *     DO
# *       WRITE 'DB2 is unavailable. I'll tell you when it's up.'
# *       WAIT SECB              /* Wait for DB2 to come up */
# *       WRITE 'DB2 is now available.'
# *     END
# *   /* *****
# *   /* Insert tests for other DB2 connection failures here. */
# *   /* CAF External Specification lists other codes you can */
# *   /* receive. Handle them in whatever way is appropriate */
# *   /* for your application. */
# *   /* *****
# *   OTHERWISE                  /* Found a code we're not ready for*/
# *     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 dataset'
# *   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
```

Figure 245. Subroutine to check return codes from CAF and DB2, in assembler (Part 1 of 3)

```

# # *****
# # * 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

```

Figure 245. Subroutine to check return codes from CAF and DB2, in assembler (Part 2 of 3)

```

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 DSNFIAR
*      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

```

Figure 245. Subroutine to check return codes from CAF and DB2, in assembler (Part 3 of 3)

Using dummy entry point DSNHLI for CAF

Each of the four DB2 attachment facilities contains an entry point named DSNHLI. When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL statements result in BALR instructions to DSNHLI in your program. To find the correct DSNHLI entry point without including DSNALI in your load module, code a subroutine with entry point DSNHLI that passes control to entry point DSNHLI2 in the DSNALI module. DSNHLI2 is unique to DSNALI and is at the same location in DSNALI as DSNHLI. 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 example that follows, 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)
```

Variable declarations for CAF

Figure 246 on page 892 shows declarations for some of the variables used in the previous subroutines.

Chapter 31. Programming for the Resource Recovery Services attachment facility

An application program can use the Resource Recovery Services attachment facility (RRSAF) to connect to and use DB2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls. Programs that run in z/OS batch, TSO foreground, and TSO background can use RRSAF.

RRSAF uses z/OS Transaction Management and Recoverable Resource Manager Services (z/OS RRS). With RRSAF, you can coordinate DB2 updates with updates made by all other resource managers that also use z/OS RRS in an z/OS system.

Prerequisite knowledge: Before you consider using RRSAF, you must be familiar with the following z/OS topics:

- 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.

RRSAF capabilities and requirements

To decide whether to use RRSAF, consider the following topics:

- “RRSAF capabilities”
- “RRSAF requirements” on page 894

RRSAF capabilities

An application program using RRSAF can:

- 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 an end 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 desired.

Task capabilities

Any task in an address space can establish a connection to DB2 through RRSF.

Number of connections to DB2: Each task control block (TCB) can have only one connection to DB2. A DB2 service request 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.

Using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention. Consider this when you write your application program.

Specifying a plan for a 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.

Providing attention processing exits and recovery routines: RRSF does not generate task structures, and it 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.

Programming language

You can write RRSF applications in assembler language, C, COBOL, Fortran, and PL/I. When choosing a language to code your application in, consider these restrictions:

- If you use z/OS macros (ATTACH, WAIT, POST, and so on), you must choose a programming language that supports them.
- The RRSF TRANSLATE function is not available from Fortran. To use the function, code it in a routine written in another language, and then call that routine from Fortran.

Tracing facility

A tracing facility provides diagnostic messages that help you debug programs and diagnose errors in the RRSF code. The trace information is available only in a SYSABEND or SYSUDUMP dump.

Program preparation

Preparing your application program to run in RRSF is similar to preparing it 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. For examples and guidance in program preparation, see Chapter 21, "Preparing an application program to run," on page 471.

RRSF requirements

When you write an application to use RRSF, be aware of the following requirements.

Program size

The RRSF code requires about 10-KB of virtual storage per address space and an additional 10-KB for each TCB that uses RRSF.

Use of LOAD

RRSF uses z/OS SVC LOAD to load a module as part of the initialization following your first service request. The module is loaded into fetch-protected storage that has the job-step protection key. If your local environment intercepts and replaces the LOAD SVC, then 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.

Commit and rollback operations

To commit work in RRSF 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. For information about coding the SRRCMIT and SRRBACK functions, see *z/OS MVS Programming: Callable Services for High-Level Languages*.

Follow these guidelines for choosing the DB2 statements or the CPIC functions for commit and rollback operations:

- Use DB2 COMMIT and ROLLBACK statements when you know that the following conditions are true:
 - The only recoverable resource accessed by your application is DB2 data managed by a single DB2 instance.
DB2 COMMIT and ROLLBACK statements will fail if your RRSF 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.

Run environment

Applications that request DB2 services must adhere to several run environment requirements. Those requirements must be met regardless of the attachment facility you use. They are not unique to RRSF.

- The application must be running in TCB mode.
- No EUT FRRs can be active when the application requests DB2 services. If an EUT FRR is active, DB2's functional recovery can fail, and your application can receive unpredictable abends.
- Different attachment facilities cannot be active concurrently within the same address space. For example:
 - An application should not use RRSF in CICS or IMS address spaces.
 - An application running in an address space that has a CAF connection to DB2 cannot connect to DB2 using RRSF.
 - An application running in an address space that has an RRSF connection to DB2 cannot connect to DB2 using CAF.
- One attachment facility cannot start another. This means your RRSF application cannot use DSN, and a DSN RUN subcommand cannot call your RRSF application.

- The language interface module for RRSF, DSNRLI, is shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load RRSF below the 16-MB line, you must link-edit DSNRLI again.

How to use RRSF

To use RRSF, you must first make available the RRSF language interface load module, DSNRLI. For information about loading or link-editing this module, see “Accessing the RRSF language interface” on page 898.

When the language interface is available, your program can make use of the RRSF in two ways:

- Implicitly, by including SQL statements or IFI calls in your program just as you would in any program. The RRSF facility establishes the connections to DB2 using default values for the pertinent parameters as described in “Implicit connections” on page 897.
- Explicitly, by issuing CALL DSNRLI statements with the appropriate options. For the general form of the statements, see “RRSF function descriptions” on page 903.

The first element of each option list is a *function*, which describes the action you want RRSF to take. For a list of available functions and what they do, see “Summary of connection functions.” The effect of any function depends in part on what functions the program has already performed. Before using any function, be sure to read the description of its usage. Also read “Summary of connection functions,” which describes the influence of previously invoked functions.

Summary of connection functions

You can use the following functions with CALL DSNRLI:

IDENTIFY

Establishes the task as a user of the named DB2 subsystem. When the first task within an address space issues a connection request, the address space is initialized as a user of DB2. See “IDENTIFY: Syntax and usage” on page 904.

SWITCH TO

Directs RRSF, SQL or IFI requests to a specified DB2 subsystem. See “SWITCH TO: Syntax and usage” on page 907.

SIGNON

Provides to DB2 a user ID and, optionally, one or more secondary authorization IDs that are associated with the connection. See “SIGNON: Syntax and usage” on page 909.

AUTH SIGNON

Provides to DB2 a user ID, an Accessor Environment Element (ACEE) and, optionally, one or more secondary authorization IDs that are associated with the connection. See “AUTH SIGNON: Syntax and usage” on page 914.

CONTEXT SIGNON

Provides to DB2 a user ID and, optionally, one or more secondary authorization IDs that are associated with the connection. You can execute CONTEXT SIGNON from an unauthorized program. See “CONTEXT SIGNON: Syntax and usage” on page 918.

SET_ID

Sets end-user information that is passed to DB2 when the next SQL request is

processed. SET_ID establishes a new value for the client program ID that can be used to identify the end user. See “SET_ID: Syntax and usage” on page 922.

SET_CLIENT_ID

Sets end-user information that is passed to DB2 when the next SQL request is processed. SET_CLIENT_ID establishes new values for the client user ID, the application program name, the workstation name, and the accounting token. See “SET_CLIENT_ID: Syntax and usage” on page 923.

CREATE THREAD

Allocates a DB2 plan or package. CREATE THREAD must complete before the application can execute SQL statements. See “CREATE THREAD: Syntax and usage” on page 926.

TERMINATE THREAD

Deallocates the plan. See “TERMINATE THREAD: Syntax and usage” on page 928.

TERMINATE IDENTIFY

Removes the task as a user of DB2 and, if this is the last or only task in the address space that has a DB2 connection, terminates the address space connection to DB2. See “TERMINATE IDENTIFY: Syntax and usage” on page 930.

TRANSLATE

Returns an SQL code and printable text, in the SQLCA, that describes a DB2 error reason code. You cannot call the TRANSLATE function from the Fortran language. See “TRANSLATE: Syntax and usage” on page 931.

Implicit connections

If you do not explicitly specify the IDENTIFY function in a CALL DSNRLI statement, RRSAP initiates an implicit connection to DB2 if the application includes SQL statements or IFI calls. An implicit connection causes RRSAP to initiate implicit IDENTIFY and CREATE THREAD requests to DB2. Although RRSAP performs the connection request by using the following default values, the request is subject to the same DB2 return codes and reason codes as are explicitly specified requests.

Implicit connections use the following defaults:

Subsystem name

The default name specified in the module DSNHDECP. RRSAP uses the installation default DSNHDECP, unless your own DSNHDECP is in a library in a STEPLIB of JOBLIB concatenation, or in the link list. In a data sharing group, the default subsystem name is the group attachment name.

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. If your program can make its first SQL call from different modules with different DBRMs, you cannot use a default plan name; you must use an explicit call using the CREATE THREAD function.

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

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 IDENTIFY or CREATE THREAD. It can execute other explicit RRSF calls
| after the implicit connection. An implicit connection does not perform any
| SIGNON processing. Your application can execute SIGNON at any point of
| consistency. To terminate an implicit connection, you must use the proper calls. See
| "Summary of RRSF behavior" on page 902 for details.

| Your application program must successfully connect, either implicitly or explicitly,
| to DB2 before it can execute any SQL calls to the RRSF DSNHLL entry point.
| Therefore, the application program must first determine the success or failure of all
| implicit connection requests.

| 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. The application program should examine the return and
| reason codes immediately after the first executable SQL statement within the
| application program. Two ways to do this are to:

- 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.

Accessing the RRSF language interface

Figure 247 on page 899 shows the general structure of RRSF and a program that uses it.

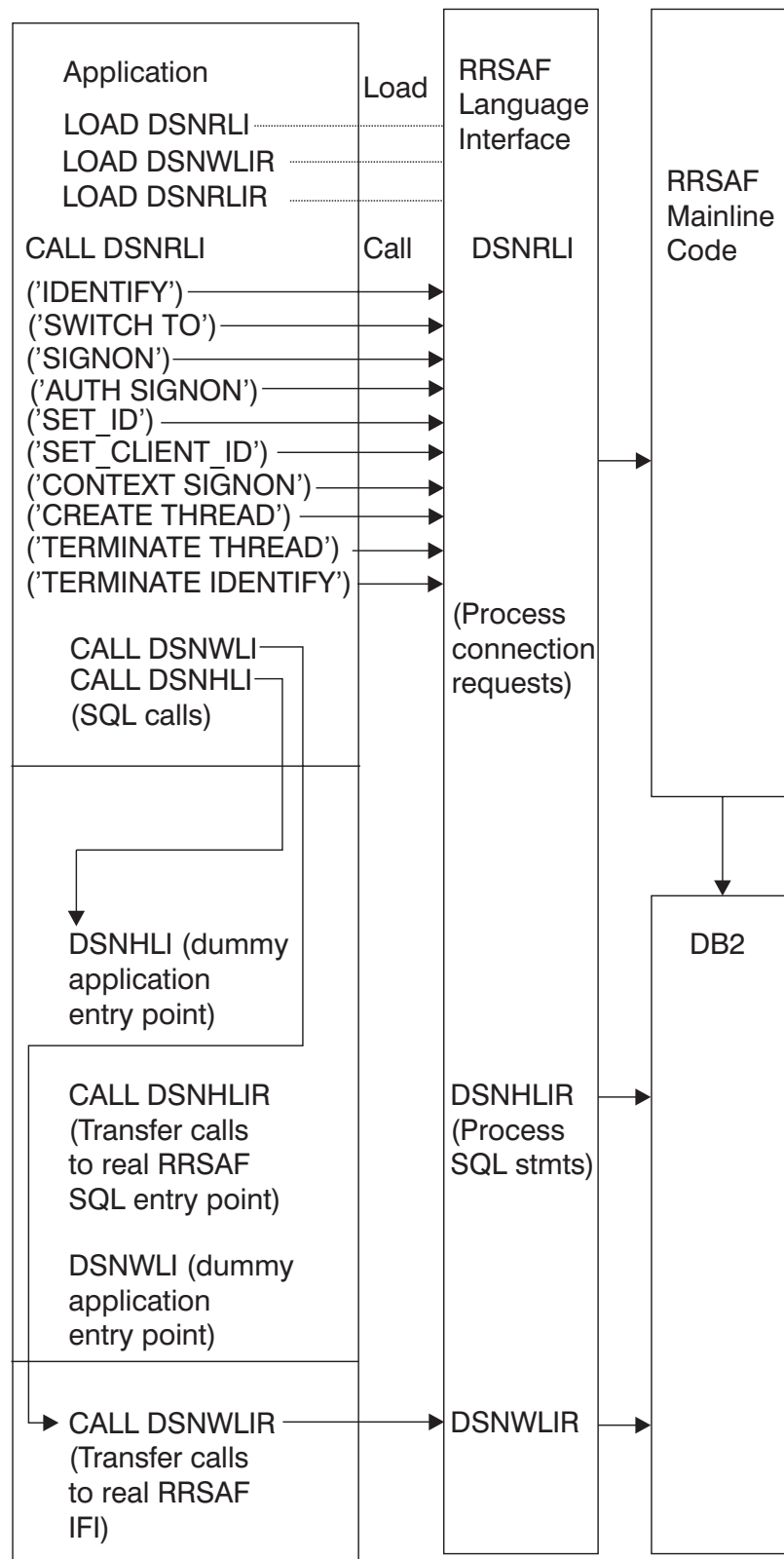


Figure 247. Sample RRSF configuration

Part of RRSF is a DB2 load module, DSNRLI, the RRSF language interface module. DSNRLI has the alias names DSNHLIR and DSNWLIR. The module has five entry points: DSNRLI, DSNHLI, DSNHLIR, DSNWLI, and DSNWLIR:

- Entry point DSNRLI handles explicit DB2 connection service requests.
- DSNHLI and DSNHLIR handle SQL calls. Use DSNHLI if your application program link-edits RRSAP; use DSNHLIR if your application program loads RRSAP.
- DSNWLI and DSNWLIR handle IFI calls. Use DSNWLI if your application program link-edits RRSAP; use DSNWLIR if your application program loads RRSAP.

You can access the DSNRLI module by explicitly issuing LOAD requests when your program runs, or by including the DSNRLI module in your load module when you link-edit your program. There are advantages and disadvantages to each approach.

Explicit Load of DSNRLI

To load DSNRLI, issue z/OS LOAD macros 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.

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.

You must indicate to DB2 which entry point to use. You can do this in one of two ways:

- Specify the precompiler option ATTACH(RRSAP).
This causes DB2 to generate calls that specify entry point DSNHLIR. 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 and is independent of the different DB2 attachment facilities. When the calls generated by the DB2 precompiler pass control to DSNHLI, your code corresponding to the dummy entry point must preserve the option list passed in register 1 and call DSNHLIR specifying the same option list. For a coding example of a dummy DSNHLI entry point, see “Using dummy entry point DSNHLI for RRSAP” on page 935.

Link-editing DSNRLI

You can include DSNRLI when you link-edit your load module. For example, you can use a linkage editor control statement like this in your JCL:

```
INCLUDE DB2LIB(DSNRLI).
```

By coding this statement, you avoid linking the wrong language interface module.

When you include DSNRLI during the link-edit, you 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 DSNWLI, which is identical to DSNWLIR.

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.

General properties of RRSAP connections

Some of the basic properties of an RRSAP connection with DB2 are:

Connection name and connection type: The connection name and connection type are RRSAF. You can use the DISPLAY THREAD command to list RRSAF applications that have the connection name RRSAF.

Authorization id: Each DB2 connection is associated with a set of authorization IDs. A connection must have a primary ID, and can have one or more secondary IDs. Those identifiers are used for:

- 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.

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.

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 assigned to the primary and secondary authorization IDs can differ from the values 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 can also indicate whether the signon request should be accepted.

For information about authorization IDs and the connection and signon exit routines, see Appendix B (Volume 2) of *DB2 Administration Guide*.

Scope: The 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.

Do not mix RRSAF connections with other connection types in a single address space. The first connection to DB2 made from an address space determines the type of connection allowed.

Task termination

If an application that is connected to DB2 through RRSAF terminates normally before the TERMINATE THREAD or TERMINATE IDENTIFY functions deallocate the plan, then 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, then 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.

DB2abend

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.

Summary of RRSF behavior

Table 143 and Table 144 summarize RRSF behavior after various inputs from application programs. Errors are identified by the DB2 reason code that RRSF returns. For a list of reason codes, see the 'XC1' reason codes in Part 3 of *DB2 Codes*. Use these tables to understand the order in which your application must issue RRSF calls, SQL statements, and IFI requests.

In these tables, the first column lists the most recent RRSF or DB2 function executed. The first row lists the next function executed. The contents of the intersection of a row and column indicate the result of calling the function in the first column followed by the function in the first row. For example, if you issue **TERMINATE THREAD**, then you execute SQL or issue an IFI call, RRSF returns reason code X'00C12219'.

Table 143 summarizes RRSF behavior when the next call is the **IDENTIFY**, **SWITCH TO**, **SIGNON**, or **CREATE THREAD** function.

Table 143. 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'	X'00C12204'	X'00C12204'
IDENTIFY	X'00F30049'	Switch to <i>ssnm</i>	Signon ¹	X'00C12217'
SWITCH TO	IDENTIFY	Switch to <i>ssnm</i>	Signon ¹	CREATE THREAD
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	X'00F30049'	Switch to <i>ssnm</i>	Signon ¹	CREATE THREAD
CREATE THREAD	X'00F30049'	Switch to <i>ssnm</i>	Signon ¹	X'00C12202'
TERMINATE THREAD	X'00C12201'	Switch to <i>ssnm</i>	Signon ¹	CREATE THREAD
IFI	X'00F30049'	Switch to <i>ssnm</i>	Signon ¹	X'00C12202'
SQL	X'00F30049'	Switch to <i>ssnm</i>	X'00F30092' ²	X'00C12202'
SRRRCMIT or SRRBACK	X'00F30049'	Switch to <i>ssnm</i>	Signon ¹	X'00C12202'

Notes:

1. Signon means the signon to DB2 through either **SIGNON**, **AUTH SIGNON**, or **CONTEXT SIGNON**.
2. **SIGNON**, **AUTH SIGNON**, or **CONTEXT SIGNON** are not allowed if any SQL operations are requested after **CREATE THREAD** or after the last **SRRRCMIT** or **SRRBACK** request.

Table 144 summarizes RRSF behavior when the next call is the **SQL** or **IFI**, **TERMINATE THREAD**, **TERMINATE IDENTIFY**, or **TRANSLATE** function.

Table 144. 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 ³	X'00C12204'	X'00C12204'	X'00C12204'
IDENTIFY	SQL or IFI call ³	X'00C12203'	TERMINATE IDENTIFY	TRANSLATE
SWITCH TO	SQL or IFI call ³	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE

Table 144. Effect of call order when next call is SQL or IFI, TERMINATE THREAD, TERMINATE IDENTIFY, or TRANSLATE (continued)

Previous function	Next function			
	SQL or IFI	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	SQL or IFI call ³	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
CREATE THREAD	SQL or IFI call ³	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
TERMINATE THREAD	SQL or IFI call ³	X'00C12203'	TERMINATE IDENTIFY	TRANSLATE
IFI	SQL or IFI call ³	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SQL	SQL or IFI call ³	X'00F30093' ¹	X'00F30093' ²	TRANSLATE
SRRCMIT or SRRBACK	SQL or IFI call ³	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE

Notes:

1. TERMINATE THREAD is not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.
2. TERMINATE IDENTIFY is not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.
3. Using implicit connect with SQL or IFI calls causes RRSF to issue an implicit IDENTIFY and CREATE THREAD. If you continue with explicit RRSF statements after an implicit connect, you must follow the standard order of explicit RRSF calls. Implicit connect does not issue a SIGNON. Therefore, you might need to issue an explicit SIGNON to satisfy the standard order requirement. For example, an SQL statement followed by an explicit TERMINATE THREAD requires an explicit SIGNON before issuing the TERMINATE THREAD.

RRSAF function descriptions

To code RRSF 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 RRSF call.

This section contains the following information:

- "Register conventions"
- "Parameter conventions for function calls" on page 904
- "IDENTIFY: Syntax and usage" on page 904
- "SWITCH TO: Syntax and usage" on page 907
- "SIGNON: Syntax and usage" on page 909
- "AUTH SIGNON: Syntax and usage" on page 914
- "CONTEXT SIGNON: Syntax and usage" on page 918
- "SET_ID: Syntax and usage" on page 922
- "SET_CLIENT_ID: Syntax and usage" on page 923
- "CREATE THREAD: Syntax and usage" on page 926
- "TERMINATE THREAD: Syntax and usage" on page 928
- "TERMINATE IDENTIFY: Syntax and usage" on page 930
- "TRANSLATE: Syntax and usage" on page 931

Register conventions

Table 145 on page 904 summarizes the register conventions for RRSF calls.

If you do not specify the return code and reason code parameters in your RRSF calls, RRSF puts a return code in register 15 and a reason code in register 0. 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. RRSF preserves the

contents of registers 2 through 14.

Table 145. 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

Parameter conventions for function calls

For assembler language: Use a standard parameter list for an z/OS CALL. This means that when you issue the 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.

In an assembler language call, code a comma for a parameter in the CALL DSNRLI statement when you want to use the default value for that parameter and specify subsequent parameters. For example, code an IDENTIFY call like this to specify all parameters except the return code parameter:

```
CALL DSNRLI,(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB,,REASCODE)
```

For all languages: When you code CALL DSNRLI statements in any language, specify all parameters that come before the return code parameter. You cannot omit any of those parameters by coding zeros or blanks. There are no defaults for those parameters.

All parameters starting with *Return Code* are optional.

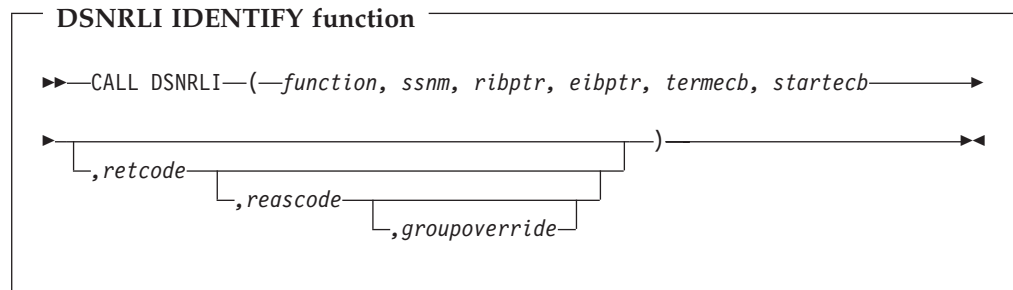
For all languages except assembler language: Code 0 for an optional parameter in the CALL DSNRLI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, suppose you are coding an IDENTIFY call in a COBOL program. You want to specify all parameters except the return code parameter. Write the call in this way:

```
CALL 'DSNRLI' USING IDFYFN SSNM RIBPTR EIBPTR TERMECB STARTECB  
BY CONTENT ZERO BY REFERENCE REASCODE.
```

IDENTIFY: Syntax and usage

IDENTIFY initializes a connection to DB2.

“DSNRLI IDENTIFY function” shows the syntax for the IDENTIFY function.



Parameters point to the following areas:

function

An 18-byte area containing IDENTIFY followed by 10 blanks.

ssnm

A 4-byte DB2 subsystem name or group 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 RRSF places the address of the release information block (RIB) after the call. This can be used 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 fields RIBCNUMB and RIBCINFO. If the value in RIBCNUMB is greater than zero, check RIBCINFO 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.

This parameter is required, although the application does not need to refer to the returned information.

eibptr

A 4-byte area in which RRSF places the address of the environment information block (EIB) after the call. The EIB contains environment information, such as the data sharing group, the member name for the DB2 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, then RRSF 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), RRSF sets the 4-byte area to zeros.

The area to which *eibptr* points is above the 16-MB line.

This parameter is required, although the application does not need to refer to the returned information.

termecb

The address of the application's event control block (ECB) used for DB2 termination. DB2 posts this ECB when the system operator enters the command STOP DB2 or when DB2 is terminating abnormally. Specify a value of 0 if you do not want to use a termination ECB.

RRSAF puts a POST code in the ECB to indicate the type of termination as shown in Table 146.

Table 146. 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 startup has completed. Enter a value of zero if you do not want to use a startup ECB.

#

DB2 posts a maximum of one startup ECB per address space. The ECB posted is associated with the most recent IDENTIFY call from that address space. The application program must examine any nonzero RRSF or DB2 reason codes before issuing a WAIT on this ECB.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify this parameter, RRSF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which RRSF places a reason code.

This parameter is optional. If you do not specify this parameter, RRSF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode* or its default (by specifying a comma or zero, depending on the language).

groupoverride

An 8-byte area that the application provides. This field is optional. If this field is provided, it contains the string '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 name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

Usage

IDENTIFY establishes the caller's task as a user of DB2 services. If no other task in the address space currently is connected to the subsystem named by *ssnm*, then IDENTIFY also initializes the address space to communicate with the DB2 address spaces. IDENTIFY establishes the cross-memory authorization of the address space to DB2 and builds address space control blocks.

During IDENTIFY processing, 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 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 DB2 resource class (CLASS=DSNR)
- The DB2 subsystem (SUBSYS=*ssnm*)
- Connection type RRSF

If that check is successful, DB2 calls the DB2 connection exit to perform additional verification and possibly change the authorization ID. DB2 then sets the connection name to RRSF and the connection type to RRSF.

In a data sharing environment, use the *groupoverride* parameter on an IDENTIFY call 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 name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

Table 147 shows an IDENTIFY call in each language.

Table 147. Examples of RRSF IDENTIFY calls

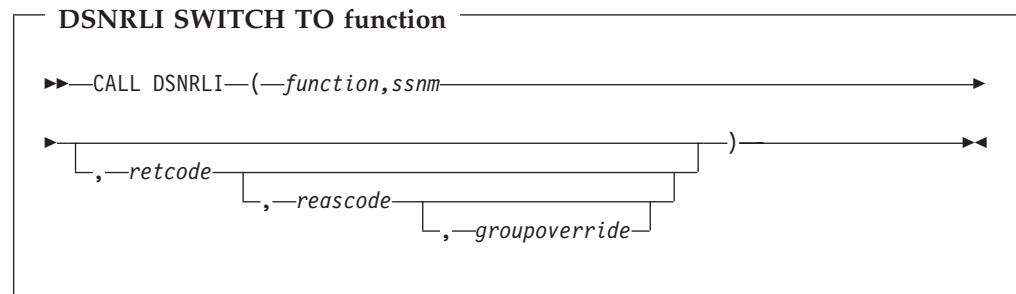
Language	Call example
Assembler	CALL DSNRLI,(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB,RETCODE,REASCODE,GRPOVER)
C	fnret=dsnrli(&idfyfn[0],&ssnm[0], &ribptr, &eibptr, &termecb, &startecb, &retcode, &reascode,&grpover[0]);
COBOL	CALL 'DSNRLI' USING IDFYFN SSNM RIBPTR EIBPTR TERMECB STARTECB RETCODE REASCODE GRPOVER.
Fortran	CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB,RETCODE,REASCODE,GRPOVER)
PL/I	CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB,RETCODE,REASCODE,GRPOVER);
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); } }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

SWITCH TO: Syntax and usage

You can use SWITCH TO to direct RRSF, SQL, or IFI requests to a specified DB2 subsystem.

SWITCH TO is useful only after a successful IDENTIFY call. If you have established a connection with one DB2 subsystem, then you must issue SWITCH TO before you make an IDENTIFY call to another DB2 subsystem.

“DSNRLI SWITCH TO function” shows the syntax for the SWITCH TO function.



Parameters point to the following areas:

function

An 18-byte area containing SWITCH TO followed by nine blanks.

ssnm

A 4-byte DB2 subsystem name or group 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 this parameter, RRSF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify this parameter, 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 field is optional. If this field is provided, it contains the string '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 name. If *groupoverride* is not provided, *ssnm* is used as the group attachment name if it matches a group attachment name. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language, you can omit the return code and reason code parameters by specifying commas as place-holders.

Usage

Use SWITCH TO to establish connections to multiple DB2 subsystems from a single task. If you make a SWITCH TO call to a DB2 subsystem before you have issued an initial IDENTIFY call, DB2 returns return Code 4 and reason code X'00C12205' as a warning that the task has not yet identified to any DB2 subsystem.

After you establish a connection to a DB2 subsystem, you must make a SWITCH TO call before you identify to another DB2 subsystem. If you do not make a SWITCH TO call before you make an IDENTIFY call to another DB2 subsystem, then DB2 returns return Code = X'200' and reason code X'00C12201'.

In a data sharing environment, use the *groupoverride* parameter on an SWITCH TO call 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 name. In general, using the *groupoverride* parameter is not desirable because it limits the ability to do dynamic workload routing in a Parallel Sysplex.

This example shows how you can use SWITCH TO to interact with three DB2 subsystems.

```
RRSAF calls for subsystem db21:
  IDENTIFY
  SIGNON
  CREATE THREAD
Execute SQL on subsystem db21
SWITCH TO db22
RRSAF calls on subsystem db22:
  IDENTIFY
  SIGNON
  CREATE THREAD
Execute SQL on subsystem db22
SWITCH TO db23
RRSAF calls on subsystem db23:
```

```

IDENTIFY
SIGNON
CREATE THREAD
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
SRRCMIT (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
SRRCMIT (to commit the UR)

```

Table 148 shows a SWITCH TO call in each language.

Table 148. Examples of RRSF SWITCH TO calls

Language	Call example
Assembler	CALL DSNRLI,(SWITCHFN,SSNM,RETCODE,REASCODE,GRPOVER)
C	fnret=dsnrli(&switchfn[0], &ssnm[0], &retcode, &reascde,&grpover[0]);
COBOL	CALL 'DSNRLI' USING SWITCHFN RETCODE REASCODE GRPOVER.
Fortran	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER)
PL/I	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER);

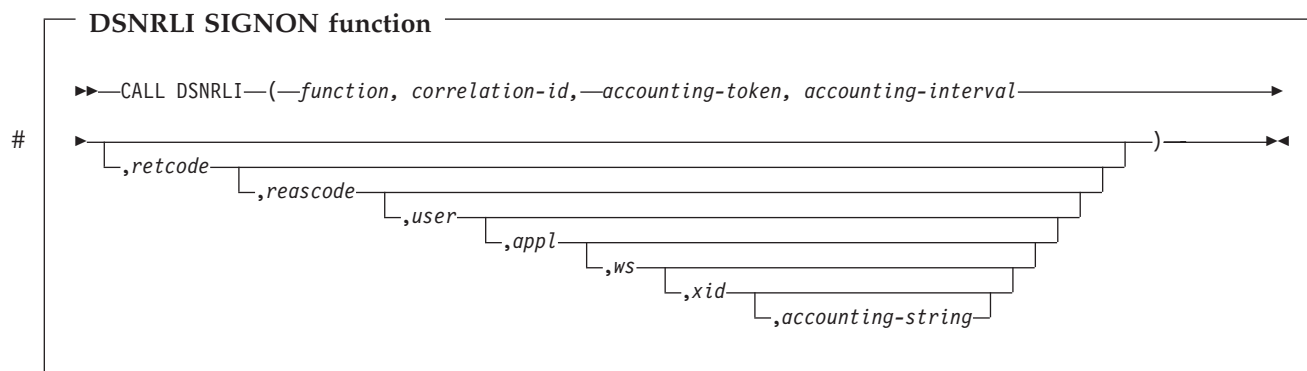
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

SIGNON: Syntax and usage

SIGNON establishes a primary authorization ID and can establish one or more secondary authorization IDs for a connection.

"DSNRLI SIGNON function" on page 910 shows the syntax for the SIGNON function.



Parameters point to the following areas:

function

An 18-byte area containing 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 output from the command DISPLAY THREAD. 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 QWHCTOKEN 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 RRS 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 with which you can control when DB2 writes an accounting record.

If you specify COMMIT in that area, DB2 writes an accounting record each time 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 SRRCMIT 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 SIGNON with a new authorization ID.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify this parameter, 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 this parameter, 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 end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output 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 field is optional. If you specify this parameter, you must also specify *retcode* and *reascode*. 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 end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the application name in the DISPLAY THREAD output 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 field is optional. If you specify this parameter, you must also specify *retcode*, *reascode*, 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 end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the workstation name in the DISPLAY THREAD output 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 this parameter, you must also specify *retcode*, *reascode*, *user*, and *appl*. If you do not specify this parameter, no workstation name is associated with the connection.

xid

A 4-byte area into which you put one of the following values:

0	Indicates that the thread is not part of a global transaction. The 0 value 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.

The global transaction ID has the format shown in Table 149.

However, 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, which is shown in Table 149, to
binary -1 ('FFFFFFFF'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.

Table 149. 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	1 to 64	Character

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.

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 field is optional. If you specify this parameter, 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 specify this field only in DB2 Version 8 new-function mode.

You can also change the value of the accounting string with RRS 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.

Usage

SIGNON causes a new primary authorization ID and an optional secondary authorization IDs to be assigned to a connection. 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 external security interface macro RACROUTE REQUEST=VERIFY to do the following:

- 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.

See *z/OS Security Server RACF Macros and Interfaces* for more information about the RACROUTE macro.

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 there are open held cursors 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.

Table 150 shows a SIGNON call in each language.

Table 150. Examples of RRSF SIGNON calls

Language	Call example
assembler	CALL DSNRLI,(SGNONFN,CORRID,ACCTTKN,ACCTINT,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&sgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &retcode, &reascodes, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING SGNONFN CORRID ACCTTKN ACCTINT RETCODE REASCODE USERID APPLNAME WSNAME.
Fortran	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT,RETCODE,REASCODE,USERID,APPLNAME,WSNAME);
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); } }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

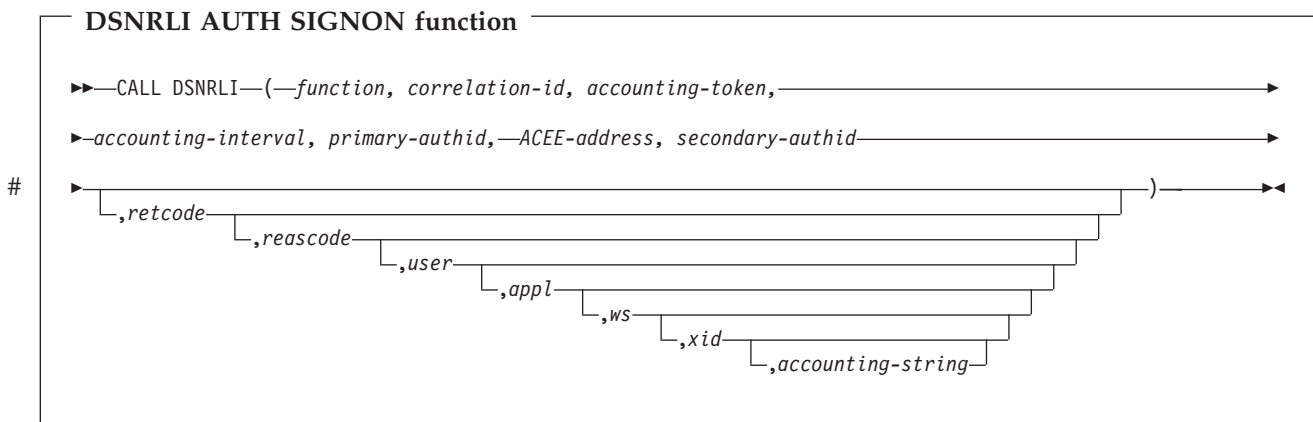
AUTH SIGNON: Syntax and usage

AUTH SIGNON allows an APF-authorized program to pass either of the following to DB2:

- A primary authorization ID and, optionally, one or more secondary authorization IDs.
- An ACEE that is used for authorization checking

AUTH SIGNON establishes a primary authorization ID and can establish one or more secondary authorization IDs for the connection.

“DSNRLI AUTH SIGNON function” shows the syntax for the AUTH SIGNON function.



Parameters point to the following areas:

function

An 18-byte area containing 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 command DISPLAY THREAD. 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
QWHCTOKEN 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 RRS
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 which you can control when DB2 writes an accounting record.

#

If you specify COMMIT in that area, DB2 writes an accounting record each time 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 SRRCMIT 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 SIGNON 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 this parameter, 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 this parameter, RRSF 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 end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output 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 field is optional. If you specify this parameter, you must also specify *retcode* and *reascode*. 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 end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the application name in the DISPLAY THREAD output 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 field is optional. If you specify this parameter, you must also specify *retcode*, *reascodes*, 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 end user. You
# can use this parameter to provide the identity of the client end user for
# accounting and monitoring purposes. DB2 displays the workstation name in
# the DISPLAY THREAD output 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 this parameter, you must also specify
# retcode, reascodes, 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 RRS 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 into which you put one of the following values:

0	Indicates that the thread is not part of a global transaction. The 0 value 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.

The global transaction ID has the format shown in Table 149 on page 912.

```
#
#
#
#
#
#
#
#
#
```

However, 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, which is shown in Table 149 on page 912, to binary -1 ('FFFFFFFF'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.

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.

```
#
# 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 field is optional. If you specify this parameter, 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 specify this field only in DB2 Version 8 new-function mode.
#
# You can also change the value of the accounting string with RRS 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.
```

Usage

AUTH SIGNON causes a new primary authorization ID and optional secondary authorization IDs to be assigned to a 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 there are open held cursors 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.

Table 151 shows a AUTH SIGNON call in each language.

Table 151. Examples of RRS AF AUTH SIGNON calls

Language	Call example
Assembler	CALL DSNRLI,(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEPTR,SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&asgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &pauthid[0], &aceptr, &sauthid[0], &retcode, &reascde, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING ASGNONFN CORRID ACCTTKN ACCTINT PAUTHID ACEPTR SAUTHID RETCODE REASCODE USERID APPLNAME WSNAME.
Fortran	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEPTR,SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEPTR,SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME);

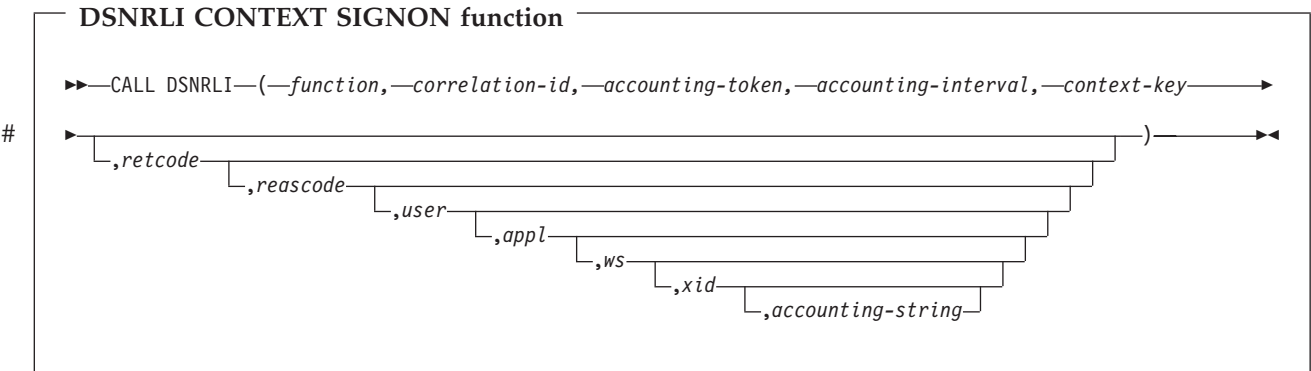
Table 151. Examples of RRSAF AUTH SIGNON calls (continued)

Language	Call example
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); } }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

CONTEXT SIGNON: Syntax and usage

CONTEXT SIGNON establishes a primary authorization ID and one or more secondary authorization IDs for a connection.

"DSNRLI CONTEXT SIGNON function" shows the syntax for the CONTEXT SIGNON function.



Parameters point to the following areas:

- function*
An 18-byte area containing 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 command DISPLAY THREAD. 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 QWHCTOKEN 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 RRS
 # 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 with which you can control when DB2 writes an accounting record.

If you specify COMMIT in that area, DB2 writes an accounting record each
 # time 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 SRRCMIT 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 SIGNON 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 this parameter, 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 this parameter, RRSF 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 end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output 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 field is optional. If you specify this parameter, you must also specify *retcode* and *reascode*. 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 end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the application name in the DISPLAY THREAD output 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 field is optional. If you specify this parameter, 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 end user. You
# can use this parameter to provide the identity of the client end user for
# accounting and monitoring purposes. DB2 displays the workstation name in
# the DISPLAY THREAD output 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 this parameter, 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 RRS SIGNON,
# AUTH SIGNON, or SET_CLIENT_ID. You can retrieve the workstation name
# with the CLIENT_WRKSTNNAME special register.
```

xid

A 4-byte area into which you put one of the following values:

```
0 Indicates that the thread is not part of a global transaction. The
0 value 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.

The global transaction ID has the format shown in Table 149 on page 912.

```
# However, 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, which is shown in Table 149 on
# page 912, to binary -1 ('FFFFFFFF'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.
```

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.

```
# 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 field is optional. If you specify this parameter, 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 specify this field only in DB2 Version 8 new-function mode.

You can also change the value of the accounting string with RRS SIGNON,
AUTH 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.

Usage

CONTEXT SIGNON relies on the RRS context services functions Set Context Data (CTXSDTA) and Retrieve Context Data (CTXRDTA). Before you invoke CONTEXT SIGNON, you must have called 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.

CONTEXT SIGNON establishes a new primary authorization ID for the connection and optionally causes one or more secondary authorization IDs to be assigned. CONTEXT SIGNON uses the context key to retrieve the primary authorization ID from data associated with the current RRS context. DB2 uses the RRS context services function 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.

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 (established at IDENTIFY time or at a previous SIGNON invocation), DB2 invokes only the signon exit. If the value has changed, then DB2 establishes a new primary authorization ID and new SQL authorization ID and then invokes the signon exit.

If you pass an ACEE address, then CONTEXT SIGNON uses the value in ACEEGRP as the secondary authorization ID if the length of the group name (ACEEGRPL) is not 0.

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 there are open held cursors 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.

Table 152 shows a CONTEXT SIGNON call in each language.

Table 152. Examples of RRSF CONTEXT SIGNON calls

Language	Call example
Assembler	CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY,RET CODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&csgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &ctxtkey[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING CSGNONFN CORRID ACCTTKN ACCTINT CTXTKEY RETCODE REASCODE USERID APPLNAME WSNAME.
Fortran	CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RETCODE,REASCODE, USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RET CODE,REASCODE,USERID,APPLNAME,WSNAME);

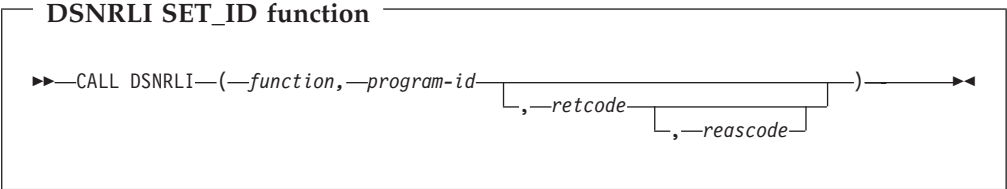
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

```
C      #pragma linkage(dsnrli, OS)
C++    extern "OS" {
        int DSNRLI(
            char * functn,
            ...); }
PL/I    DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RET  
CODE);
```

SET_ID: Syntax and usage

SET_ID sets end-user information that is passed to DB2 when the next SQL request is processed. SET_ID establishes a new value for the client program ID that can be used to identify the end user.

“DSNRLI SET_ID function” shows the syntax of the SET_ID function.



Parameters point to the following areas:

function

An 18-byte area containing SET_ID followed by 12 blanks.

program-id

An 80-byte area containing 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.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify this parameter, RRSF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage

SET_ID establishes a new value for *program-id* that can be used to identify the end user. The calling program defines the contents of *program-id*. 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.

Table 153 shows a SET_ID call in each language.

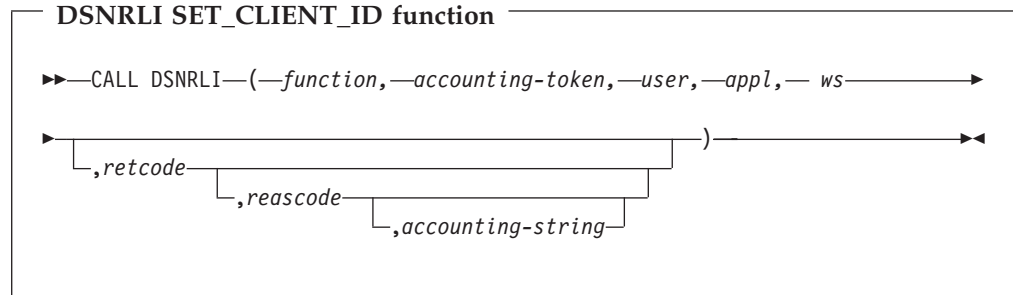
Table 153. Examples of RRSF SET_ID calls

Language	Call example
Assembler	CALL DSNRLI,(SETIDFN,PROGID,RETCODE,REASCODE)
C	fnret=dsnrli(&setidfn[0], &progid[0], &retcode, &reascde);
COBOL	CALL 'DSNRLI' USING SETIDFN PROGID RETCODE REASCODE.
Fortran	CALL DSNRLI(SETIDFN,PROGID,RETCODE,REASCODE)
PL/I	CALL DSNRLI(SETIDFN,PROGID,RETCODE,REASCODE);
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

SET_CLIENT_ID: Syntax and usage

SET_CLIENT_ID sets end-user information that is passed to DB2 when the next SQL request is processed. SET_CLIENT_ID establishes new values for the client user ID, the application program name, the workstation name, the accounting token, and the DDF client accounting string.

"DSNRLI SET_CLIENT_ID function" shows the syntax of the SET_CLIENT_ID function.



Parameters point to the following areas:

function

An 18-byte area containing 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 QWHCTOKEN 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.

You can also change the value of the DB2 accounting token with RRS 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 DISPLAY THREAD output 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 RRS 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 DISPLAY THREAD output 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 RRS 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 DISPLAY THREAD output 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 RRS 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 RRSF places the return code.

This parameter is optional. If you do not specify this parameter, RRSF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSF places the reason code in register 0.

If you specify this parameter, 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 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 field is optional. If you specify this parameter, you must also specify *retcode* and *reascde*. 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 RRS SIGNON, AUTH SIGNON, or CONTEXT SIGNON.

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.

Usage

SET_CLIENT_ID establishes new values that can be used to identify the end user. The calling program defines the contents of these parameters. DB2 places the parameter values in DISPLAY THREAD output and in DB2 accounting and statistics trace records.

Table 154 on page 926 shows a SET_CLIENT_ID call in each language.

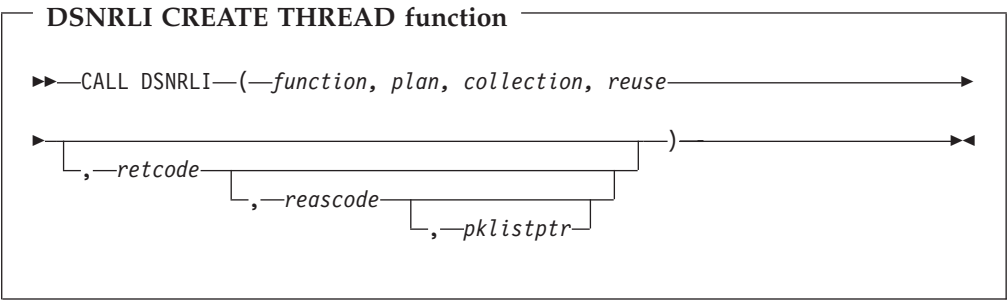
Table 154. Examples of RRSF SET_CLIENT_ID calls

Language	Call example
Assembler	CALL DSNRLI,(SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE)
C	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.
Fortran	CALL DSNRLI(SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE)
PL/I	CALL DSNRLI(SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE);
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

CREATE THREAD: Syntax and usage

CREATE THREAD allocates DB2 resources for the application.

“DSNRLI CREATE THREAD function” shows the syntax of the CREATE THREAD function.



Parameters point to the following areas:

function

An 18-byte area containing CREATE THREAD followed by five blanks.

plan

An 8-byte DB2 plan name. If you provide a collection name instead of a plan name, specify the character ? in the first byte of this field. DB2 then allocates a special plan named ?RRSAF and uses the *collection* parameter. 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. When you provide a collection name and put the character ? in the *plan* field, DB2 allocates a plan named ?RRSAF and a package list that contains two entries:

- This collection name
- An entry that contains * for the location, collection name, and package name

If you provide a plan name in the *plan* field, DB2 ignores the value in this field.

reuse

An 8-byte area that controls the action DB2 takes if a SIGNON call is issued after a CREATE THREAD call. Specify either of these values in this field:

- RESET - to release any held cursors and reinitialize the special registers
- INITIAL - to disallow the SIGNON

This parameter is required. If the 8-byte area does not contain either RESET or INITIAL, then the default value is INITIAL.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify this parameter, RRSF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

pklistptr

A 4-byte field that can contain 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 a pointer to a set of collection IDs (in the *pklistptr* parameter) and the character ? in the plan parameter, DB2 allocates a plan named ?RRSAF and a package list in the data area that *pklistptr* points to. If you also specify a value for the collection parameter, 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 specified in Chapter 1 of *DB2 Command Reference*.

This parameter is optional. If you specify this parameter, you must also specify *retcode* and *reascde*.

If you provide a plan name in the plan field, DB2 ignores the *pklistptr* value.

Using a package list can have a negative impact on performance. For better performance, specify a short package list.

Usage

CREATE THREAD allocates the DB2 resources required to issue SQL or IFI requests. If you specify a plan name, RRSF allocates the named plan.

If you specify ? in the first byte of the plan name and provide a collection name, DB2 allocates a special plan named ?RRSAF and a package list that contains the following entries:

- The collection name
- An entry that contains * for the location, collection ID, and package name

If you specify ? in the first byte of the plan name and specify pklistptr, DB2 allocates a special plan named ?RRSAF 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

The collection names are used to locate a package 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 SQL statement SET CURRENT PACKAGESET to change the collection ID that DB2 uses to locate a package.

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 UDB for z/OS. See Part 3 (Volume 1) of *DB2 Administration Guide* for more information about authorization checking for package execution.

Table 155 shows a CREATE THREAD call in each language.

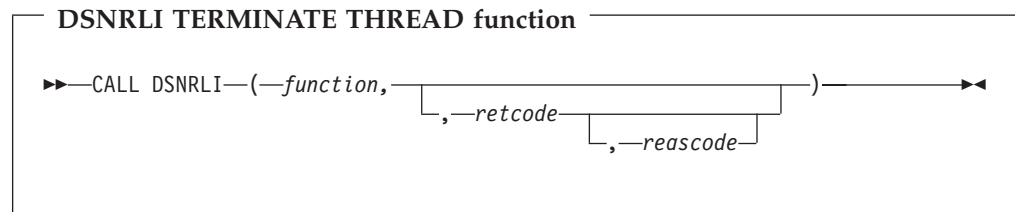
Table 155. Examples of RRSAF CREATE THREAD calls

Language	Call example
Assembler	CALL DSNRLI,(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLISTPTR)
C	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	CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLSTPTR);
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); } }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

TERMINATE THREAD: Syntax and usage

TERMINATE THREAD deallocates DB2 resources that were previously allocated for an application by CREATE THREAD.

"DSNRLI TERMINATE THREAD function" on page 929 shows the syntax of the TERMINATE THREAD function.



Parameters point to the following areas:

function

An 18-byte area containing TERMINATE THREAD followed by two blanks.

retcode

A 4-byte area in which RRSAP places the return code.

This parameter is optional. If you do not specify this parameter, RRSAP places the return code in register 15 and the reason code in register 0.

reascodes

A 4-byte area in which RRSAP places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAP places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage

TERMINATE THREAD deallocates the DB2 resources associated with a plan. Those resources were previously allocated through CREATE THREAD. You can then use CREATE THREAD to allocate another plan using the same connection.

If you issue TERMINATE THREAD, and the application is not at a point of consistency, RRSAP returns reason code X'00C12211'.

Table 156 shows a TERMINATE THREAD call in each language.

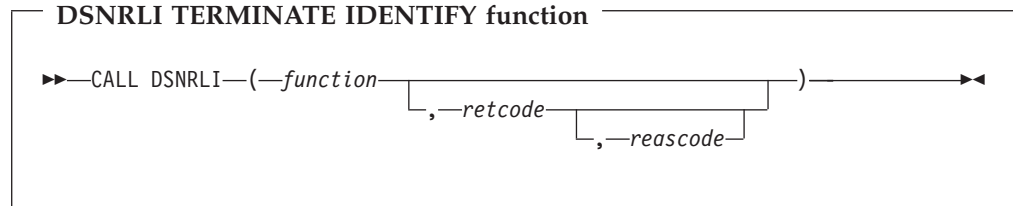
Table 156. Examples of RRSAP TERMINATE THREAD calls

Language	Call example
Assembler	CALL DSNRLI,(TRMTHDFN,RETCODE,REASCODE)
C	fnret=dsnrli(&trmthdfn[0], &retcode, &reascodes);
COBOL	CALL 'DSNRLI' USING TRMTHDFN RETCODE REASCODE.
Fortran	CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE)
PL/I	CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE);
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

TERMINATE IDENTIFY: Syntax and usage

TERMINATE IDENTIFY terminates a connection to DB2.

“DSNRLI TERMINATE IDENTIFY function” shows the syntax of the TERMINATE IDENTIFY function.



Parameters point to the following areas:

function

An 18-byte area containing TERMINATE IDENTIFY.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify this parameter, 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 this parameter, RRSF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage

TERMINATE IDENTIFY 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 created for the address space and removes the cross-memory authorization.

If the application is not at a point of consistency when you issue TERMINATE IDENTIFY, RRSF returns reason code X'00C12211'.

If the application allocated a plan, and you issue TERMINATE IDENTIFY without first issuing TERMINATE THREAD, DB2 deallocates the plan before terminating the connection.

Issuing TERMINATE IDENTIFY is optional. If you do not, 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 ensures that future connection requests from the task are successful when DB2 restarts.

Table 157 on page 931 shows a TERMINATE IDENTIFY call in each language.

Table 157. Examples of RRSF TERMINATE IDENTIFY calls

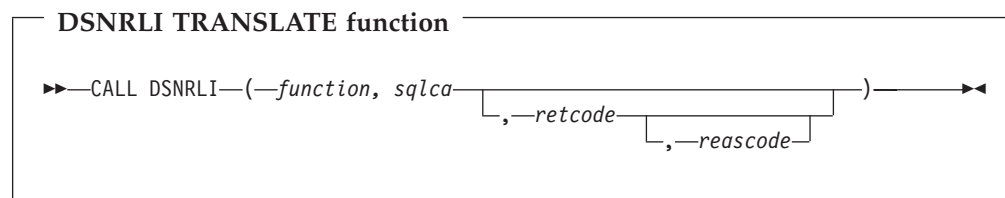
Language	Call example
Assembler	CALL DSNRLI,(TMIDFYFN,RETCODE,REASCODE)
C	fnret=dsnrli(&tmidfyfn[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING TMIDFYFN RETCODE REASCODE.
Fortran	CALL DSNRLI(TMIDFYFN,RETCODE,REASCODE)
PL/I	CALL DSNRLI(TMIDFYFN,RETCODE,REASCODE);
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:	
C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

TRANSLATE: Syntax and usage

TRANSLATE converts a hexadecimal reason code for a DB2 error into a signed integer SQLCODE and a printable error message. The SQLCODE and message text are placed in the caller's SQLCA. You cannot call the TRANSLATE function from the Fortran language.

Issue TRANSLATE only after a successful IDENTIFY operation. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.

"DSNRLI TRANSLATE function" shows the syntax of the TRANSLATE function.



Parameters point to the following areas:

function

An 18-byte area containing 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 this parameter, 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 this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage

Use TRANSLATE to get a corresponding SQL error code and message text for the DB2 error reason codes that RRSAF returns in register 0 following a CREATE THREAD service request. DB2 places this information in the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

The TRANSLATE function translates codes that begin with X'00F3', but it does not translate RRSAF reason codes that begin with X'00C1'. If you receive error reason code X'00F30040' (resource unavailable) after an OPEN request, TRANSLATE returns the name of the unavailable database object in the last 44 characters of field SQLERRM. If the DB2 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.

Table 158 shows a TRANSLATE call in each language.

Table 158. Examples of RRSAF TRANSLATE calls

Language	Call example
Assembler	CALL DSNRLI(XLATFN,SQLCA,RETCODE,REASCODE)
C	fnret=dsnrli(&connfn[0], &sqlca, &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING XLATFN SQLCA RETCODE REASCODE.
PL/I	CALL DSNRLI(XLATFN,SQLCA,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C	#pragma linkage(dsnrli, OS)
C++	extern "OS" { int DSNRLI(char * functn, ...); }
PL/I	DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

RRSAF connection examples

This section shows sample examples for connecting tasks to DB2.

Example of a single task

This example shows a single task running in an address space. z/OS RRS controls commit processing when the task terminates normally.

```
IDENTIFY
SIGNON
CREATE THREAD
SQL or IFI
:
:
TERMINATE IDENTIFY
```

Example of multiple tasks

This example shows multiple tasks in an address space. 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			

Example of calling SIGNON to reuse a DB2 thread

This example shows a DB2 thread that is to be used again by another user at a point of consistency. The application calls SIGNON for user B, using the DB2 plan that is allocated by the CREATE THREAD issued for user A.

```
IDENTIFY
SIGNON user A
CREATE THREAD
SQL
...
SRRCMIT
SIGNON user B
SQL
...
SRRCMIT
```

Example of switching DB2 threads between tasks

This example shows how you can switch the threads for four users (A, B, C, and D) among two tasks (1 and 2). The steps that the applications perform are:

- Task 1 creates context a, performs a context switch to make context a active for task 1, then identifies to a subsystem. A task must always perform an identify operation 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, performs a context switch to make context b active for task 2, identifies to the subsystem, then allocates a thread for user B and also 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, identifies to the subsystem, performs a context switch to make context c active for task 1, then allocates a thread for user C and performs SQL operations for user C.

Task 2 does the same for user D.

When the SQL operations for user C complete, task 1 performs a context switch operation to:

- 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:
 - Disassociate the thread for user D from task 2.
 - Associate the thread for user A with task 2.

Task 1	Task 2
CTXBEGC (create context a)	CTXBEGC (create context b)
CTXSWCH(a,0)	CTXSWCH(b,0)
IDENTIFY	IDENTIFY
SIGNON user A	SIGNON user B
CREATE THREAD (Plan A)	CREATE THREAD (plan B)
SQL	SQL
...	...
CTXSWCH(0,a)	CTXSWCH(0,b)
CTXBEGC (create context c)	CTXBEGC (create context d)
CTXSWCH(c,0)	CTXSWCH(d,0)
IDENTIFY	IDENTIFY
SIGNON user C	SIGNON user D
CREATE THREAD (plan C)	CREATE THREAD (plan D)
SQL	SQL
...	...
CTXSWCH(b,c)	CTXSWCH(0,d)
SQL (plan B)	...
...	CTXSWCH(a,0)
	SQL (plan A)

RRSAF return codes and reason codes

For an implicit connection request, register 15 contains the return code, and register 0 contains the reason code. If you specify return code and reason code parameters in an explicit RRSAF call, RRSAF puts the return code and reason code in those parameters. Otherwise, RRSAF puts the return code in register 15 and the reason code in register 0. See Part 3 of *DB2 Codes* for detailed explanations of the reason codes.

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. See Part 2 of *DB2 Codes* for a list of those return codes and their meanings. RRSAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA). See Part 3 of *DB2 Codes* for a list of those return codes and their meanings.

Table 159. RRSAF return codes

Return code	Explanation
0	Successful completion.
4	Status information. See the reason code for details.
>4	The call failed. See the reason code for details.

Program examples for RRSAF

This section contains sample JCL for running an RRSAF application and assembler code for accessing RRSAF.

Sample JCL for using RRSAF

Use the sample JCL that follows as a model for using RRSAF in a batch environment. The DD statement for DSNRRSAF 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=*
```

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
```

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 precompiler option ATTACH(RRSAF), 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 example that follows, 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)

Connecting to DB2 for RRSF

Figure 248 shows how to issue requests for certain RRSF functions (IDENTIFY, SIGNON, CREATE THREAD, TERMINATE THREAD, and TERMINATE IDENTIFY).

The code in Figure 248 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. The task in Figure 248 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
      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
      WRITE 'The current DB2 release level is' RIBREL
***** 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),(CRTHRDFN,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

```

Figure 248. Using RRSF to connect to DB2

Figure 249 shows declarations for some of the variables that are used in Figure 248.

```
***** 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
CRTHRDFN 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
```

Figure 249. Declarations for variables used in the RRSF connection routine

Chapter 32. CICS-specific programming techniques

This section discusses some special topics of importance to CICS application programmers:

- Controlling the CICS attachment facility from an application
- Improving thread reuse
- Detecting whether the CICS attachment facility is operational

Controlling the CICS attachment facility from an application

You can start and stop the CICS attachment facility from within an application program. To start the attach facility, include this statement in your source code:

```
EXEC CICS LINK PROGRAM('DSN2COM0')
```

To stop the attachment facility, include this statement:

```
EXEC CICS LINK PROGRAM('DSN2COM2')
```

In addition, you can start and stop the CICS attachment facility from within an application program by using the system programming interface SET DB2CONN. For more information, see the *CICS Transaction Server for z/OS System Programming Reference*.

Improving thread reuse

In general, you want transactions to reuse threads whenever possible, because there is a high processor cost associated with thread creation. Part 5 (Volume 2) of *DB2 Administration Guide* contains a discussion of what factors affect CICS thread reuse and how you can write your applications to control these factors.

One of the most important things you can do to maximize thread reuse is to close all cursors that you declared WITH HOLD before each sync point, because DB2 does not automatically close them. A thread for an application that contains an open cursor cannot be reused. It is a good programming practice to close all cursors immediately after you finish using them. For more information about the effects of declaring cursors WITH HOLD in CICS applications, see “Held and non-held cursors” on page 122.

Detecting whether the CICS attachment facility is operational

You can use the INQUIRE EXITPROGRAM command in your applications to test whether the CICS attachment is available. The following example shows how to do this:

```
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 isn't up yet
```

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, it is safe to execute SQL statements. For more information about the INQUIRE EXITPROGRAM command, see *CICS Transaction Server for z/OS System Programming Reference*.

Attention

The *stormdrain* effect is a condition that occurs when a system continues to receive work, even though that system is down.

When both of the following conditions are true, the stormdrain effect can occur:

- The CICS attachment facility is down.
- You are using INQUIRE EXITPROGRAM to avoid AEY9 abends.

For more information on the stormdrain effect and how to avoid it, see Chapter 2 of *DB2 Data Sharing: Planning and Administration*.

If the CICS attachment facility is started and you are using standby mode, you do not need to test whether the CICS attachment facility is up before executing SQL. 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. See *CICS DB2 Guide* for information about the STANDBYMODE and CONNECTERROR parameters, and *DB2 Codes* for an explanation of SQLCODE -923.

Chapter 33. 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)
- Application Messaging Interface (AMI)
- WebSphere MQ classes for Java
- WebSphere MQ classes for Java Message Service (JMS)

Restriction: The AMI has been deprecated.

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 either the AMI or the MQI.

Restriction: All DB2 MQ functions that use AMI are deprecated. You can convert those applications that use the AMI-based functions to use the MQI-based functions.

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.

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 either the MQI or the AMI.

Restriction: The AMI and all DB2 MQ functions that use the AMI have been deprecated.

```

# When you send a message by using the AMI or MQI, you must specify 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.
#
# How services and policies are stored and managed depends on whether you are
# using the AMI or the MQI.
#
# 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.
#
# DB2 MQI services: A service describes a destination to which an application
# sends messages or from which an application receives messages. DB2 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.
#
# DB2 MQI policies: A policy controls how the MQ messages are handled. DB2
# 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.

|
| WebSphere MQ message handling with the AMI
| One way to send and receive WebSphere MQ messages from DB2 applications is to
# use the DB2 MQ functions that use AMI. However, be aware that this interface and
# the associated DB2 MQ functions have been deprecated.

# Restriction: The AMI and the DB2 MQ functions that use the AMI have been
# deprecated. You can convert those applications that use the AMI-based functions to
# use the MQI-based functions.

|
| The AMI-based functions use the services and policies that are defined in AMI
| configuration files, which are in XML format. Typically, these files are created and
| maintained by a system administrator. These files also define any default services
| and policies, including the defaults that are provided by DB2.

|
| The application program does not need know the details of the services and
| policies that are defined in these files. 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.

|
| The AMI uses the service and policy to interpret and construct the MQ headers
| and message descriptors. The AMI does not act on the message data.

|
| AMI services: A service describes a destination to which an application sends
| messages or from which an application receives messages. AMI services are
| defined in AMI configuration files.

# Restriction: The AMI and the DB2 MQ functions that use the AMI have been
# deprecated.

|
| The AMI-based DB2 MQ functions use the services that are defined in AMI
| configuration files, which are in XML format. These files are typically created and
| maintained by a system administrator. These files contain all of the defined
| services, including your customized services and any default services, such as the
| one that DB2 provides.

|
| The application program does not need know the details of the defined services.
| When an application program calls an AMI-based DB2 MQ function, the program
| selects a service from the AMI configuration file by specifying it as a parameter.

|
| AMI policies: A policy controls how the MQ messages are handled. AMI policies
| are defined in AMI configuration files.

# Restriction: The AMI and the DB2 MQ functions that use the AMI have been
# deprecated.

```


The AMI-based DB2 MQ functions use the policies that are defined in AMI configuration files, which are in XML format. These files are typically created and maintained by a system administrator. These files contain all of the defined policies, including your customized policies and any default policies, such as the one that DB2 provides.

The application program does not need know the details of the defined policies. When an application program calls an AMI-based DB2 MQ function, the program selects a policy from the AMI configuration file by specifying it as a parameter.

WebSphere MQ functions and 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 WebSphere MQ server is located on the same z/OS system as the DB2 database server. The DB2 MQ functions and stored procedures are registered with the DB2 database server and provide access to the WebSphere MQ server by using either the AMI or the MQI.

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 or a version that uses the AMI. (Any exceptions are noted in the description of these functions.) The function signatures are the same. However, the qualifying schema names are different. To call an MQI-based function, specify the schema name DB2MQ. To call an AMI-based function, specify the schema names DB2MQ1C, DB2MQ1N, DB2MQ2C, or DB2MQ2N.

Restriction: All DB2 MQ functions that use the AMI have been deprecated. You can convert those applications that use the AMI-based functions to use the MQI-based functions.

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 160. DB2 MQ scalar functions

Scalar function	Description
MQPUBLISH (<i>publisher-service, service-policy, msg-data, topic-list, correlation-id</i>)	MQPUBLISH publishes a message, as specified in the <i>msg-data</i> variable, to the WebSphere MQ publisher that is specified in the <i>publisher-service</i> variable. It uses the quality of service policy as specified in the <i>service-policy</i> variable. The <i>topic-list</i> variable specifies a list of topics for the message. The optional <i>correlation-id</i> variable specifies the correlation id that is to be associated with this message. The return value is 1 if successful or 0 if not successful.
# #	Restriction: MQPublish uses the AMI only. A version of MQPublish that uses the MQI is not available.
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.
MQSUBSCRIBE (<i>subscriber-service, service-policy, topic-list</i>)	MQSUBSCRIBE registers interest in WebSphere MQ messages that are published to the list of topics that are specified in the <i>topic-list</i> variable. The <i>subscriber-service</i> variable specifies a logical destination for messages that match the specified list of topics. Messages that match each topic are placed on the queue at the specified destination, using the policy specified in the <i>service-policy</i> variable. These messages can be read or received by issuing a subsequent call to MQREAD, MQREADALL, MQREADCLOB, MQREADALLCLOB, MQRECEIVE, MQRECEIVEALL, MQRECEIVECLOB, or MQRECEIVEALLCLOB. The return value is 1 if successful or 0 if not successful.
# #	Restriction: MQSUBSCRIBE uses the AMI only. A version of MQSUBSCRIBE that uses the MQI is not available.
MQUNSUBSCRIBE (<i>subscriber-service, service-policy, topic-list</i>)	MQUNSUBSCRIBE unregisters previously specified interest in WebSphere MQ messages that are published to the list of topics that are specified in the <i>topic-list</i> variable. The <i>subscriber-service</i> , <i>service-policy</i> , and <i>topic-list</i> variables specify which subscription is to be cancelled. The return value is 1 if successful or 0 if not successful.
# #	Restriction: MQUNSUBSCRIBE uses the AMI only. A version of MQSUBSCRIBE that uses the MQI is not available.

| Table 160. DB2 MQ scalar functions (continued)

Scalar function	Description
# 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. Restriction: The versions of these MQ functions that are in the DB2MQ1C, DB2MQ1N, DB2MQ2C, and DB2MQ2N schemas are deprecated. (Those functions use the AMI.) Instead use the version of these functions in the DB2MQ schema. (Those functions use the MQI.) The exceptions are MQPUBLISH, MQSUBSCRIBE, and MQUNSUBSCRIBE. Although the AMI-based versions of these functions are deprecated, a version of these functions does not exist in the DB2MQ schema.	

The following table describes the MQ table functions that DB2 can use.

| Table 161. DB2 MQ table functions

Table functions	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.
MQRECEIVEALL (<i>receive-service, service-policy, correlation-id, 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, service-policy, correlation-id, 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. For a description of the other columns, see <i>DB2 SQL Reference</i> . # 3. Restriction: The versions of these MQ functions that are in the DB2MQ1C, DB2MQ1N, DB2MQ2C, and DB2MQ2N schemas are deprecated. (Those functions use the AMI.) Instead use the version of these functions in the DB2MQ schema. (Those functions use the MQI.)	

The following table describes the MQ functions that DB2 can use to work with XML data.

Restriction: All of these DB2 MQ XML-specific functions have been deprecated.

| *Table 162. DB2 MQ XML-specific functions*

XML-specific function	Description
MQREADXML (<i>receive-service, service-policy</i>)	MQREADXML returns the first message in a queue without removing the message from the queue.
MQREADALLXML (<i>receive-service, service-policy</i>)	MQREADALLXML returns a table that contains messages from a queue without removing the messages from the queue.
MQRECEIVEXML (<i>receive-service, service-policy, correlation-id</i>)	MQRECEIVEXML returns a message from the queue and removes that message from the queue.
MQRECEIVEALLXML (<i>receive-service, service-policy, correlation-id</i>)	MQRECEIVEALLXML returns a table that contains messages from a queue and removes the messages from the queue.
MQSENDXML (<i>send-service, service-policy, correlation-id</i>)	MQSENDXML sends a message and does not expect a reply.
MQSENDXMLFILE (<i>send-service, service-policy, correlation-id</i>)	MQSENDXMLFILE sends a message that contains a file and does not expect a reply.
MQSENDXMLFILECLOB (<i>send-service, service-policy, correlation-id</i>)	MQSENDXMLFILECLOB sends a message that contains a file and does not expect a reply.
MQPUBLISHXML (<i>publisher-service, service-policy, correlation-id</i>)	MQPUBLISHXML sends a message to a queue to be picked up by applications that monitor the queue.

You can use the WebSphere MQ XML stored procedures to retrieve an XML document from a message queue, decompose it into untagged data, and store the data in DB2 UDB tables. You can also compose an XML document from DB2 data and send the document to an MQSeries(R) message queue.

The following table shows WebSphere MQ XML stored procedures for decomposition.

Restriction: All of these DB2 MQ XML decomposition stored procedures have been deprecated.

| *Table 163. DB2 MQ XML decomposition stored procedures*

XML decomposition stored procedure	Description
DXXMQINSERT and DXXMQINSERTALL	Decompose incoming XML documents from a message queue and store the data in new or existing database tables. The DXXMQINSERT and DXXMQINSERTALL stored procedures require an enabled XML collection name as input.
DXXMQINSERTCLOB and DXXMQINSERTALLCLOB	Decompose incoming XML documents from a message queue and store the data in new or existing database tables. The DXXMQINSERTCLOB and DXXMQINSERTALLCLOB stored procedures require an enabled XML collection name as input.
DXXMQSHRED and DXXMQSHREDAll	Shred incoming XML documents from a message queue and store the data in new or existing database tables. The DXXMQSHRED and DXXMQSHREDAll stored procedures take a DAD file as input; they do not require an enabled XML collection name as input.
DXXMQSHREDCLOB and DXXMQSHREDALLCLOB	Shred incoming XML documents from a message queue and store the data in new or existing database tables. The DXXMQSHREDCLOB and DXXMQSHREDALLCLOB stored procedures take a DAD file as input; they do not require an enabled XML collection name as input.

|

The following table shows WebSphere MQ XML stored procedures for
composition.

Restriction: All of these DB2 MQ XML composition stored procedures have been
deprecated.

| *Table 164. DB2 MQ XML composition stored procedures*

XML composition stored procedure	Description
DXXMQGEN and DXXMQGENALL	Generate XML documents from existing database tables and send the generated XML documents to a message queue. The DXXMQGEN and DXXMQGENALL stored procedures take a DAD file as input; they do not require an enabled XML collection name as input.
DXXMQRETRIEVE and DXXMQRETRIEVECLOB	Generate XML documents from existing database tables and send the generated XML documents to a message queue. The DXXMQRETRIEVE and DXXMQRETRIEVECLOB stored procedures require an enabled XML collection name as input.

| See Appendix J, “DB2-supplied stored procedures,” on page 1127 for more
| information about WebSphere MQ stored procedures for composing and
| decomposing XML data.

| Commit environment for AMI-based DB2 MQ functions and | stored procedures

DB2 provides two versions of commit when you use AMI-based DB2 MQ functions
and stored procedures: a single-phase commit and a two-phase commit.

Restriction: The AMI and the DB2 MQ functions that use the AMI have been
deprecated.

The schema name when you use AMI-based DB2 MQ functions and stored
procedures for a single-phase commit is DB2MQ1N. The schema name when you
use AMI-based DB2 MQ functions and stored procedures for a two-phase commit
is DB2MQ2N. The schema names DB2MQ1C and DB2MQ2C, are still valid, but
they do not support the parameter style that allows the value to contain binary '0'.

| You need to assign these two versions of the AMI-based DB2 MQ functions and
| stored procedures to different WLM environments, which guarantees that the
| versions are never invoked from the same address space.

For MQI-based DB2 MQ functions, you can specify whether the function is for
one-phase commit or two-phase commit by using the value in the SYNCPOINT
column of the table SYSIBM.MQPOLICY_TABLE.

| Single-phase commit in WebSphere MQ

| If your application uses single-phase commit, any DB2 COMMIT or ROLLBACK
| operations are independent of WebSphere MQ operations. If a transaction is rolled
| back, the messages that have been sent to a queue within the current unit of work
| are not discarded.

| This type of commit is typically used in the case of application error. You might
| want to use WebSphere MQ messaging functions to notify a system programmer

that an application error has occurred. The application issues a ROLLBACK after the error occurs, but the message is still delivered to the queue that contains the error messages.

In a single-phase commit environment, WebSphere MQ controls its own queue operations. A DB2 COMMIT or ROLLBACK does not affect when or if messages are added to or deleted from an MQ queue.

Two-phase commit in WebSphere MQ

If your application uses two-phase commit, RRS coordinates the commit process. If a transaction is rolled back, the messages that have been sent to a queue within the current unit of work are discarded.

This type of commit is typically used when a transaction causes a message to be sent, which causes another transaction to be initiated. For example, assume that a sales transaction causes a WebSphere MQ message to be sent to a queue. The message causes your inventory system to order replacement merchandise. That message should be discarded if the transaction representing the sale is rolled back.

In a two-phase commit environment, if you want to force messages to be added to or deleted from an MQ queue, you need to issue a COMMIT in your application program after you call a DB2 MQ function.

DB2 MQ tables

The DB2 MQ tables contain service and policy definitions that are used by the 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. Sample job DSNTIJMQ 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.

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.

Column name	Description
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.
ENCODING	This column contains the encoding value for the numeric data in the messages that are sent and received by the MQ functions.
	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.
	The default value for this column is 0, which sets the Encoding field in the MQMD to the value MQENC_NATIVE.
DESC_SHORT	This column contains the short description of the service.
DESC_LONG	This column contains the detailed description of the service.

The following table describes the columns for SYSIBM.MQPOLICY_TABLE.

Table 165. SYSIBM.MQPOLICY_TABLE column descriptions

Column name	Description
POLICYNAME	This column contains the policy name, which is an optional input parameter of the MQ functions.
	This column is the primary key for the SYSIBM.MQPOLICY_TABLE table.
SEND_PRIORITY	This column contains the priority of the message.
	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.
	The default value for this column is -1, which sets the Priority field in the MQMD to the value MQQPRI_PRIORITY_AS_Q_DEF.

Table 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_PERSISTENCE	This column indicates whether the message persists despite any system failures or instances of restarting the queue manager. 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. This column can have the following values: 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.
SEND_EXPIRY	This column contains the message expiration time, in tenths of a second. 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. The default value is -1, which sets the Expiry field to the value MQEI_UNLIMITED.
SEND_RETRY_COUNT	This column contains the number of times that the MQ function is to try to send a message if the procedure fails. The default value is 5.
SEND_RETRY_INTERVAL	This column contains the interval, in milliseconds, between each attempt to send a message. The default value is 1000.

Table 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_NEW_CORRELID	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). This column can have one of the following values: 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	This column specifies how the MsgId field in the message descriptor structure (MQMD) is to be set for report and reply messages. This column corresponds to the Report field in the MQMD. MQ functions use the value in this column to set the Report field. This column can have one of the following values: 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.
SEND_RESPONSE_CORRELID	This column specifies how the CorrelID field in the message descriptor structure (MQMD) is to be set for report and reply messages. This column corresponds to the Report field in the MQMD. MQ functions use the value in this column to set the Report field. This column can have one of the following values: 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.

Table 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_EXCEPTION_ACTION	This column specifies what to do with the original message when it cannot be delivered to the destination queue. 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. This column can have one of the following values: 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	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. 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. This column can have one of the following values: 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. F Sets the MQRO_EXCEPTION_WITH_FULL_DATA option in the Report field in the MQMD.

Table 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_REPORT_COA	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. 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. This column can have one of the following values: 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	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. 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. This column can have one of the following values: 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 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_REPORT_EXPIRY	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. 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. This column can have one of the following values: 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	This column specifies whether the receiving application sends a positive action notification (PAN), a negative action notification (NAN), or both. 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. This column can have one of the following values: 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 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_MSG_TYPE	This column contains the type of message. 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. This column can have one of the following values: DTG Sets the MsgType field in the MQMD to MQMT_DATAGRAM. This value is the default. REQ Sets the MsgType field in the MQMD to MQMT_REQUEST. RLY Sets the MsgType field in the MQMD to MQMT_REPLY. RPT Sets the MsgType field in the MQMD to MQMT_REPORT.
REPLY_TO_Q	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. 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. 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.
REPLY_TO_QMGR	This column contains the name of the queue manager to which the reply and report messages are to be sent. 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. 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.

Table 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
RCV_WAIT_INTERVAL	This column contains the time, in milliseconds, that DB2 is to wait for messages to arrive in the queue. 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. The default is 10.
RCV_CONVERT	This column indicates whether to convert the application data in the message to conform to the CodedCharSetId and Encoding values that are defined for the queue manager. 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. This column can have one of the following values: Y Sets the MQGMO_CONVERT option in the Options field in the MQGMO. This value is the default. N Specifies that no data is to be converted.
RCV_ACCEPT_TRUNC_MSG	This column specifies the behavior of the MQ function when oversized messages are retrieved. 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. This column can have one of the following values: Y Sets the MQGMO_ACCEPT_TRUNCATED_MSG option in the Options field in the MQGMO. This value is the default. 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. 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.

Table 165. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
REV_OPEN_SHARED	This column specifies the input queue mode when messages are retrieved.
	This column corresponds to the Options parameter for an MQOPEN call. MQ functions use the value in this column to set the Options parameter.
	This column can have one of the following values:
	S Sets the MQOO_INPUT_SHARED option. This value is the default.
	E Sets the MQ option MQOO_INPUT_EXCLUSIVE option.
	D Sets the MQ option MQOO_INPUT_AS_Q_DEF option.
SYNCPOINT	This column indicates whether the MQ function is to operate within the protocol for a normal unit of work.
	This column can have one of the following values:
	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.
	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.
DESC_SHORT	This column contains the short description of the policy.
DESC_LONG	This column contains the long description of the policy.

Converting applications to use the MQI functions

The AMI-based DB2 MQ functions have been deprecated. You need to make some changes to any applications that reference these AMI-based functions so that they reference the replacement MQI-based functions.

The DB2 MQ functions were originally implemented with the Application Messaging Interface (AMI). These AMI-based functions have since been deprecated. A newer version of these MQ functions is implemented with the Message Queue Interface (MQI). These MQI-based functions have the same signatures as the AMI-based functions, but have different qualifying schema names. Also, the MQI-based functions use DB2 tables instead of AMI configuration files.

To convert an application to use the MQI functions, perform the following actions:

1. Set up the DB2 MQ functions that are based on MQI by performing the following actions:

- a. Run installation job DSNTIJMQ. This job binds the new MQI-based DB2 MQ functions and creates the tables SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE.
 - b. Convert the contents of the AMI configuration files to rows in the tables SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE.
2. If the application contains unqualified references to DB2 MQ functions, set the CURRENT PATH special register to the schema name DB2MQ.
3. If the application contains qualified references to DB2 MQ functions, change the schema names in those references from the old names (DB2MQ1N, DB2MQ2N, DB2MQ1C, and DB2MQ2C) to DB2MQ.
4. Change the size of any host variables to accommodate for the following larger message sizes:
 - DB2 MQ functions for VARCHAR data can have a maximum message size of 32 KB.
 - DB2 MQ functions for CLOB data can have a maximum message size of 2 MB.

How to use WebSphere MQ functions

This section describes some of the common scenarios for using DB2 MQ functions and provides examples of their use:

- “Basic messaging”
- “Sending messages with WebSphere MQ” on page 960
- “Retrieving messages ” on page 961
- “Application-to-application connectivity ” on page 962

For information about using WebSphere MQ stored procedures, see Appendix J, “DB2-supplied stored procedures,” on page 1127.

Basic messaging

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 ” on page 962.

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, retrying 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 use MQSEND, 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 only sends a message, relying on WebSphere MQ to ensure that the message reaches its destination.

If you send more than one column of information, separate the columns with the characters || ' ' ||.

Example: MQSEND (LASTNAME || ' ' || FIRSTNAME)

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. For more information about two-phase commit, see “Commit environment for AMI-based DB2 MQ functions and stored procedures” on page 948.

Example: The following SQL SELECT statement sends a message that consists of the string “Testing msg”:

```
SELECT DB2MQ2N.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 DB2MQ1N.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 DB2MQ2N.MQSEND (LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT)
FROM EMPLOYEE WHERE DEPARTMENT = '51GA';
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.

Retrieving messages

The DB2 MQ functions allow messages to be either read or received. The difference between reading and receiving is that reading returns the message at the head of a queue without removing it from the queue, whereas receiving causes the message to be removed from the queue. 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. For more information about two-phase commit, see “Commit environment for AMI-based DB2 MQ functions and stored procedures” on page 948.

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

Application-to-application connectivity is typically used to solve the problem of putting together a diverse set of application subsystems. To facilitate application integration, WebSphere MQ provides the means to interconnect applications. This section describes two common scenarios:

- *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 DB2MQ1N schema for single-phase commit. For more information about single-phase commit, see “Commit environment for AMI-based DB2 MQ functions and stored procedures” on page 948.

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 DB2MQ1N.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 DB2MQ1N.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 DB2MQ2N.MQSEND ('InterestedParties','Information of general interest')
# FROM SYSIBM.SYSDUMMY1;
```

| When you require more control over the messages that a particular service should
| receive, you can use the MQPUBLISH function, in conjunction with the WebSphere
| MQSeries Integrator facility. This facility provides a publish-and-subscribe system,
| which provides a scalable, secure environment in which many subscribers can
| register to receive messages from multiple publishers. Subscribers are defined by
| queues, which are represented by service names.

| MQPUBLISH allows you to specify a list of topics that are associated with a
| message. Topics allow subscribers to more clearly specify the messages they
| receive. The following sequence illustrates how the publish-and-subscribe
| capabilities are used:

- | 1. An MQSeries administrator configures the publish-and-subscribe capability of
| the WebSphere MQSeries Integrator facility.
- | 2. Interested applications subscribe to subscriber services that are defined in the
| WebSphere MQSeries Integrator configuration. Each subscriber selects relevant
| topics and can also use the content-based subscription techniques that are
| provided by Version 2 of the WebSphere MQSeries Integrator facility.

3. A DB2 application publishes a message to a specified publisher service. The message indicates the topic it concerns.
4. The MQSeries functions provided by DB2 UDB for z/OS handle the mechanics of publishing the message. The message is sent to the WebSphere MQSeries Integrator facility by using the specified service policy.
5. The WebSphere MQSeries Integrator facility accepts the message from the specified service, performs any processing defined by the WebSphere MQSeries Integrator configuration, and determines which subscriptions the message satisfies. It then forwards the message to the subscriber queues that match the subscriber service and topic of the message.
6. Applications that subscribe to the specific service, and register an interest in the specific topic, will receive the message in their receiving service.

Example: To publish the last name, first name, department, and age of employees who are in department 5LGA, using all the defaults and a topic of EMP, you can use the following statement:

```
SELECT DB2MQ2N.MQPUBLISH (LASTNAME || ' ' || FIRSTNAME || ' ' ||
                          DEPARTMENT || ' ' || char(AGE), 'EMP')
FROM DSN8810.EMP
WHERE DEPARTMENT = '5LGA';
```

Example: The following statement publishes messages that contain only the last name of employees who are in department 5LGA to the HR_INFO_PUB publisher service using the SPECIAL_POLICY service policy:

```
SELECT DB2MQ2N.MQPUBLISH ('HR_INFO_PUB', 'SPECIAL_POLICY', LASTNAME,
                          'ALL_EMP:5LGA', 'MANAGER')
FROM DSN8810.EMP
WHERE DEPARTMENT = '5LGA';
```

The messages indicate that the sender has the MANAGER correlation id. The topic string demonstrates that multiple topics, concatenated using a ':' (a colon) can be specified. In this example, the use of two topics allows subscribers of both the ALL_EMP and the 5LGA topics to receive these messages.

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 DB2MQ2N.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 DB2MQ2N.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 (DB2MQ2N.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 (DB2MQ2N.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 DB2MQ2N.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 UDB 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 allows 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 UDB for z/OS with a trigger to publish a message each time a new employee is hired:

```
# CREATE TRIGGER new_employee AFTER INSERT ON DSN8810.EMP
# REFERENCING NEW AS n
# FOR EACH ROW MODE DB2SQL
# SELECT DB2MQ2N.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 DSN8810.EMP table.

1

##

##

#

#

##

#

##

#

- #####

- 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® Universal Database either directly through WebSphere MQ or through some gateway that supports WebSphere MQ.

MQListener in DB2 for OS/390 and z/OS

DB2 UDB for OS/390 and 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 is a daemon process, so it can run without attaching to any terminal.

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, 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 and running MQListener in DB2 UDB for OS/390 and z/OS

```
# 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 (SYSMQL.LISTENERS) in the default
#    database DSNDB04.
# 3. Bind the DBRM's to the plan DB2MQLSN.
```

```
# Note: If MQListener is not installed in the default path, '/usr/lpp/db2', you must
# replace all occurrences of the string '/usr/lpp/db2' in the samples DSNTIJML,
# DSNTJSP and DSNTIJML with the path name where MQListener is installed
# before you run DSNTIJML.
```

```
# The samples DSNTIJML, 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.
```

```
# Logging and Tracing: Two new environment variables are introduced that control
# logging and tracing. The environment variables are defined in .profile file. Refer to
# the README file for more details.
```

MQLSNTRC

```
# When this ENV variable is set to 1, it will write function entry, data, and
# exit points to a unique HFS file. A unique trace file is generated when 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. Alternatively, it can be configured to use the syslogd daemon
 # interface for writing log records. 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)
 # because they are used by the daemon process for writing. Follow the
 # instructions in the README file that is created in the MQListener
 # installation path.

Configuration table : SYSMQL.LISTENERS: Table 166 describes each of the
 # columns of the configuration table:

Table 166. Description of Columns of SYSMQL.LISTENERS

Column name	Description
CONFIGURATIONNAME ¹	The configuration name. The configuration name allows 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 ¹	The name of the Websphere MQ subsystem that contains the queues that are to be used.
INPUTQUEUE ¹	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

Note:

1. Composite primary key.

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, do the
following:

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  
# /* DEADLLETTER_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
# /*

```

Configuring MQListener tasks

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:

```

db2mqln1/db2mqln2 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:
#
# db2mqln1/db2mqln2 show
#           -ssID <subsystem name>
#           -config <configuration name>
#
#
# To display information about all the configurations, issue the following
# command:
#
# db2mqln1/db2mqln2 show
#           -ssID <subsystem name>
#           -config all
#
# • To remove the messaging tasks, issue the following command:
#
# db2mqln1/db2mqln2 remove
#           -ssID <subsystem name>
#           -config <configuration name>
#           -queueManager <queuemanager name>
#           -inputQueue <inputqueue name>
#
# • To run the MQListener task, issue the following command:
#
# db2mqln1/db2mqln2 run
#           -ssID <subsystem name>
#           -config <configuration name>
#           -adminQueue <adminqueue name>
#           -adminQMGr <adminqueuemanager name>
#
# • To shutdown the MQListener daemon, issue the following command:
#
# db2mqln1/db2mqln2 admin
#           -adminQueue <adminqueue name>
#           -adminQMGr <adminqueuemanager name>
#           -adminCommand shutdown
#
# • To restart the MQListener daemon, issue the following command:
#
# db2mqln1/db2mqln2 admin
#           -adminQueue <adminqueue name>
#           -adminQMGr <adminqueuemanager name>
#           -adminCommand restart
#
# • To get help with the command and the valid parameters, issue the following
# command:
#
# db2mqln1/db2mqln2 help
#
# • To get help for a particular parameter, issue the following command, where
# 'command' is a specific parameter:
#
# db2mqln1/db2mqln2 help <command>

```

Restrictions:

- 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.

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 conditions are such that 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 167. 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: • 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: • 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.

Notes:

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

For more information about the report field, see the WebSphere MQ Information Center at <http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp>.

Creating a sample stored procedure to use with MQListener

Create a sample stored procedure, named 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 simple table using SPUFI or DSNTEP2 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 stored 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. Bind the collection TESTLSRN to the plan DB2MQLSN, which is used by MQListener:

```
BIND PLAN(DB2MQLSN)          +
    PKLIST(LSNR.*,TESTLSRN.*) +
    ACTION(REP) DISCONNECT(EXPLICIT);
```

MQListener examples

The following examples show a simple MQListener application. 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 with threshold = 3
 - REPLY_Q : Output queue or Reply queue
 - DEADLETTER_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 reply queue and original message is sent to the
# deadletter queue.
```

Chapter 34. Using 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 allows you to be a consumer of web services in your DB2 applications. web services employs Simple Object Access Protocol (SOAP). 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

The SOAPHTTPV and SOAPHTTTPC user-defined functions

DB2 provides user-defined functions with which you can 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 SOAPHTTTPC for CLOB data. 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 SOAPHTTTPC are user-defined functions that allow 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 SOAPHTTTPC 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 with which you can 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.

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>'))
```

SQLSTATEs for DB2 as a web services consumer

Table 168 shows the possible SQLSTATE values that DB2 returns for error conditions related to using DB2 as a web services consumer.

Table 168. SQLSTATE values that DB2 returns for error conditions related to using DB2 as a web services consumer

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 on 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 hostname.
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 codepage.
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.

Table 168. SQLSTATE values that DB2 returns for error conditions related to using DB2 as a web services consumer (continued)

SQLSTATE	Description
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.
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.

DB2 as a web services provider

DB2 allows you to 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.

Chapter 35. Programming techniques: Questions and answers

This chapter answers some frequently asked questions about database programming techniques. It contains the following topics.

Providing a unique key for 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.

For more information about using DB2-generated values as unique keys, see Chapter 11, “Using DB2-generated values as keys,” on page 269.

Scrolling through previously retrieved data

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.

These options are described in more detail in “Using a scrollable cursor” and “Using a ROWID or identity column” on page 984.

Using a scrollable cursor

Using a scrollable cursor to fetch backward through data involves these basic steps:

1. Declare the cursor with the SCROLL parameter.
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 DSN8810.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 DSN8810.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 DSN8810.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 DSN8810.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 DSN8810.DEPT
        WHERE DEPTROWID=:hv_dept_rowid;
}

```

Scrolling through a table in any 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 DSN8810.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;

```

Updating data as it is retrieved from the database

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.

Updating previously retrieved data

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. Using a scrollable cursor to update backward involves these basic steps:

1. Declare the cursor with the SENSITIVE STATIC SCROLL parameters.
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.

Updating thousands of rows

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.

Retrieving thousands of rows

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. Review the description of locks in “The ISOLATION option” on page 412 while designing your program. Then review the expected use of tables to predict whether you could have locking problems.

Using SELECT *

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. One alternative 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).

Optimizing retrieval for a small set 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*+1st 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.

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”.

To get the effect of adding data to the “end” of a table, 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 naming that column. The newest insert appears last.

Translating requests from end users into SQL statements

Question: A program translates requests from end users into SQL statements before executing them, and users can save a request. How can the corresponding SQL statement be saved?

Answer: You can 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. (For a description of dynamic SQL, see Chapter 24, “Coding dynamic SQL in application programs,” on page 593.)

Changing the table definition

Question: How can I write an SQL application that allows users to create new tables, add columns to them, increase the length of character columns, rearrange the columns, and delete columns?

Answer: Your program can dynamically execute CREATE TABLE and ALTER TABLE statements entered by users to create new tables, add columns to existing tables, or increase the length of VARCHAR columns. Added columns initially contain either the null value or a default value. Both statements, like any data definition statement, are relatively expensive to execute; consider the effects of locks.

You cannot rearrange or delete columns in a table without dropping the entire table. You can, however, create a view on the table, which includes only the columns you want, in the order you want. This has the same effect as redefining the table.

For a description of dynamic SQL execution, see Chapter 24, "Coding dynamic SQL in application programs," on page 593.

Storing data that does not have a tabular format

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 VARCHAR column or a LOB column.

Finding a violated referential or check constraint

Question: When a referential or check constraint has been violated, how do I determine which one it is?

Answer: When you receive an SQL error because of a constraint violation, print out the SQLCA. You can use the DSNTIAR routine described in "Calling DSNTIAR to display SQLCA fields" on page 98 to format the SQLCA for you. Check the SQL error message insertion text (SQLERRM) for the name of the constraint. For information on possible violations, see SQLCODEs -530 through -548 in Part 2 of *DB2 Codes*.

Part 7. Appendixes

Appendix A. DB2 sample tables

Most of the examples in this book refer to the tables described in this appendix. As a group, the tables include information that describes employees, departments, projects, and activities, and make up a sample application that exemplifies most of the features of DB2. The sample storage group, databases, tablespaces, 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. The contents of any table can easily be reviewed by executing an SQL statement, for example SELECT * FROM DSN8810.PROJ. For convenience in interpreting the examples, the department and employee tables are listed here in full.

The following topics provide additional information:

- “Activity table (DSN8810.ACT)”
- “Department table (DSN8810.DEPT)” on page 994
- “Employee table (DSN8810.EMP)” on page 996
- “Employee photo and resume table (DSN8810.EMP_PHOTO_RESUME)” on page 999
- “Project table (DSN8810.PROJ)” on page 1000
- “Project activity table (DSN8810.PROJACT)” on page 1001
- “Employee to project activity table (DSN8810.EMPPROJACT)” on page 1002
- “Unicode sample table (DSN8810.DEMO_UNICODE)” on page 1003
- “Relationships among the sample tables” on page 1004
- “Views on the sample tables” on page 1004
- “Storage of sample application tables” on page 1009

Activity table (DSN8810.ACT)

The activity table describes the activities that can be performed during a project. The table resides in database DSN8D81A and is created with the following statement:

```
CREATE TABLE DSN8810.ACT
  (ACTNO    SMALLINT      NOT NULL,
   ACTKWD   CHAR(6)       NOT NULL,
   ACTDESC  VARCHAR(20)   NOT NULL,
   PRIMARY KEY (ACTNO)
)
IN DSN8D81A.DSN8S81P
CCSID EBCDIC;
```

Content of the activity table:

Table 169 shows the content of the columns.

Table 169. 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 these indexes:

Table 170. Indexes of the activity table

Name	On Column	Type of Index
DSN8810.XACT1	ACTNO	Primary, ascending
DSN8810.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 (DSN8810.DEPT)

The department table describes each department in the enterprise and identifies its manager and the department to which it reports.

The table, shown in Table 173 on page 995, resides in table space DSN8D81A.DSN8S81D and is created with the following statement:

```
CREATE TABLE DSN8810.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 DSN8D81A.DSN8S81D
CCSID EBCDIC;
```

Because the table is self-referencing, and also is part of a cycle of dependencies, its foreign keys must be added later with these statements:

```
ALTER TABLE DSN8810.DEPT
  FOREIGN KEY RDD (ADMRDEPT) REFERENCES DSN8810.DEPT
  ON DELETE CASCADE;
```

```
ALTER TABLE DSN8810.DEPT
  FOREIGN KEY RDE (MGRNO) REFERENCES DSN8810.EMP
  ON DELETE SET NULL;
```

Content of the department table:

Table 171 shows the content of the columns.

Table 171. Columns of the department table

Column	Column Name	Description
1	DEPTNO	Department ID, the primary key

Table 171. Columns of the department table (continued)

Column	Column Name	Description
2	DEPTNAME	A name describing 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

Table 172 shows the indexes of the department table:

Table 172. Indexes of the department table

Name	On Column	Type of Index
DSN8810.XDEPT1	DEPTNO	Primary, ascending
DSN8810.XDEPT2	MGRNO	Ascending
DSN8810.XDEPT3	ADMRDEPT	Ascending

Table 173 shows the content of the department table:

Table 173. DSN8810.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 nulls until sample job DSNTJ6 updates this column with the location name.

Relationship to other tables:

The table is self-referencing: the value of the administering department must be a department ID.

The table is a parent table of:

- The employee table, through a foreign key on column WORKDEPT
- The project table, through a foreign key on column DEPTNO.

It is a dependent of the employee table, through its foreign key on column MGRNO.

Employee table (DSN8810.EMP)

The employee table identifies all employees by an employee number and lists basic personnel information.

The table shown in Table 176 on page 997 and Table 177 on page 998 resides in the partitioned table space DSN8D81A.DSN8S81E. Because it has a foreign key referencing DEPT, that table and the index on its primary key must be created first. Then EMP is created with the following statement:

```
CREATE TABLE DSN8810.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 RED (WORKDEPT) REFERENCES DSN8810.DEPT
     ON DELETE SET NULL              )
EDITPROC DSN8EAE1
IN DSN8D81A.DSN8S81E
CCSID EBCDIC;
```

Content of the employee table:

Table 174 shows the content of the columns. The table has a check constraint, NUMBER, which checks that the phone number is in the numeric range 0000 to 9999.

Table 174. 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
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

Table 174. Columns of the employee table (continued)

Column	Column Name	Description
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

Table 175 shows the indexes of the employee table:

Table 175. Indexes of the employee table

Name	On Column	Type of Index
DSN8810.XEMP1	EMPNO	Primary, partitioned, ascending
DSN8810.XEMP2	WORKDEPT	Ascending

Table 176 and Table 177 on page 998 show the content of the employee table:

Table 176. Left half of DSN8810.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
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

Table 176. Left half of DSN8810.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
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 177. Right half of DSN8810.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
(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

Table 177. Right half of DSN8810.EMP: employee table (continued)

(EMPNO)	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
(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 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.

It is a dependent of the department table, through its foreign key on column WORKDEPT.

Employee photo and resume table (DSN8810.EMP_PHOTO_RESUME)

The employee photo and resume table complements the employee table. 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 DSN8D81A.DSN8S81E. The following statement creates the table:

```
CREATE TABLE DSN8810.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 DSN8D81L.DSN8S81B
  CCSID EBCDIC;
```

DB2 requires an auxiliary table for each LOB column in a table. These statements define the auxiliary tables for the three LOB columns in DSN8810.EMP_PHOTO_RESUME:

```
CREATE AUX TABLE DSN8810.AUX_BMP_PHOTO
  IN DSN8D81L.DSN8S81M
  STORES DSN8810.EMP_PHOTO_RESUME
  COLUMN BMP_PHOTO;
```

```
CREATE AUX TABLE DSN8810.AUX_PSEG_PHOTO
  IN DSN8D81L.DSN8S81L
  STORES DSN8810.EMP_PHOTO_RESUME
  COLUMN PSEG_PHOTO;
```

```
CREATE AUX TABLE DSN8810.AUX_EMP_RESUME
  IN DSN8D81L.DSN8S81N
  STORES DSN8810.EMP_PHOTO_RESUME
  COLUMN RESUME;
```

Content of the employee photo and resume table:

Table 178 shows the content of the columns.

Table 178. Columns of the employee photo and resume table

Column	Column Name	Description
1	EMPNO	Employee ID (the primary key)

Table 178. Columns of the employee photo and resume table (continued)

Column	Column Name	Description
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

Table 179 shows the indexes for the employee photo and resume table:

Table 179. Indexes of the employee photo and resume table

Name	On Column	Type of Index
DSN8810.XEMP_PHOTO_RESUME	EMPNO	Primary, ascending

Table 180 shows the indexes for the auxiliary tables for the employee photo and resume table:

Table 180. Indexes of the auxiliary tables for the employee photo and resume table

Name	On Table	Type of Index
DSN8810.XAUX_BMP_PHOTO	DSN8810.AUX_BMP_PHOTO	Unique
DSN8810.XAUX_PSEG_PHOTO	DSN8810.AUX_PSEG_PHOTO	Unique
DSN8810.XAUX_EMP_RESUME	DSN8810.AUX_EMP_RESUME	Unique

Relationship to other tables:

The table is a parent table of the project table, through a foreign key on column RESPEMP.

Project table (DSN8810.PROJ)

The project table describes each project that the business is currently undertaking. Data contained in each row include the project number, name, person responsible, and schedule dates.

The table resides in database DSN8D81A. Because it has foreign keys referencing DEPT and EMP, those tables and the indexes on their primary keys must be created first. Then PROJ is created with the following statement:

```
CREATE TABLE DSN8810.PROJ
  (PROJNO CHAR(6) PRIMARY KEY NOT NULL,
   PROJNAME VARCHAR(24) NOT NULL WITH DEFAULT
   'PROJECT NAME UNDEFINED',
   DEPTNO CHAR(3) NOT NULL REFERENCES
   DSN8810.DEPT ON DELETE RESTRICT,
   RESPEMP CHAR(6) NOT NULL REFERENCES
   DSN8810.EMP ON DELETE RESTRICT,
   PRSTAFF DECIMAL(5, 2) ,
   PRSTDATE DATE ,
   PRENDATE DATE ,
   MAJPROJ CHAR(6))
IN DSN8D81A.DSN8S81P
CCSID EBCDIC;
```

Because the table is self-referencing, the foreign key for that restraint must be added later with:

```
ALTER TABLE DSN8810.PROJ
    FOREIGN KEY RPP (MAJPROJ) REFERENCES DSN8810.PROJ
    ON DELETE CASCADE;
```

Content of the project table:

Table 181 shows the content of the columns.

Table 181. 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 needed between PRSTDATE and PRENDATE to achieve 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

Table 182 shows the indexes for the project table:

Table 182. Indexes of the project table

Name	On Column	Type of Index
DSN8810.XPROJ1	PROJNO	Primary, ascending
DSN8810.XPROJ2	RESPEMP	Ascending

Relationship to other tables:

The table is self-referencing: a nonnull value of MAJPROJ must be a 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 department table, through its foreign key on DEPTNO
- The employee table, through its foreign key on RESPEMP.

Project activity table (DSN8810.PROJACT)

The project activity table lists the activities performed for each project. The table resides in database DSN8D81A. Because it has foreign keys referencing PROJ and ACT, those tables and the indexes on their primary keys must be created first. Then PROJACT is created with the following statement:

```
CREATE TABLE DSN8810.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 DSN8810.PROJ
```

```

                                ON DELETE RESTRICT,
FOREIGN KEY RPAA (ACTNO) REFERENCES DSN8810.ACT
                                ON DELETE RESTRICT)
IN DSN8D81A.DSN8S81P
CCSID EBCDIC;

```

Content of the project activity table:

Table 183 shows the content of the columns.

Table 183. 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 needed to staff the activity
4	ACSTDATE	Estimated activity start date
5	ACENDATE	Estimated activity completion date

Table 184 shows the index of the project activity table:

Table 184. Index of the project activity table

Name	On Columns	Type of Index
DSN8810.XPROJAC1	PROJNO, ACTNO, ACSTDATE	primary, ascending

Relationship to other tables:

The 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 activity table, through its foreign key on column ACTNO
- The project table, through its foreign key on column PROJNO

Employee to project activity table (DSN8810.EMPPROJACT)

The employee to project activity table identifies the employee who performs an activity for a project, tells the proportion of the employee's time required, and gives a schedule for the activity.

The table resides in database DSN8D81A. Because it has foreign keys referencing 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 DSN8810.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 DSN8810.PROJACT
     ON DELETE RESTRICT,

```

```

FOREIGN KEY REPAE (EMPNO) REFERENCES DSN8810.EMP
ON DELETE RESTRICT)
IN DSN8D81A.DSN8S81P
CCSID EBCDIC;

```

Content of the employee to project activity table:

Table 185 shows the content of the columns.

Table 185. Columns of the employee to project activity table

Column	Column Name	Description
1	EMPNO	Employee ID number
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) to be spent on the activity
5	EMSTDATE	Date the activity starts
6	EMENDATE	Date the activity ends

Table 186 shows the indexes for the employee to project activity table:

Table 186. Indexes of the employee to project activity table

Name	On Columns	Type of Index
DSN8810.XEMPPROJECT1	PROJNO, ACTNO, EMSTDATE, EMPNO	Unique, ascending
DSN8810.XEMPPROJECT2	EMPNO	Ascending

Relationship to other tables:

The table is a dependent of:

- The employee table, through its foreign key on column EMPNO
- The project activity table, through its foreign key on columns PROJNO, ACTNO, and EMSTDATE.

Unicode sample table (DSN8810.DEMO_UNICODE)

The Unicode sample table is used to verify that data conversions to and from EBCDIC and Unicode are working as expected. The table resides in database DSN8D81A, and is defined with the following statement:

```

CREATE TABLE DSN8810.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;

```

Content of the Unicode sample table:

Table 187 shows the content of the columns:

Table 187. 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

Figure 250 shows relationships among the tables. These are established by foreign keys in dependent tables that reference primary keys in parent tables. You can find descriptions of the columns with descriptions of the tables.

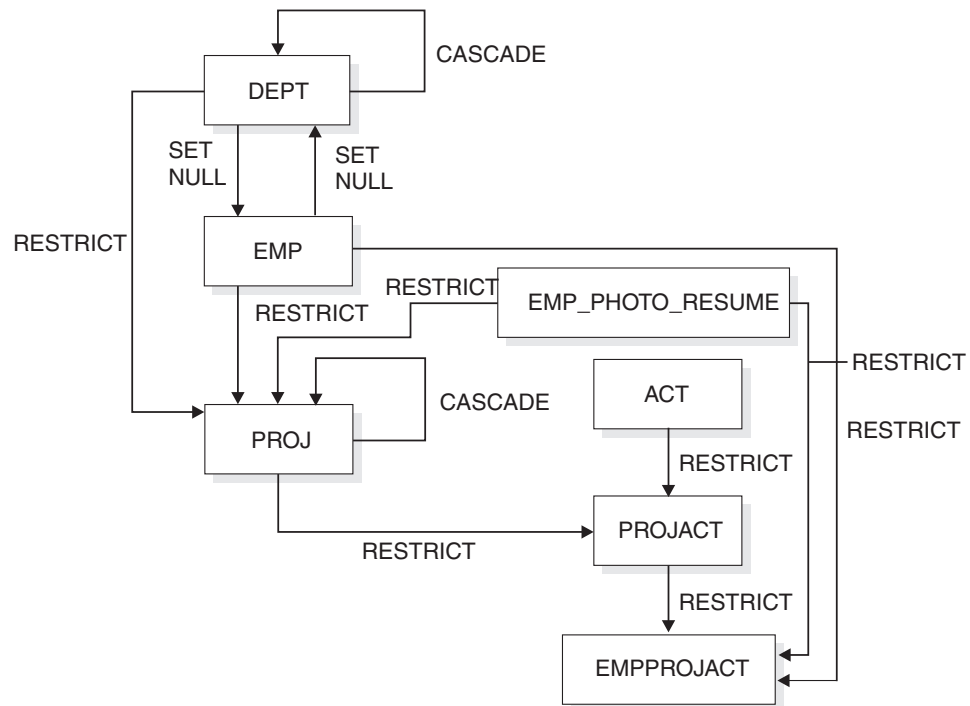


Figure 250. Relationships among tables in the sample application

Views on the sample tables

DB2 creates a number of views on the sample tables for use in the sample applications. Table 188 on page 1005 indicates the tables on which each view is defined and the sample applications that use the view. All view names have the qualifier DSN8810.

Table 188. 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
VSTAFAC1	PROJACT ACT	Project
VSTAFAC2	EMPPROJACT ACT EMP	Project
VPHONE	EMP DEPT	Phone
VEMPLP	EMP	Phone

The following SQL statements are used to create the sample views:

```
CREATE VIEW DSN8810.VDEPT
  AS SELECT ALL    DEPTNO ,
                   DEPTNAME,
                   MGRNO ,
                   ADMRDEPT
  FROM DSN8810.DEPT;
```

Figure 251. VDEPT

```

CREATE VIEW DSN8810.VHDEPT
  AS SELECT ALL      DEPTNO ,
                     DEPTNAME,
                     MGRNO ,
                     ADMRDEPT,
                     LOCATION
  FROM DSN8810.DEPT;

```

Figure 252. VHDEPT

```

CREATE VIEW DSN8810.VEMP
  AS SELECT ALL      EMPNO ,
                     FIRSTNME,
                     MIDINIT ,
                     LASTNAME,
                     WORKDEPT
  FROM DSN8810.EMP;

```

Figure 253. VEMP

```

CREATE VIEW DSN8810.VPROJ
  AS SELECT ALL
      PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF,
      PRSTDATE, PRENDATE, MAJPROJ
  FROM DSN8810.PROJ ;

```

Figure 254. VPROJ

```

CREATE VIEW DSN8810.VACT
  AS SELECT ALL      ACTNO ,
                     ACTKWD ,
                     ACTDESC
  FROM DSN8810.ACT ;

```

Figure 255. VACT

```

CREATE VIEW DSN8810.VPROJACT
  AS SELECT ALL
      PROJNO,ACTNO, ACSTAFF, ACSTDATE, ACENDATE
  FROM DSN8810.PROJACT ;

```

Figure 256. VPROJACT

```

CREATE VIEW DSN8810.VEMPPROJACT
  AS SELECT ALL
      EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, EMENDATE
  FROM DSN8810.EMPPROJACT ;

```

Figure 257. VEMPPROJACT

```

CREATE VIEW DSN8810.VDEPMG1
  (DEPTNO, DEPTNAME, MGRNO, FIRSTNME, MIDINIT,
   LASTNAME, ADMRDEPT)
  AS SELECT ALL
      DEPTNO, DEPTNAME, EMPNO, FIRSTNME, MIDINIT,
      LASTNAME, ADMRDEPT
  FROM DSN8810.DEPT LEFT OUTER JOIN DSN8810.EMP
      ON MGRNO = EMPNO ;

```

Figure 258. VDEPMG1

```

CREATE VIEW DSN8810.VEMPDPT1
  (DEPTNO, DEPTNAME, EMPNO, FRSTINIT, MIDINIT,
   LASTNAME, WORKDEPT)
AS SELECT ALL
  DEPTNO, DEPTNAME, EMPNO, SUBSTR(FIRSTNME, 1, 1), MIDINIT,
  LASTNAME, WORKDEPT
  FROM DSN8810.DEPT RIGHT OUTER JOIN DSN8810.EMP
    ON WORKDEPT = DEPTNO ;

```

Figure 259. VEMPDPT1

```

CREATE VIEW DSN8810.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 DSN8810.VDEPMG1 D1, DSN8810.VDEPMG1 D2
  WHERE D1.DEPTNO = D2.ADMRDEPT ;

```

Figure 260. VASTRDE1

```

CREATE VIEW DSN8810.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 DSN8810.VDEPMG1 D1, DSN8810.EMP E2
  WHERE D1.DEPTNO = E2.WORKDEPT;

```

Figure 261. VASTRDE2

```

CREATE VIEW DSN8810.VPROJRE1
  (PROJNO,PROJNAME,PROJDEP,RESPEMP,FIRSTNME,MIDINIT,
   LASTNAME,MAJPROJ)
AS SELECT ALL
  PROJNO,PROJNAME,DEPTNO,EMPNO,FIRSTNME,MIDINIT,
  LASTNAME,MAJPROJ
  FROM DSN8810.PROJ, DSN8810.EMP
  WHERE RESPEMP = EMPNO ;

```

Figure 262. VPROJRE1

```

CREATE VIEW DSN8810.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 DSN8810.VPROJRE1 P1,
  DSN8810.VPROJRE1 P2
  WHERE P1.PROJNO = P2.MAJPROJ ;

```

Figure 263. VPSTRDE1


```

CREATE VIEW DSN8810.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 DSN8810.VPROJRE1 P1
WHERE NOT EXISTS
    (SELECT * FROM DSN8810.VPROJRE1 P2
     WHERE P1.PROJNO = P2.MAJPROJ) ;

```

Figure 264. VPSTRDE2

```

CREATE VIEW DSN8810.VFORPLA
(PROJNO,PROJNAME,RESPEMP,PROJDEP,FRSTINIT,MIDINIT,LASTNAME)
AS SELECT ALL
    F1.PROJNO,PROJNAME,RESPEMP,PROJDEP, SUBSTR(FIRSTNME, 1, 1),
    MIDINIT, LASTNAME
FROM DSN8810.VPROJRE1 F1 LEFT OUTER JOIN DSN8810.EMPPROJACT F2
ON F1.PROJNO = F2.PROJNO;

```

Figure 265. VFORPLA

```

CREATE VIEW DSN8810.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 DSN8810.PROJACT PA, DSN8810.ACT AC
WHERE PA.ACTNO = AC.ACTNO ;

```

Figure 266. VSTAFAC1

```

CREATE VIEW DSN8810.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 DSN8810.EMPPROJACT EP, DSN8810.ACT AC, DSN8810.EMP EM
WHERE EP.ACTNO = AC.ACTNO AND EP.EMPNO = EM.EMPNO ;

```

Figure 267. VSTAFAC2

```

CREATE VIEW DSN8810.VPHONE
  (LASTNAME,
   FIRSTNAME,
   MIDDLEINITIAL,
   PHONENUMBER,
   EMPLOYEENUMBER,
   DEPTNUMBER,
   DEPTNAME)
AS SELECT ALL      LASTNAME,
                   FIRSTNAME,
                   MIDINIT ,
                   VALUE(PHONENO, '   '),
                   EMPNO,
                   DEPTNO,
                   DEPTNAME
FROM DSN8810.EMP, DSN8810.DEPT
WHERE WORKDEPT = DEPTNO;

```

Figure 268. VPHONE

```

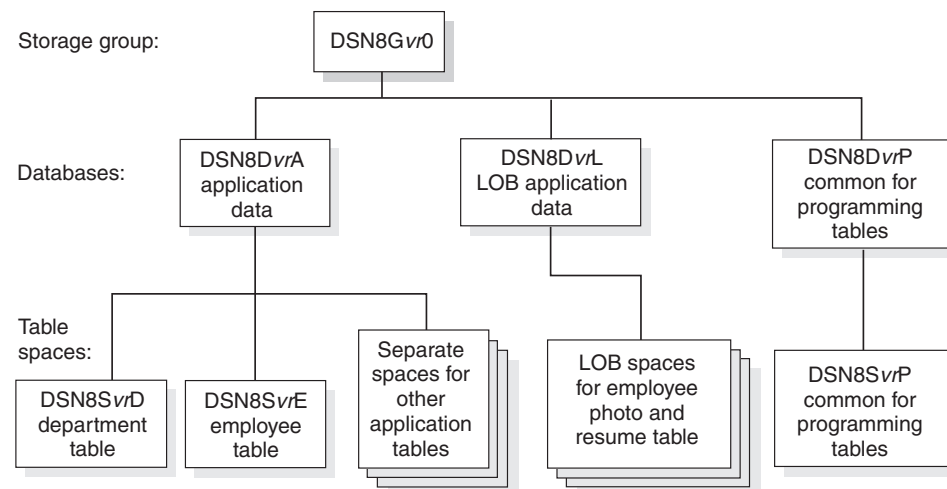
CREATE VIEW DSN8810.VEMPLP
  (EMPLOYEENUMBER,
   PHONENUMBER)
AS SELECT ALL      EMPNO ,
                   PHONENO
FROM DSN8810.EMP ;

```

Figure 269. VEMPLP

Storage of sample application tables

Figure 270 shows how the sample tables are related to databases and storage groups. Two databases are used to illustrate the possibility. Normally, related data is stored in the same database.



vr is a 2-digit version identifier.

Figure 270. Relationship among sample databases and table spaces

In addition to the storage group and databases shown in Figure 270, the storage group **DSN8G81U** and database **DSN8D81U** are created when you run **DSNTEJ2A**.

Storage group

The default storage group, SYSDEFLT, created when DB2 is installed, is not used to store sample application data. The storage group used to store sample application data is defined by this statement:

```
CREATE STOGROUP DSN8G810
  VOLUMES (DSNV01)
  VCAT DSN810;
```

Databases

The default database, created when DB2 is installed, is not used to store the sample application data. DSN8D81P is the database that is used for tables that are related to programs. The remainder of the databases are used for tables that are related to applications. They are defined by the following statements:

```
CREATE DATABASE DSN8D81A
  STOGROUP DSN8G810
  BUFFERPOOL BP0
  CCSID EBCDIC;
```

```
CREATE DATABASE DSN8D81P
  STOGROUP DSN8G810
  BUFFERPOOL BP0
  CCSID EBCDIC;
```

```
CREATE DATABASE DSN8D81L
  STOGROUP DSN8G810
  BUFFERPOOL BP0
  CCSID EBCDIC;
```

```
CREATE DATABASE DSN8D81E
  STOGROUP DSN8G810
  BUFFERPOOL BP0
  CCSID UNICODE;
```

```
CREATE DATABASE DSN8D81U
  STOGROUP DSN8G81U
  CCSID EBCDIC;
```

Table spaces

The following table spaces are explicitly defined by the following statements. The table spaces not explicitly defined are created implicitly in the DSN8D81A database, using the default space attributes.

```
CREATE TABLESPACE DSN8S81D
  IN DSN8D81A
  USING STOGROUP DSN8G810
  PRIQTY 20
  SECQTY 20
  ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;
```

```
CREATE TABLESPACE DSN8S81E
  IN DSN8D81A
  USING STOGROUP DSN8G810
  PRIQTY 20
  SECQTY 20
  ERASE NO
  NUMPARTS 4
  (PART 1 USING STOGROUP DSN8G810
  PRIQTY 12
```

```

        SECQTY 12,
PART 3 USING STOGROUP DSN8G810
        PRIQTY 12
        SECQTY 12)
LOCKSIZE PAGE LOCKMAX SYSTEM
BUFFERPOOL BP0
CLOSE NO
COMPRESS YES
CCSID EBCDIC;

CREATE TABLESPACE DSN8S81B
  IN DSN8D81L
  USING STOGROUP DSN8G810
    PRIQTY 20
    SECQTY 20
    ERASE NO
  LOCKSIZE PAGE
  LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE LOB TABLESPACE DSN8S81M
  IN DSN8D81L
  LOG NO;

CREATE LOB TABLESPACE DSN8S81L
  IN DSN8D81L
  LOG NO;

CREATE LOB TABLESPACE DSN8S81N
  IN DSN8D81L
  LOG NO;

CREATE TABLESPACE DSN8S81C
  IN DSN8D81P
  USING STOGROUP DSN8G810
    PRIQTY 160
    SECQTY 80
  SEGSIZE 4
  LOCKSIZE TABLE
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S81P
  IN DSN8D81A
  USING STOGROUP DSN8G810
    PRIQTY 160
    SECQTY 80
  SEGSIZE 4
  LOCKSIZE ROW
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S81R
  IN DSN8D81A
  USING STOGROUP DSN8G810
    PRIQTY 20
    SECQTY 20
    ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S81S
  IN DSN8D81A
  USING STOGROUP DSN8G810

```

```

        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;

```

Appendix B. Sample applications

This appendix 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.

Several sample applications come with DB2 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 to company.

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*.

Types of sample applications

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. For more information on the utilities stored procedure, see Appendix B of *DB2 Utility Guide and Reference*.
- **SQL procedure applications**
These applications demonstrate how to write, prepare, and invoke SQL 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.
- **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.

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. Part 2 of *DB2 Installation Guide* contains detailed information about using each application. All sample objects have PUBLIC authorization, which makes the samples easier to run.

Application languages and environments: Table 189 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	
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.

Application programs: Tables 190 through 192 on pages 1016 through 1018 provide the program names, JCL member names, and a brief description of some of the programs included for each of the three environments: TSO, IMS, and CICS.

TSO

Table 190. Sample DB2 applications for TSO

Application	Program name	Preparation JCL member name ¹	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.
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 allows you to unload 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.

Table 190. Sample DB2 applications for TSO (continued)

Application	Program name	Preparation JCL member name ¹	Attachment facility	Description
Stored procedures	DSN8EP1	DSNTEJ6P	DSNELI	These applications consist of a calling program, a stored procedure program, or both. Samples that are prepared by jobs DSNTEJ6P, DSNTEJ6S, DSNTEJ6D, and DSNTEJ6T execute DB2 commands using the instrumentation facility interface (IFI). DSNTEJ6P and DSNTEJ6S prepare a PL/I version of the application. DSNTEJ6D and DSNTEJ6T prepare a version in C. The C stored procedure uses result sets to return commands to the client. The sample that is prepared by DSNTEJ61 and DSNTEJ62 demonstrates a stored procedure that accesses IMS databases through the ODBA interface. The sample that is prepared by DSNTEJ6U invokes the utilities stored procedure. The sample that is prepared by jobs DSNTEJ63 and DSNTEJ64 demonstrates how to prepare an SQL procedure using JCL. The sample that is prepared by job DSNTEJ65 demonstrates how to prepare an SQL procedure using the SQL procedure processor. 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. 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.
	DSN8EP2	DSNTEJ6S	DSNALI	
	DSN8EPU	DSNTEJ6U	DSNELI	
	DSN8ED1	DSNTEJ6D	DSNELI	
	DSN8ED2	DSNTEJ6T	DSNALI	
	DSN8EC1	DSNTEJ61	DSNRLI	
	DSN8EC2	DSNTEJ62	DSNELI	
	DSN8ES1	DSNTEJ63	DSNELI	
	DSN8ED3	DSNTEJ64	DSNELI	
	DSN8ES2	DSNTEJ65	DSNELI	
	DSN8ED6	DSNTEJ6W	DSNELI	
	DSN8ED7	DSNTEJ6Z	DSNELI	
User-defined functions	DSN8DUAD	DSNTEJ2U	DSNELI	These applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2 and one user-defined table function, DSN8DUWF, that can be invoked by client program DSN8DUWC. DSN8EUDN and DSN8EUMN are written in C++. All other programs are written in C.
	DSN8DUAT	DSNTEJ2U	DSNELI	
	DSN8DUCD	DSNTEJ2U	DSNELI	
	DSN8DUCT	DSNTEJ2U	DSNELI	
	DSN8DUCY	DSNTEJ2U	DSNELI	
	DSN8DUTI	DSNTEJ2U	DSNELI	
	DSN8DUWC	DSNTEJ2U	DSNELI	
	DSN8DUWF	DSNTEJ2U	DSNELI	
	DSN8EUDN	DSNTEJ2U	DSNELI	
	DSN8EUMN	DSNTEJ2U	DSNELI	
LOBs	DSN8DLPL	DSNTEJ71	DSNELI	These applications demonstrate how to populate a LOB column that is greater than 32KB, manipulate the data using the POSSTR and SUBSTR built-in functions, and display the data in ISPF using GDDM®.
	DSN8DLCT	DSNTEJ71	DSNELI	
	DSN8DLRV	DSNTEJ73	DSNELI	
	DSN8DLPV	DSNTEJ75	DSNELI	

Notes:

1. For information about the DD statements in the sample JCL, see “Using the DB2 precompiler” on page 474.

IMS

Table 191. Sample DB2 applications for IMS

Application	Program name	JCL member name ¹	Description
Organization	DSN8IC0	DSNTEJ4C	IMS COBOL Organization Application
	DSN8IC1		
	DSN8IC2		
Organization	DSN8IP0	DSNTEJ4P	IMS PL/I Organization Application
	DSN8IP1		
	DSN8IP2		
Project	DSN8IP6	DSNTEJ4P	IMS PL/I Project Application
	DSN8IP7		
	DSN8IP8		
Phone	DSN8IP3	DSNTEJ4P	IMS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

Notes:

1. For information about the DD statements in the sample JCL, see “Using the DB2 precompiler” on page 474.

CICS

Table 192. Sample DB2 applications for CICS

Application	Program name	JCL member name ¹	Description
Organization	DSN8CC0	DSNTEJ5C	CICS COBOL Organization Application
	DSN8CC1		
	DSN8CC2		
Organization	DSN8CP0	DSNTEJ5P	CICS PL/I Organization Application
	DSN8CP1		
	DSN8CP2		
Project	DSN8CP6	DSNTEJ5P	CICS PL/I Project Application
	DSN8CP7		
	DSN8CP8		
Phone	DSN8CP3	DSNTEJ5P	CICS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

Notes:

1. For information about the DD statements in the sample JCL, see “Using the DB2 precompiler” on page 474.

Appendix C. Running the 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. You can unload up to 32 KB of data from a LOB column. 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. See “Running DSNTIAUL” on page 1020.

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. See “Running DSNTIAD” on page 1024.

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. See “Running DSNTEP2 and DSNTEP4” on page 1026.

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. See “Running DSNTEP2 and DSNTEP4” on page 1026.

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. Table 193 on page 1020 indicates which installation job prepares each sample program. All installation jobs are in data set DSN810.SDSNSAMP.

Table 193. Jobs that prepare DSNTIAUL, DSNTIAD, DSNTPE2, and DSNTPE4

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, which is described in detail in Chapter 2 of *DB2 Command Reference*. Table 194 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 sections for the meaning of each parameter.

Table 194. DSN RUN option values for DSNTIAUL, DSNTIAD, DSNTEP2, and DSNTEP4

Program name	Load module	Plan	Parameters
DSNTIAUL	DSNTIAUL	DSNTIB81	SQL number of rows per fetch TOLWARN(NO YES)
DSNTIAD	DSNTIAD	DSNTIA81	RC0 SQLTERM(<i>termchar</i>)
DSNTEP2	DSNTEP2	DSNTEP81	ALIGN(MID) or ALIGN(LHS) NOMIXED or MIXED SQLTERM(<i>termchar</i>) TOLWARN(NO YES)
DSNTEP4	DSNTEP4	DSNTEP481	ALIGN(MID) or ALIGN(LHS) NOMIXED or MIXED SQLTERM(<i>termchar</i>) TOLWARN(NO YES)

The remainder of this chapter 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

See the sample jobs that are listed in Table 193 for a working example of each program.

Running DSNTIAUL

This section contains information that you need when you run DSNTIAUL, including parameters, data sets, return codes, and invocation examples.

To retrieve data from a remote site by using the multi-row fetch capability for enhanced performance, bind DSNTIAUL with the DBPROTOCOL(DRDA) option.

```
# To run DSNTIAUL remotely when it is bound with the DBPROTOCOL(PRIVATE)
# option, switch DSNTIAUL to single-row fetch mode by specifying 1 for the
# number of rows per fetch parameter.
```

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. 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.
```

```
# Specify 1 to retrieve data from a remote site when DSNTIAUL is bound with
# the DBPROTOCOL(PRIVATE) option.
```

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.
```

Exception:

```
# YES If a warning occurs when DSNTIAUL executes an OPEN or FETCH to
# retrieve rows, DSNTIAUL continues to retrieve rows.
```

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 SYSREC nn data sets. The maximum record length for the SYSPUNCH and SYSREC nn data sets is 32760 bytes.

DSNTIAUL return codes:

Table 195. DSNTIAUL return codes

Return code	Meaning
0	Successful completion.
4	An SQL statement received a warning code. If the SQL statement was a SELECT statement, DB2 did not perform the associated unload operation. If DB2 return a +394, which indicates that it is using optimization hints, DB2 performs the unload operation.
8	An SQL statement received an error code. If the SQL statement was a SELECT statement, DB2 did not perform the associated unload operation.
12	DSNTIAUL could not open a data set, an SQL statement returned a severe error code (-8 nn or -9 nn), or an error occurred in the SQL message formatting routine.

Examples of DSNTIAUL invocation: 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 Figure 271 on page 1023:


```
//UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB81) -
LIB('DSN810.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=DSN8UNLD.SYSREC00,
// 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 *
DSN8810.PROJ WHERE DEPTNO='D01'
```

Figure 271. DSNTIAUL invocation without the SQL parameter

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 nn 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 Figure 272:

```
//UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB81) -
LIB('DSN810.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 *
DSN8810.DEPT
```

Figure 272. DSNTIAUL invocation to obtain LOAD control statements

Now suppose that you also want to use DSNTIAUL to do these things:

- 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 Figure 273 on page 1024:

#

```
//UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
  DSN SYSTEM(DSN)
  RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB81) PARM('SQL,250') -
    LIB('DSN810.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 DSN8810.EMP IN SHARE MODE;
LOCK TABLE DSN8810.PROJ IN SHARE MODE;
SELECT * FROM DSN8810.PROJ;
SELECT * FROM DSN8810.EMP
  WHERE WORKDEPT LIKE 'D%'
  ORDER BY EMPNO;
```

Figure 273. DSNTIAUL invocation with the SQL parameter

Running DSNTIAD

This section contains information that you need when you run DSNTIAD, including parameters, data sets, return codes, and invocation examples.

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 Table 196. SQLTERM(;) is the default.

Table 196. 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 197. 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 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 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 Figure 274:

```
//RUNTIAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA81) PARM('RC0') -
LIB('DSN810.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
UPDATE DSN8810.PROJ SET DEPTNO='J01' WHERE DEPTNO='A01';
UPDATE DSN8810.PROJ SET DEPTNO='J02' WHERE DEPTNO='A02';
:
UPDATE DSN8810.PROJ SET DEPTNO='J20' WHERE DEPTNO='A20';
```

Figure 274. DSNTIAD Invocation with the RC0 Parameter

Running DSNTPE2 and DSNTPE4

This section contains information that you need when you run DSNTPE2 or DSNTPE4, including parameters, data sets, return codes, and invocation examples.

DSNTPE2 and DSNTPE4 parameters:

ALIGN(MID) or ALIGN(LHS)

Specifies the alignment.

ALIGN(MID)

Specifies that DSNTPE2 or DSNTPE4 output should be centered.

ALIGN(MID) is the default.

ALIGN(LHS)

Specifies that the DSNTPE2 or DSNTPE4 output should be left-justified.

NOMIXED or MIXED

Specifies whether DSNTPE2 or DSNTPE4 contains any DBCS characters.

NOMIXED

Specifies that the DSNTPE2 or DSNTPE4 input contains no DBCS characters. NOMIXED is the default.

MIXED

Specifies that the DSNTPE2 or DSNTPE4 input contains some DBCS characters.

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 196 on page 1024.

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 EMP PROJACT;
SELECT * FROM PROJ;
SELECT * FROM PROJACT;
--#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
# Specify NO (the default) or YES to indicate whether DSNTEP2 or DSNTEP4
# continues to process SQL SELECT statements after receiving an SQL warning:
#
# NO    If a warning occurs when DSNTEP2 or DSNTEP4 executes an OPEN or
#        FETCH for a SELECT statement, DSNTEP2 or DSNTEP4 stops
#        processing the SELECT statement. If SQLCODE +445 or SQLCODE
#        +595 occurs when DSNTEP2 or DSNTEP4 executes a FETCH for a
#        SELECT statement, DSNTEP2 or DSNTEP4 continues to process the
#        SELECT statement. If SQLCODE +802 occurs when DSNTEP2 or
#        DSNTEP4 executes a FETCH for a SELECT statement, DSNTEP2 or
#        DSNTEP4 continues to process the SELECT statement if the
#        TOLARTHWRN control statement is set to YES.
#
# YES   If a warning occurs when DSNTEP2 or DSNTEP4 executes an OPEN or
#        FETCH for a SELECT statement, DSNTEP2 or DSNTEP4 continues to
#        process the SELECT statement.
#

```

DSNTEP2 and DSNTEP4 data sets:

Data Set	Description
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 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*

The control options are:

TERMINATOR

The SQL statement terminator. *value* is any single-byte character other than one of those that are listed in Table 196 on page 1024. 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 and DSNTEP4 continue to process
an SQL SELECT after an SQL warning is returned. *value* is
either NO (the default) or YES.

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.

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 198. 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 Figure 275:

```
//RUNTEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
  DSN SYSTEM(DSN)
  RUN  PROGRAM(DSNTEP2) PLAN(DSNTEP81) PARM('/ALIGN(LHS) MIXED') -
      LIB('DSN810.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
SELECT * FROM DSN8810.PROJ;
```

Figure 275. DSNTEP2 invocation with the ALIGN(LHS) and MIXED parameters

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 Figure 276:

```
//RUNTEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
  DSN SYSTEM(DSN)
  RUN  PROGRAM(DSNTEP4) PLAN(DSNTEP481) PARM('/ALIGN(MID) MIXED') -
      LIB('DSN810.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
--#SET MULT_FETCH 250
SELECT * FROM DSN8810.EMP;
```

Figure 276. DSNTEP4 invocation with the ALIGN(MID) and MIXED parameters and using the MULT_FETCH control option

Appendix D. Programming examples

This appendix contains the following programming examples:

- Sample COBOL dynamic SQL program
- “Sample dynamic and static SQL in a C program” on page 1043
- “Sample DB2 REXX application” on page 1047
- “Sample COBOL program using DRDA access” on page 1061
- “Sample COBOL program using DB2 private protocol access” on page 1069
- “Examples of using stored procedures” on page 1075

To prepare and run these applications, use the JCL in DSN810.SDSNSAMP as a model for your JCL. See Appendix B, “Sample applications,” on page 1013 for a list of JCL procedures for preparing sample programs. See Part 2 of *DB2 Installation Guide* for information on the appropriate compiler options to use for each language.

Sample COBOL dynamic SQL program

Chapter 24, “Coding dynamic SQL in application programs,” on page 593 describes three variations of dynamic SQL statements:

- Non-SELECT statements
- Fixed-List SELECT statements
- Varying-List SELECT statements.

In this case, you know the number of columns returned and their data types when you write the program.

In this case, you do **not** know the number of columns returned and their data types when you write the program.

This appendix documents a technique of coding varying list SELECT statements in
COBOL. See *DB2 Program Directory* for a list of the supported versions of COBOL.

This example program does not support BLOB, CLOB, or DBCLOB data types.

Pointers and based variables

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. Figure 278 on page 1034 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

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

Figure 277 shows an example of the initial program DSN8BCU1 that allocates the storage and calls the second program DSN8BCU2 shown in Figure 278 on page 1034. DSN8BCU2 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.

```

**** DSN8BCU1- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM ****
*
*   MODULE NAME = DSN8BCU1
*
*   DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION
*                       UNLOAD PROGRAM
*                       BATCH
*                       ENTERPRISE COBOL FOR Z/OS OR
*                       IBM COBOL FOR MVS & VM
*
*   FUNCTION = THIS MODULE PROVIDES THE STORAGE NEEDED BY
*               DSN8BCU2 AND CALLS THAT PROGRAM.
*
*   NOTES =
*       DEPENDENCIES = NONE.
*
*   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 = COBOL PROGRAM
*       PROCESSOR   = ENTERPRISE COBOL FOR Z/OS OR
*                   IBM COBOL FOR MVS & VM
*       MODULE SIZE = SEE LINK EDIT
*       ATTRIBUTES  = REENTRANT
*

```

Figure 277. Initial program that allocates storage (Part 1 of 2)

```

*   ENTRY POINT = DSN8BCU1                               *
*   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 =                                  *
*       DSN8BCU2 - ACTUAL UNLOAD PROGRAM                  *
*                                                         *
*   DATA-AREAS      = NONE                                *
*   CONTROL-BLOCKS   = NONE                                *
*                                                         *
*   TABLES = NONE                                         *
*   CHANGE-ACTIVITY = NONE                                *
*                                                         *
* *PSEUDOCODE*                                           *
*                                                         *
*   PROCEDURE                                              *
*   CALL DSN8BCU2.                                         *
*   END.                                                    *
*-----*
/
IDENTIFICATION DIVISION.
*-----*
PROGRAM-ID.    DSN8BCU1
*
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 'DSN8BCU2' USING WORKAREA-IND RECWORK.
    GOBACK.

```

Figure 277. Initial program that allocates storage (Part 2 of 2)

```

**** DSN8BCU2- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM ****
*
*  MODULE NAME = DSN8BCU2
*
*  DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
*                     UNLOAD PROGRAM
*                     BATCH
*                     ENTERPRISE COBOL FOR Z/OS OR
*                     IBM COBOL FOR MVS & VM
*
*  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 SYSREC01.
*             WRITE RESULTS TO SYSPRINT.
*
*  NOTES =
*         DEPENDENCIES = NONE.
*
*  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 = COBOL PROGRAM
*  PROCESSOR   = DB2 PRECOMPILER
*              ENTERPRISE COBOL FOR Z/OS OR
*              IBM COBOL FOR MVS & VM
*  MODULE SIZE = SEE LINK EDIT
*  ATTRIBUTES  = REENTRANT
*
*  ENTRY POINT = DSN8BCU2
*  PURPOSE     = SEE FUNCTION
*  LINKAGE     =
*              CALL 'DSN8BCU2' USING WORKAREA-IND RECWORK.
*
*  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 = RECWORK
*                DESCRIPTION = WORK AREA FOR OUTPUT RECORD
*                01 RECWORK.
*                02 RECWORK-LEN PIC S9(8) COMP.
*
*                SYMBOLIC LABEL/NAME = SYSIN
*                DESCRIPTION = INPUT REQUESTS - TABLE OR VIEW
*
*

```

Figure 278. Called program that does pointer manipulation (Part 1 of 10)

```

*      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.                 *
*
* EXTERNAL REFERENCES =                                 *
* ROUTINES/SERVICES =                                  *
*   DSNTIAR - TRANSLATE SQLCA INTO MESSAGES            *
* DATA-AREAS = NONE                                   *
* CONTROL-BLOCKS =                                     *
*   SQLCA - SQL COMMUNICATION AREA                    *
*
* TABLES = NONE                                       *
* CHANGE-ACTIVITY = NONE                             *
*

```

Figure 278. Called program that does pointer manipulation (Part 2 of 10)

```

* *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.    DSN8BCU2
*
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).

```

Figure 278. Called program that does pointer manipulation (Part 3 of 10)

```

*
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  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  MSGRETCD.
    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 ' '.

```

Figure 278. Called program that does pointer manipulation (Part 4 of 10)

```

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 *
00
*****
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 *
*****
EXEC SQL INCLUDE SQLDA END-EXEC.

*
* 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 HWTYP             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.
*

```

Figure 278. Called program that does pointer manipulation (Part 5 of 10)

```

*****
*   THE REDEFINES CLAUSES BELOW ARE FOR 31-BIT ADDRESSING.
*   IF YOUR COMPILER SUPPORTS ONLY 24-BIT ADDRESSING,
*   CHANGE THE DECLARATIONS TO THESE:
*   01 RECNUM REDEFINES RECPTR PICTURE S9(8) COMPUTATIONAL.
*   01 IRECNUM REDEFINES IRECPTR PICTURE S9(8) COMPUTATIONAL.
*****
01 RECPTR POINTER.
01 RECNUM REDEFINES RECPTR PICTURE S9(9) COMPUTATIONAL.
01 IRECPTR POINTER.
01 IRECNUM REDEFINES IRECPTR PICTURE S9(9) 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 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).
/

```

Figure 278. Called program that does pointer manipulation (Part 6 of 10)


```

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).
*
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.
*
CLOSE SYSIN
SYSPRINT
SYSREC01.
GOBACK.

/
*****
*
* PERFORMED SECTION:
* PROCESSING FOR THE TABLE OR VIEW JUST READ
*
*****
PROCESS-INPUT.
*
MOVE TNAME TO STMTTAB.
MOVE STMTBLD TO STMTCHAR.
EXEC SQL PREPARE SEL INTO :SQLDA FROM :STMTBUF END-EXEC.
*****
*
* SET UP ADDRESSES IN THE SQLDA FOR DATA.
*
*****
IF SQLD = ZERO THEN
WRITE MSGREC FROM MSGNOCOL
AFTER ADVANCING 2 LINES
GO TO IND-RESULT.
MOVE ZERO TO ROWCOUNT.
MOVE ZERO TO REC1-LEN.
SET RECPTR TO IRECPTR.
MOVE ONE TO I.
PERFORM COLADDR UNTIL I > SQLD.

```

Figure 278. Called program that does pointer manipulation (Part 7 of 10)

```

*****
*
*   SET LENGTH OF OUTPUT RECORD.
*   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.
*
*****
*                               **OPEN CURSOR
*   EXEC SQL OPEN DT  END-EXEC.
*   PERFORM BLANK-REC.
*   EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.
*                               **NO ROWS FOUND
*                               **PRINT ERROR MESSAGE
*       IF SQLCODE = NOT-FOUND
*           WRITE MSGREC FROM MSG-NOROW
*               AFTER ADVANCING 2 LINES
*       ELSE
*                               **WRITE ROW AND
*                               **CONTINUE UNTIL
*                               **NO MORE ROWS
*           PERFORM WRITE-AND-FETCH
*               UNTIL SQLCODE IS NOT EQUAL TO ZERO.
*
*   EXEC SQL WHENEVER NOT FOUND GOTO CLOSEDT  END-EXEC.
*
*   CLOSEDT.
*   EXEC SQL CLOSE DT  END-EXEC.
*
*****
*
*   INDICATE THE RESULTS OF THE UNLOAD OPERATION.
*
*****
*   IND-RESULT.
*       MOVE TNAME TO TABLENAM.
*       MOVE ROWCOUNT TO ROWS.
*       WRITE MSGREC FROM UNLOADED
*           AFTER ADVANCING 2 LINES.
*       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.

```

Figure 278. Called program that does pointer manipulation (Part 8 of 10)

```

*****
*   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 ON 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 ON 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,
        WHEN VARLTYPE
            ADD TWO TO COLUMN-LEN,
        WHEN GTYPE
            MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN,
        WHEN VARGTYPE
            PERFORM CALC-VARG-LEN,
        WHEN LVARGTYP
            PERFORM CALC-VARG-LEN,
        WHEN HWTYPE
            MOVE TWO TO COLUMN-LEN,
        WHEN INTTYPE
            MOVE FOUR TO COLUMN-LEN,
        WHEN DECTYPE
            PERFORM CALC-DECIMAL-LEN,
        WHEN OTHER
            PERFORM UNRECOGNIZED-ERROR,
    END-EVALUATE.
    ADD COLUMN-LEN TO RECNUM.
    ADD COLUMN-LEN TO REC1-LEN.

```

Figure 278. Called program that does pointer manipulation (Part 9 of 10)

```

*****
*
*   IF THIS COLUMN CAN BE NULL, AN INDICATOR VARIABLE IS
*   NEEDED.  WE ALSO RESERVE SPACE IN THE OUTPUT RECORD TO
*   NOTE THAT THE VALUE IS NULL.
*
*****
      MOVE ZERO TO IND(I).
      IF COLUMN-IND = ONE THEN
        SET SQLIND(I) TO ADDRESS OF IND(I)
        SET WORKINDPTR(I) TO RECPT
        ADD ONE TO RECNUM
        ADD ONE TO REC1-LEN.
*
      ADD ONE TO I.
*   PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH
*   FOR A DECIMAL DATA TYPE COLUMN
      CALC-DECIMAL-LEN.
        DIVIDE COLUMN-LEN BY 256 GIVING COLUMN-PREC
          REMAINDER COLUMN-SCALE.
        MOVE COLUMN-PREC TO COLUMN-LEN.
        ADD ONE TO COLUMN-LEN.
        DIVIDE COLUMN-LEN BY TWO GIVING COLUMN-LEN.
*   PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH
*   FOR A VARGRAPHIC DATA TYPE COLUMN
      CALC-VARG-LEN.
        MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN.
        ADD TWO TO COLUMN-LEN.
*   PERFORMED PARAGRAPH TO NOTE AN UNRECOGNIZED
*   DATA TYPE COLUMN
      UNRECOGNIZED-ERROR.
*
*   ERROR MESSAGE FOR UNRECOGNIZED DATA TYPE
*
      MOVE SQLTYPE(I) TO TYPCOD.
      WRITE MSGREC FROM BADTYPE
        AFTER ADVANCING 2 LINES.
      GO TO IND-RESULT.
*
*****
*   SQL ERROR OCCURRED - GET MESSAGE
*****
      DBERROR.
*
*   **SQL ERROR
      MOVE SQLCODE TO MSG-PRINT-CODE.
      IF SQLCODE < 0 THEN MOVE '-' TO MSG-MINUS.
      WRITE MSGREC FROM MSG-SQLERR
        AFTER ADVANCING 2 LINES.
      CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
      IF RETURN-CODE = ZERO
        PERFORM ERROR-PRINT VARYING ERROR-INDEX
          FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 8
      ELSE
*
*   **ERROR FOUND IN DSNTIAR
*   **PRINT ERROR MESSAGE
      MOVE RETURN-CODE TO RETCODE
      WRITE MSGREC FROM MSGRETC
        AFTER ADVANCING 2 LINES.
      GO TO PROG-END.
*
*****
*   PRINT MESSAGE TEXT
*****
      ERROR-PRINT.
      WRITE MSGREC FROM ERROR-TEXT (ERROR-INDEX)
        AFTER ADVANCING 1 LINE.

```

Figure 278. Called program that does pointer manipulation (Part 10 of 10)

Sample dynamic and static SQL in a C program

Figure 279 on page 1044 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 */
/* */

```

Figure 279. Sample SQL in a C program (Part 1 of 4)

```

/* 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 "stddef.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
EXEC SQL BEGIN DECLARE SECTION;
short edlevel;
struct { short len;
        char x1[56];
        } stmtbf1, stmtbf2, 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];
        } 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) ) ;

```

Figure 279. Sample SQL in a C program (Part 2 of 4)

```

EXEC SQL DECLARE DEPT TABLE
(
    DEPTNO          CHAR(3)          ,
    DEPTNAME        VARCHAR(36)      ,
    MGRNO           CHAR(6)          ,
    ADMRDEPT        CHAR(3)          );

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,"123456");
    strcpy(hv2.admrdept,"abc");
    /*****
    /* 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;

```

Figure 279. Sample SQL in a C program (Part 3 of 4)

```

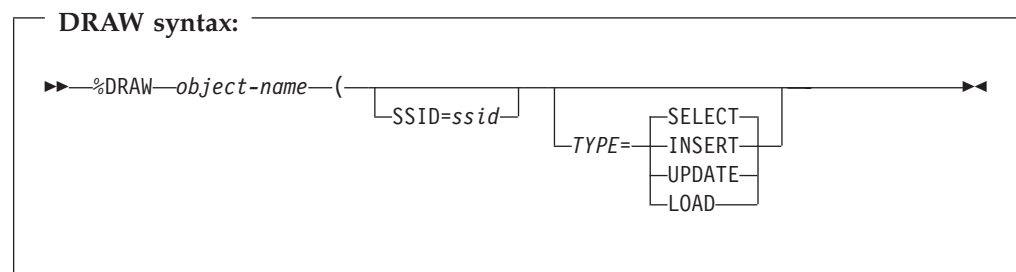
/*****
/* Dynamic SQL 3: PREPARE with parameter markers          */
/* Insert into with four values.                          */
*****/
sprintf (stmbf1.x1,
        "INSERT INTO DEPT VALUES (?, ?, ?, ?)");
stmbf1.len = strlen(stmbf1.x1);
printf("??/n***      begin prepare                ***");
EXEC SQL 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);
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);
progend: ;
}

```

Figure 279. Sample SQL in a C program (Part 4 of 4)

Sample DB2 REXX application

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 <i>object-name</i> . S can be used as an abbreviation for SELECT.
INSERT	Builds a template for an INSERT statement that inserts values into all columns of <i>object-name</i> . 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 <i>object-name</i> . 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 <i>object-name</i> . 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:

- 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 DSN8810.EMP at the local subsystem. Use the default DB2I subsystem ID.

The DRAW invocation is:

```
DRAW DSN8810.EMP (TYPE=SELECT
```

The output is:

```
SELECT "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" ,
      "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" ,
      "SALARY" , "BONUS" , "COMM"
FROM DSN8810.EMP
```

Generate a template for an INSERT statement that inserts values into table DSN8810.EMP at location SAN_JOSE. The local subsystem ID is DSN.

The DRAW invocation is:

```
DRAW SAN_JOSE.DSN8810.EMP (TYPE=INSERT SSID=DSN
```

The output is:

```
INSERT INTO SAN_JOSE.DSN8810.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 DSN8810.EMP. The local subsystem ID is DSN.

The DRAW invocation is:

```
DRAW DSN8810.EMP (TYPE=UPDATE SSID=DSN
```

The output is:

```
UPDATE DSN8810.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 DSN8810.EMP. The local subsystem ID is DSN.

The draw invocation is:

```
DRAW DSN8810.EMP (TYPE=LOAD SSID=DSN
```

The output is:

```

LOAD DATA INDDN SYSREC INTO TABLE DSN8810.EMP
( "EMPNO"          POSITION( 1) CHAR(6)
, "FIRSTNAME"     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 DSN8810.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.DSN8810.EMP

```

Figure 280. REXX sample program DRAW (Part 1 of 10)

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 DSN8810.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. See *DB2 SQL Reference* for more information on INSERT queries.

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:

Figure 280. REXX sample program DRAW (Part 2 of 10)

```

UPDATE DSN8810.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 don't need. Be sure to complete the WHERE clause. For information on writing queries to update data, refer to *DB2 SQL Reference*.

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 DSN8810 .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)='?'
)

```

Figure 280. REXX sample program DRAW (Part 3 of 10)

To use this LOAD statement, type the changes you want to make, and delete the lines you don't need. For information on writing queries to update data, refer to *DB2 Utility Guide and Reference*.

```

*/
  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
      zedsmg = '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

```

Figure 280. REXX sample program DRAW (Part 4 of 10)

```

SQLTYPE. = "UNKNOWN TYPE"
VCHTYPE = 448; SQLTYPES.VCHTYPE = 'VARCHAR'
CHTYPE = 452; SQLTYPES.CHTYPE = 'CHAR'
LVCHTYPE = 456; SQLTYPES.LVCHTYPE = 'VARCHAR'
VGRTYP = 464; SQLTYPES.VGRTYP = 'VARGRAPHIC'
GRTYP = 468; SQLTYPES.GRTYP = 'GRAPHIC'
LVGRTYP = 472; SQLTYPES.LVGRTYP = 'VARGRAPHIC'
FLOTYPE = 480; SQLTYPES.FLOTYPE = 'FLOAT'
DCTYPE = 484; SQLTYPES.DCTYPE = 'DECIMAL'
INTYPE = 496; SQLTYPES.INTYPE = 'INTEGER'
SMTYPE = 500; SQLTYPES.SMTYPE = 'SMALLINT'
DATYPE = 384; SQLTYPES.DATYPE = 'DATE'
TITYPE = 388; SQLTYPES.TITYPE = 'TIME'
TSTYPE = 392; SQLTYPES.TSTYPE = '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' = DATA LINE (Line)'
End
line1 = zdest + 1
'ISREDIT CURSOR = 'line1 0
Exit

```

Figure 280. REXX sample program DRAW (Part 5 of 10)


```

/*****/
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
  End

```

Figure 280. REXX sample program DRAW (Part 6 of 10)

```

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 = "                                ) --"
  Line = Line Left(SQLDA.I.SQLNAME,18)
  Type = SQLDA.I.SQLETYPE
  Null = Type//2
  If Null Then Type = Type - 1
  Len = SQLDA.I.SQLELEN
  Prdsn = SQLDA.I.SQLELEN.SQLEPRECISION
  Scale = SQLDA.I.SQLELEN.SQLESCALE
  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
Return

```

Figure 280. REXX sample program DRAW (Part 7 of 10)

```

/*****
/* 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.SQLLEN
      Prcsn = SQLDA.I.SQLLEN.SQLPRECISION
      Scale = SQLDA.I.SQLLEN.SQLSCALE
      Select
        When (Type = CHTYPE ,
              Type = VCHTYPE ,
              Type = LVCHTYPE ,
              Type = GRTYP ,
              Type = VGRTP ,
              Type = LVGRTP ) THEN
          Type = SQLTYPES.Type("STRIP(LEN)")
        When (Type = FLCTYPE ) 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
  L = Line.0 + 1; Line.0 = L
  Line.L = "WHERE"
  Return

```

Figure 280. REXX sample program DRAW (Part 8 of 10)

```

/*****
/* 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.SQlLen
    Prcsn = SQLDA.I.SQlLen.SQlPrecision
    Scale = SQLDA.I.SQlLen.SQlScale
    Select
      When (Type = CHType ,
            |Type = GRType ) THEN
        Type = SQLTYPES.Type("STRIP(LEN)")
      When (Type = FLType ) 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 = GRType ,
        |Type = VGRType ,
        |Type = LVGRType ) THEN
      Len = Len * 2
    If (Type = VCHType ,
        |Type = LVCHType ,
        |Type = VGRType ,
        |Type = LVGRType ) THEN
      Len = Len + 2
    Line = Line Type
    L = Line.0 + 1; Line.0 = L
  End Do

```

Figure 280. REXX sample program DRAW (Part 9 of 10)

```

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

```

Figure 280. REXX sample program DRAW (Part 10 of 10)

Sample COBOL program using DRDA access

The following sample program demonstrates distributed data access using DRDA access.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TWOPHASE.
AUTHOR.
REMARKS.
*****
*
* MODULE NAME = TWOPHASE
*
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING
*                   TWO PHASE COMMIT AND THE DRDA DISTRIBUTED
*                   ACCESS METHOD
*
* COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1989
* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083
*
* STATUS = VERSION 5
*
* 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.EMP 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
*
*
*
*
```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 1 of 8)

```

* 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.

```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 2 of 8)

```

*   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).

```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 3 of 8)

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,

```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 4 of 8)

```

        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.

```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 5 of 8)

```

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.

```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 6 of 8)

```

*****
* 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.

```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 7 of 8)

```

*****
* 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.

```

Figure 281. Sample COBOL two-phase commit application for DRDA access (Part 8 of 8)

Sample COBOL program using DB2 private protocol access

The following sample program demonstrates distributed access data using DB2 private protocol access with two-phase commit.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TWOPHASE.
AUTHOR.
REMARKS.
*****
*
* MODULE NAME = TWOPHASE
*
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING
*                   TWO PHASE COMMIT AND DB2 PRIVATE PROTOCOL
*                   DISTRIBUTED ACCESS METHOD
*
* COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1989
* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083
*
* STATUS = VERSION 5
*
* 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.EMP 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
*
*
```

Figure 282. Sample COBOL two-phase commit application for DB2 private protocol access (Part 1 of 7)

```

*
* 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.
*

```

Figure 282. Sample COBOL two-phase commit application for DB2 private protocol access (Part 2 of 7)

```

*      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.

```

Figure 282. Sample COBOL two-phase commit application for DB2 private protocol access (Part 3 of 7)


```

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 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   STLEC1.SYSADM.EMP
WHERE  EMPNO = :TEMP-EMPNO
END-EXEC.

```

Figure 282. Sample COBOL two-phase commit application for DB2 private protocol access (Part 4 of 7)

```

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.

```

Figure 282. Sample COBOL two-phase commit application for DB2 private protocol access (Part 5 of 7)

```

        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 STLEC1.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.

```

Figure 282. Sample COBOL two-phase commit application for DB2 private protocol access (Part 6 of 7)

```

*****
* 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 STLEC2.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.

```

Figure 282. Sample COBOL two-phase commit application for DB2 private protocol access (Part 7 of 7)

Examples of using stored procedures

This section contains sample programs that you can refer to when programming your stored procedure applications. DSN810.SDSNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

Calling a stored procedure from a C program

This example shows how to 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.

```

#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];
    char schema[9];
    long int out_code;

    struct {
        short int parmlen;
        char parmtxt[254];
    } parmlst;

    struct indicators {
        short int procnm_ind;
        short int schema_ind;
        short int out_code_ind;
        short int parmlst_ind;
    } parmind;

    struct sqlda *proc_da;
    struct sqlda *res_da;

    static volatile
    SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;

    EXEC SQL END DECLARE SECTION;
    /* INPUT parm -- PROCEDURE name */
    /* INPUT parm -- User's schema */
    /* OUTPUT -- SQLCODE from the */
    /* SELECT operation. */
    /* OUTPUT -- RUNOPTS values */
    /* for the matching row in */
    /* catalog table SYSROUTINES */
    /* Indicator variable structure */
    /* SQLDA for DESCRIBE PROCEDURE */
    /* SQLDA for DESCRIBE CURSOR */
    /* Locator variables */

```

Figure 283. Calling a stored procedure from a C program (Part 1 of 4)

```

/*****
/* 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 doesn't 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 = %d\n",SQLCODE);
printf("sqlca.sqlerrmc = ");
for(i=0;i<sqlca.sqlerrml;i++)
printf("%c",sqlca.sqlerrmc[i]);
printf("\n");
}

```

Figure 283. Calling a stored procedure from a C program (Part 2 of 4)

```

else                                /* If the CALL worked,          */
if(out_code!=0)                      /* Did GETPRML hit an error?    */
    printf("GETPRML failed due to RC = %d\n",out_code);
/*****
/* If everything worked, do the following:
/* - Print out the parameters returned.
/* - Retrieve the result sets returned.
*****/
else
{
    printf("RUNOPTS = %s\n",parmlst.parmtxt);
    /* 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;

```

Figure 283. Calling a stored procedure from a C program (Part 3 of 4)

```

/*****
/* 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;
}

```

Figure 283. Calling a stored procedure from a C program (Part 4 of 4)

Calling a stored procedure from a COBOL program

This example shows how to call a version of the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention from a COBOL program on an MVS system. Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets.


```

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.

```

Figure 284. Calling a stored procedure from a COBOL program (Part 1 of 3)

```

*****
* 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 DOESN'T 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.

```

Figure 284. Calling a stored procedure from a COBOL program (Part 2 of 3)

```

*           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(1)
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 PARMLLEN GIVING NUMLINES ROUNDED
*           FIND OUT HOW MANY LINES TO PRINT
PERFORM PARMPRT 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.

```

Figure 284. Calling a stored procedure from a COBOL program (Part 3 of 3)

Calling a stored procedure from a PL/I program

This example shows how to call a version of the GETPRML stored procedure that uses the GENERAL linkage convention from a PL/I program on an MVS system.

```

*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 doesn't 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);

```

Figure 285. Calling a stored procedure from a PL/I program (Part 1 of 2)

```

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 285. Calling a stored procedure from a PL/I program (Part 2 of 2)

C stored procedure: GENERAL

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;
```

```

#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;

```

Figure 286. A C stored procedure with linkage convention GENERAL (Part 1 of 2)

```

    /*****
    /* 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;
}

```

Figure 286. A C stored procedure with linkage convention GENERAL (Part 2 of 2)

C stored procedure: GENERAL WITH NULLS

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;
```

```

#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));

```

Figure 287. A C stored procedure with linkage convention GENERAL WITH NULLS (Part 1 of 2)


```

/*****
/* 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;
}

```

Figure 287. A C stored procedure with linkage convention GENERAL WITH NULLS (Part 2 of 2)

COBOL stored procedure: GENERAL

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.

```

Figure 288. A COBOL stored procedure with linkage convention GENERAL (Part 1 of 2)

```

*****
* 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.

```

Figure 288. A COBOL stored procedure with linkage convention GENERAL (Part 2 of 2)

COBOL stored procedure: GENERAL WITH NULLS

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;
```

```

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.

```

*Figure 289. A COBOL stored procedure with linkage convention GENERAL WITH NULLS
(Part 1 of 2)*

```

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.

```

Figure 289. A COBOL stored procedure with linkage convention GENERAL WITH NULLS (Part 2 of 2)

PL/I stored procedure: GENERAL

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;

*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 */
                          /* the matching row in */
                          /* SYSIBM.SYSROUTINES */
                          VARYING;

  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;

```

Figure 290. A PL/I stored procedure with linkage convention GENERAL

PL/I stored procedure: GENERAL WITH NULLS

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;

```

```

*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;

```

Figure 291. A PL/I stored procedure with linkage convention GENERAL WITH NULLS

Appendix E. Recursive common table expression examples

Bill of materials (BOM) applications are a common requirement in many business environments. Recursive SQL is very useful in creating BOM applications. To illustrate the some of the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by each part. For more information about recursive SQL, refer to “Using recursive SQL” on page 15.

For the examples in this appendix, 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 Table 199:

Table 199. 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 Table 200:

Table 200. 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
17	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 Table 201:

Table 201. Result table for example 2

PART	SUBPART	Total QTY Used
01	02	2
01	03	3
01	04	4
01	05	14
01	06	15

Table 201. Result table for example 2 (continued)

PART	SUBPART	Total QTY Used
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 Table 202:

Table 202. 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
02	2	06	6

Table 202. Result table for example 3 (continued)

PART	LEVEL	SUBPART	QUANTITY
03	2	07	6
04	2	08	10
04	2	09	11
06	2	12	10
06	2	13	10

Appendix F. REBIND subcommands for lists of plans or packages

If a list of packages or plans that you want to rebind is not easily specified using asterisks, you might be able to create the needed REBIND subcommands automatically, 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:

- Overview of the procedure for generating lists of REBIND commands
- “Sample SELECT statements for generating REBIND commands”
- “Sample JCL for running lists of REBIND commands” on page 1106

Overview of the procedure for generating lists of REBIND commands

The following list is an overview of the procedures for REBIND PLAN:

1. Use DSNTIAUL to generate the REBIND PLAN subcommands for the selected plans.
2. Use TSO edit commands to add TSO DSN commands to the sequential data set.
3. Use DSN to execute the REBIND PLAN subcommands for the selected plans.

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 SELECT statements for generating REBIND commands

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 rebind the selected plans or packages.

If the SELECT statement returns no qualifying rows, then DSNTIAUL does not generate REBIND subcommands.

The examples in this section generate REBIND subcommands that work in DB2 UDB for z/OS Version 8. You might need to modify the examples for prior releases of DB2 that do not allow all of the same syntax.

Example 1:

REBIND all plans without terminating because of unavailable resources.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
              CONCAT')'              ',1,45)
FROM SYSIBM.SYSPPLAN;
```

Example 2:

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 3:

REBIND all plans bound before a given date and time.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
              CONCAT')'              ',1,45)
FROM SYSIBM.SYSPPLAN
WHERE BINDDATE <= 'yyyymmdd' AND
      BINDTIME <= 'hhmmsssthh';
```

where *yyyymmdd* represents the date portion and *hhmmsssthh* represents the time portion of the timestamp string.

Example 4:

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 5:

REBIND all plans bound since a given date and time.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
CONCAT')' ,1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE >= 'yyyymmdd' AND
      BINDTIME >= 'hhmmsssthh';
```

where *yyyymmdd* represents the date portion and *hhmmsssthh* represents the time portion of the timestamp string.

Example 6:

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 7:

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 >= 'yyyymmdd' AND
 BINDTIME >= 'hhmmsssthh') AND
BINDDATE <= 'yyyymmdd' AND
BINDTIME <= 'hhmmsssthh');
```

where *yyyymmdd* represents the date portion and *hhmmsssthh* represents the time portion of the timestamp string.

Example 8:

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 9:

REBIND all invalid plans.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
CONCAT')' ,1,45)
FROM SYSIBM.SYSPLAN
WHERE VALID = 'N';
```

Example 10:

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 11:

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 12:

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

Figure 292 shows the JCL that is used to rebind all versions of all packages that are bound within the 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 Figure 292, 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: 2003.
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
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB81) PARM('SQL') -
LIB('DSN810.RUNLIB.LOAD')
END
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSPUNCH DD SYSOUT=*
//SYSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//          UNIT=SYSDA,DISP=SHR

```

Figure 292. Example JCL: Rebind all packages that were bound within a specified date and time period (Part 1 of 2)

```

//*****
//*
//* GENER= '<SUBCOMMANDS TO REBIND ALL PACKAGES BOUND IN 1994
//*
//*****
//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 *
DSNTEDIT 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=DSN810.DBRMLIB.DATA,
// DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD DSN=SYSADM.SYSTSIN.DATA,
// UNIT=SYSDA,DISP=SHR
/*

```

Figure 292. Example JCL: Rebind all packages that were bound within a specified date and time period (Part 2 of 2)

Figure 293 on page 1108 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(DSNTIB81) 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=DSN810.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
/*

```

Figure 293. Example JCL: Rebind selected plans with a different bind option

Appendix G. Reserved schema names and reserved words

There are restrictions on the use of certain names that are used by the database manager. In some cases, names are reserved and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs though not prevented by the database manager.

Reserved schema names

In general, for distinct types, user-defined functions, stored procedures, sequences, and triggers, schema names that begin with the prefix SYS are reserved. The schema name for these objects cannot begin with SYS with these exceptions:

- SYSADM. The schema name can be SYSADM for all these objects.
- SYSIBM. The schema name can be SYSIBM for procedures.
- SYSPROC. The schema name can be SYSPROC for procedures.
- SYSTOOLS. The schema name can be SYSTOOLS for distinct types, user-defined functions, procedures, and triggers if the user who executes the CREATE statement has the SYSADM or SYSCTRL privilege.

It is also recommended not to use SESSION name as a schema name.

Reserved words

Table 203 on page 1110 lists the words that cannot be used as ordinary identifiers in some contexts because they might be interpreted as SQL keywords. For example, ALL cannot be a column name in a SELECT statement. Each word, however, can be used as a delimited identifier in contexts where it otherwise cannot be used as an ordinary identifier. For example, if the quotation mark (") is the escape character that begins and ends delimited identifiers, "ALL" can appear as a column name in a SELECT statement. In addition, some sections of this book might indicate words that cannot be used in the specific context that is being described.

Table 203. SQL reserved words

ADD	DATABASE	HOUR	ON	SEQUENCE ²
AFTER	DAY	HOURS	OPEN	SELECT
ALL	DAYS	IF	OPTIMIZATION	SENSITIVE
ALLOCATE	DBINFO	IMMEDIATE	OPTIMIZE	SET
ALLOW	DECLARE	IN	OR	SIGNAL ²
ALTER	DEFAULT	INCLUSIVE ²	ORDER	SIMPLE
AND	DELETE	INDEX	OUT	SOME
ANY	DESCRIPTOR	INHERIT	OUTER	SOURCE
AS	DETERMINISTIC	INNER	PACKAGE	SPECIFIC
ASENSITIVE ²	DISALLOW	INOUT	PARAMETER	STANDARD
ASSOCIATE	DISTINCT	INSENSITIVE	PART	STATIC
ASUTIME	DO	INSERT	PADDED ²	STAY
AUDIT	DOUBLE	INTO	PARTITION ²	STOGROUP
AUX	DROP	IS	PARTITIONED ²	STORES
AUXILIARY	DSSIZE	ISOBID	PARTITIONING ²	STYLE
BEFORE	DYNAMIC	ITERATE ²	PATH	SUMMARY ²
BEGIN	EDITPROC	JAR	PIECESIZE	SYNONYM
BETWEEN	ELSE	JOIN	PLAN	SYSFUN
BUFFERPOOL	ELSEIF	KEY	PRECISION	SYSIBM
BY	ENCODING	LABEL	PREPARE	SYSPROC
CALL	ENCRYPTION ²	LANGUAGE	PREVVAL ²	SYSTEM
CAPTURE	END	LC_CTYPE	PRIQTY	TABLE
CASCADE	ENDING ²	LEAVE	PRIVILEGES	TABLESPACE
CASE	END-EXEC ¹	LEFT	PROCEDURE	THEN
CAST	ERASE	LIKE	PROGRAM	TO
CCSID	ESCAPE	LOCAL	PSID	TRIGGER
CHAR	EXCEPT	LOCALE	QUERY ²	UNDO
CHARACTER	EXCEPTION ²	LOCATOR	QUERYNO	UNION
CHECK	EXECUTE	LOCATORS	READS	UNIQUE
CLOSE	EXISTS	LOCK	REFERENCES	UNTIL
CLUSTER	EXIT	LOCKMAX	REFRESH ²	UPDATE
COLLECTION	EXPLAIN	LOCKSIZE	RESIGNAL ²	USER
COLLID	EXTERNAL	LONG	RELEASE	USING
COLUMN	FENCED	LOOP	RENAME	VALIDPROC
COMMENT	FETCH	MAINTAINED ²	REPEAT	VALUE ²
COMMIT	FIELDPROC	MATERIALIZED ²	RESTRICT	VALUES
CONCAT	FINAL	MICROSECOND	RESULT	VARIABLE ²
CONDITION	FOR	MICROSECONDS	RESULT_SET_LOCATOR	VARIANT
CONNECT	FREE	MINUTE	RETURN	VCAT
CONNECTION	FROM	MINUTES	RETURNS	VIEW
CONSTRAINT	FULL	MODIFIES	REVOKE	VOLATILE ²
CONTAINS	FUNCTION	MONTH	RIGHT	VOLUMES
CONTINUE	GENERATED	MONTHS	ROLLBACK	WHEN
CREATE	GET	NEXTVAL ²	ROWSET ²	WHENEVER
CURRENT	GLOBAL	NO	RUN	WHERE
CURRENT_DATE	GO	NONE ²	SAVEPOINT	WHILE
CURRENT_LC_CTYPE	GOTO	NOT	SCHEMA	WITH
CURRENT_PATH	GRANT	NULL	SCRATCHPAD	WLM
CURRENT_TIME	GROUP	NULLS	SECOND	XMLEMENT ²
CURRENT_TIMESTAMP	HANDLER	NUMPARTS	SECONDS	YEAR
CURSOR	HAVING	OBID	SECQTY ²	YEARS
DATA	HOLD ²	OF	SECURITY ²	

Notes:

1. COBOL only
2. New reserved word for Version 8.

IBM SQL has additional reserved words that DB2 UDB for z/OS does not enforce. Therefore, we suggest that you do not use these additional reserved words as ordinary identifiers in names that have a continuing use. See *IBM DB2 Universal Database SQL Reference for Cross-Platform Development* for a list of the words.

Appendix H. Characteristics of SQL statements in DB2 UDB for z/OS

Actions allowed on SQL statements

Table 204 shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler. The letter Y means *yes*.

Table 204. Actions allowed on SQL statements in DB2 UDB for z/OS

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
ALLOCATE CURSOR ¹	Y	Y	Y		
ALTER ²	Y	Y		Y	
ASSOCIATE LOCATORS ¹	Y	Y	Y		
BEGIN DECLARE SECTION					Y
CALL ¹	Y			Y	
CLOSE	Y			Y	
COMMENT	Y	Y		Y	
COMMIT ⁸	Y	Y		Y	
CONNECT	Y		Y		
CREATE ²	Y	Y		Y	
DECLARE CURSOR					Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y		Y	
DECLARE STATEMENT					Y
DECLARE TABLE					Y
DELETE	Y	Y		Y	
DESCRIBE prepared statement or table	Y			Y	
DESCRIBE CURSOR	Y		Y		
DESCRIBE INPUT	Y			Y	
DESCRIBE PROCEDURE	Y		Y		
DROP ²	Y	Y		Y	
END DECLARE SECTION					Y
EXECUTE	Y			Y	
EXECUTE IMMEDIATE	Y			Y	
EXPLAIN	Y	Y		Y	
FETCH	Y			Y	
FREE LOCATOR ¹	Y	Y		Y	
I GET DIAGNOSTICS	Y			Y	
GRANT ²	Y	Y		Y	

Table 204. Actions allowed on SQL statements in DB2 UDB for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
HOLD LOCATOR ¹	Y	Y		Y	
INCLUDE					Y
INSERT	Y	Y		Y	
LABEL	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
OPEN	Y			Y	
PREPARE	Y			Y ⁴	
REFRESH TABLE	Y	Y		Y	
RELEASE connection	Y		Y		
RELEASE SAVEPOINT	Y	Y		Y	
RENAME ²	Y	Y		Y	
REVOKE ²	Y	Y		Y	
ROLLBACK ⁸	Y	Y		Y	
SAVEPOINT	Y	Y		Y	
SELECT INTO	Y			Y	
SET CONNECTION	Y		Y		
SET CURRENT APPLICATION ENCODING SCHEME	Y		Y		
SET CURRENT DEGREE	Y	Y		Y	
SET CURRENT LC_CTYPE	Y	Y		Y	
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Y	Y		Y	
SET CURRENT OPTIMIZATION HINT	Y	Y		Y	
SET CURRENT PACKAGE PATH	Y		Y		
SET CURRENT PACKAGESET	Y			Y	
SET CURRENT PRECISION	Y	Y		Y	
SET CURRENT REFRESH AGE	Y	Y		Y	
SET CURRENT RULES	Y	Y		Y	
SET CURRENT SQLID ⁵	Y	Y		Y	
SET <i>host-variable</i> = CURRENT APPLICATION ENCODING SCHEME	Y	Y	Y		
SET <i>host-variable</i> = CURRENT DATE	Y			Y	
SET <i>host-variable</i> = CURRENT DEGREE	Y			Y	
SET <i>host-variable</i> = CURRENT MEMBER	Y			Y	

Table 204. Actions allowed on SQL statements in DB2 UDB for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
SET <i>host-variable</i> = CURRENT PACKAGESET	Y		Y		
SET <i>host-variable</i> = CURRENT PATH	Y			Y	
SET <i>host-variable</i> = CURRENT QUERY OPTIMIZATION LEVEL	Y			Y	
SET <i>host-variable</i> = CURRENT SERVER	Y		Y		
SET <i>host-variable</i> = CURRENT SQLID	Y			Y	
SET <i>host-variable</i> = CURRENT TIME	Y			Y	
SET <i>host-variable</i> = CURRENT TIMESTAMP	Y			Y	
SET <i>host-variable</i> = CURRENT TIMEZONE	Y			Y	
SET PATH	Y	Y		Y	
I SET SCHEMA	Y	Y		Y	
SET <i>transition-variable</i> = CURRENT DATE	Y			Y	
SET <i>transition-variable</i> = CURRENT DEGREE	Y			Y	
SET <i>transition-variable</i> = CURRENT PATH	Y			Y	
SET <i>transition-variable</i> = CURRENT QUERY OPTIMIZATION LEVEL	Y			Y	
SET <i>transition-variable</i> = CURRENT SQLID	Y			Y	
SET <i>transition-variable</i> = CURRENT TIME	Y			Y	
SET <i>transition-variable</i> = CURRENT TIMESTAMP	Y			Y	
SET <i>transition-variable</i> = CURRENT TIMEZONE	Y			Y	
SIGNAL SQLSTATE ⁶	Y			Y	
UPDATE	Y	Y		Y	
VALUES ⁶	Y			Y	
VALUES INTO ⁷	Y			Y	
WHENEVER					Y

Table 204. Actions allowed on SQL statements in DB2 UDB for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
Notes:					
1. The statement can be dynamically prepared. It cannot be issued dynamically.					
2. The statement can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.					
3. The statement can be dynamically prepared, but only from an ODBC or CLI driver that supports dynamic CALL statements.					
4. The requesting system processes the PREPARE statement when the statement being prepared is ALLOCATE CURSOR or ASSOCIATE LOCATORS.					
5. The value to which special register CURRENT SQLID is set is used as the SQL authorization ID and the implicit qualifier for dynamic SQL statements only when DYNAMICRULES run behavior is in effect. The CURRENT SQLID value is ignored for the other DYNAMICRULES behaviors.					
6. This statement can be used only in the triggered action of a trigger.					
7. Local special registers can be referenced in a VALUES INTO statement if it results in the assignment of a single host-variable, not if it results in setting more than one value.					
8. Some processing also occurs at the requester.					

SQL statements allowed in external functions and stored procedures

Table 205 shows which SQL statements in an external stored procedure or in an external user-defined function can execute. Whether the statements can be executed depends on the level of SQL data access with which the stored procedure or external function is defined (NO SQL, CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA). The letter **Y** means *yes*.

In general, if an executable SQL statement is encountered in a stored procedure or function defined as NO SQL, SQLSTATE 38001 is returned. If the routine is defined to allow some level of SQL access, SQL statements that are not supported in any context return SQLSTATE 38003. SQL statements not allowed for routines defined as CONTAINS SQL return SQLSTATE 38004, and SQL statements not allowed for READS SQL DATA return SQL STATE 38002.

Table 205. SQL statements in external user-defined functions and stored procedures

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALLOCATE CURSOR			Y	Y
ALTER				Y
ASSOCIATE LOCATORS			Y	Y
BEGIN DECLARE SECTION	Y ¹	Y	Y	Y
CALL		Y ²	Y ²	Y ²
CLOSE			Y	Y
COMMENT				Y
COMMIT ³		Y	Y	Y
CONNECT		Y	Y	Y
CREATE				Y

Table 205. SQL statements in external user-defined functions and stored procedures (continued)

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
DECLARE CURSOR	Y ¹	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE				Y
DECLARE STATEMENT	Y ¹	Y	Y	Y
DECLARE TABLE	Y ¹	Y	Y	Y
DELETE				Y
DESCRIBE			Y	Y
DESCRIBE CURSOR			Y	Y
DESCRIBE INPUT			Y	Y
DESCRIBE PROCEDURE			Y	Y
DROP				Y
END DECLARE SECTION	Y ¹	Y	Y	Y
EXECUTE		Y ⁴	Y ⁴	Y
EXECUTE IMMEDIATE		Y ⁴	Y ⁴	Y
EXPLAIN				Y
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GET DIAGNOSTICS		Y	Y	Y
GRANT				Y
HOLD LOCATOR		Y	Y	Y
INCLUDE	Y ¹	Y	Y	Y
INSERT				Y
LABEL				Y
LOCK TABLE		Y	Y	Y
OPEN			Y	Y
PREPARE		Y	Y	Y
REFRESH TABLE				Y
RELEASE connection		Y	Y	Y
RELEASE SAVEPOINT ⁶				Y
REVOKE				Y
ROLLBACK ^{6, 7, 8}		Y	Y	Y
ROLLBACK TO SAVEPOINT ^{6, 7, 8}				Y
SAVEPOINT ⁶				Y
SELECT			Y ⁹	Y
SELECT INTO			Y	Y
SET CONNECTION		Y	Y	Y
SET host-variable Assignment		Y ⁵	Y	Y

Table 205. SQL statements in external user-defined functions and stored procedures (continued)

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
SET special register		Y	Y	Y
SET transition-variable Assignment		Y ⁵	Y	Y
SIGNAL SQLSTATE		Y	Y	Y
UPDATE				Y
VALUES			Y	Y
VALUES INTO		Y ⁵	Y	Y
WHENEVER	Y ¹	Y	Y	Y

Notes:

1. Although the SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. The stored procedure that is called must have the same or more restrictive level of SQL data access than the current level in effect. For example, a routine defined as MODIFIES SQL DATA can call a stored procedure defined as MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL. A routine defined as CONTAINS SQL can only call a procedure defined as CONTAINS SQL.
3. The COMMIT statement cannot be executed in a user-defined function. The COMMIT statement cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.
4. The statement specified for the EXECUTE statement must be a statement that is allowed for the particular level of SQL data access in effect. For example, if the level in effect is READS SQL DATA, the statement must not be an INSERT, UPDATE, or DELETE.
5. The statement is supported only if it does not contain a subquery or query-expression.
6. RELEASE SAVEPOINT, SAVEPOINT, and ROLLBACK (with the TO SAVEPOINT clause) cannot be executed from a user-defined function.
7. If the ROLLBACK statement (without the TO SAVEPOINT clause) is executed in a user-defined function, an error is returned to the calling program, and the application is placed in a *must rollback* state.
8. The ROLLBACK statement (without the TO SAVEPOINT clause) cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.
9. If the SELECT statement contains an INSERT in its FROM clause (INSERT within SELECT), the SQL level of access must be MODIFIES SQL DATA.

SQL statements allowed in SQL procedures

Table 206 on page 1119 lists the statements that are valid in an SQL procedure body, in addition to SQL procedure statements. The table lists the statements that can be used as the only statement in the SQL procedure and as the statements that can be nested in a compound statement. An SQL statement can be executed in an SQL procedure depending on whether MODIFIES SQL DATA, CONTAINS SQL, or READS SQL DATA is specified in the stored procedure definition. See Table 205 on page 1116 for a list of SQL statements that can be executed for each of these parameter values.

Table 206. Valid SQL statements in an SQL procedure body

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
ALLOCATE CURSOR		Y
ALTER DATABASE	Y	Y
ALTER FUNCTION	Y	Y
ALTER INDEX	Y	Y
ALTER PROCEDURE	Y	Y
ALTER SEQUENCE	Y	Y
ALTER STOGROUP	Y	Y
ALTER TABLE	Y	Y
ALTER TABLESPACE	Y	Y
ALTER VIEW	Y	Y
ASSOCIATE LOCATORS		Y
BEGIN DECLARE SECTION		
CALL		Y
CLOSE		Y
COMMENT	Y	Y
COMMIT ¹	Y	Y
CONNECT	Y	Y
CREATE ALIAS	Y	Y
CREATE DATABASE	Y	Y
CREATE DISTINCT TYPE	Y	Y
CREATE FUNCTION ²	Y	Y
CREATE GLOBAL TEMPORARY TABLE	Y	Y
CREATE INDEX	Y	Y
CREATE PROCEDURE ²	Y	Y
CREATE SEQUENCE	Y	Y
CREATE STOGROUP	Y	Y
CREATE SYNONYM	Y	Y
CREATE TABLE	Y	Y
CREATE TABLESPACE	Y	Y
CREATE TRIGGER		
CREATE VIEW	Y	Y
DECLARE CURSOR		Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y
DECLARE STATEMENT		
DECLARE TABLE		
DELETE	Y	Y
DESCRIBE prepared statement or table		
DESCRIBE CURSOR		

Table 206. Valid SQL statements in an SQL procedure body (continued)

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
DESCRIBE INPUT		
DESCRIBE PROCEDURE		
DROP	Y	Y
END DECLARE SECTION		
EXECUTE		Y
EXECUTE IMMEDIATE	Y	Y
EXPLAIN		
FETCH		Y
FREE LOCATOR		
GET DIAGNOSTICS	Y	Y
GRANT	Y	Y
HOLD LOCATOR		
INCLUDE		
INSERT	Y	Y
LABEL	Y	Y
LOCK TABLE	Y	Y
OPEN		Y
PREPARE FROM		Y
REFRESH TABLE	Y	Y
RELEASE connection	Y	Y
RELEASE SAVEPOINT	Y	Y
RENAME	Y	Y
REVOKE	Y	Y
ROLLBACK ¹	Y	Y
ROLLBACK TO SAVEPOINT ¹	Y	Y
SAVEPOINT	Y	Y
SELECT		
SELECT INTO	Y	Y
SET CONNECTION	Y	Y
SET host-variable Assignment ³		
SET special register ^{3, 4}	Y	Y
SET transition-variable Assignment ³		
SIGNAL SQLSTATE		
UPDATE	Y	Y
VALUES		
VALUES INTO	Y	Y
WHENEVER		

Table 206. Valid SQL statements in an SQL procedure body (continued)

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
Notes:		
1. The COMMIT statement and the ROLLBACK statement (without the TO SAVEPOINT clause) cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger		
2. CREATE FUNCTION with LANGUAGE SQL (specified either implicitly or explicitly) and CREATE PROCEDURE with LANGUAGE SQL are not allowed within the body of an SQL procedure.		
3. SET host-variable assignment, SET transition-variable assignment, and SET special register are the SQL SET statements not the SQL procedure assignment statement		
4. The SET SCHEMA statement cannot be executed within a SQL procedure.		

Appendix I. Program preparation options for remote packages

Table 207 gives generic descriptions of program preparation options, lists the equivalent DB2 option for each one, and indicates if appropriate, whether it is a bind package (B) or a precompiler (P) option. In addition, the table indicates whether a DB2 server supports the option.

Table 207. Program preparation options for packages

Generic option description	Equivalent for Requesting DB2	Bind or Precompile Option	DB2 Server Support
Package replacement: protect existing packages	ACTION(ADD)	B	Supported
Package replacement: replace existing packages	ACTION(REPLACE)	B	Supported
Package replacement: version name	ACTION(REPLACE REPLVER (version-id))	B	Supported
Statement string delimiter	APOSTSQL/QUOTESQL	P	Supported
DRDA access: SQL CONNECT (Type 1)	CONNECT(1)	P	Supported
DRDA access: SQL CONNECT (Type 2)	CONNECT(2)	P	Supported
Block protocol: Do not block data for an ambiguous cursor	CURRENTDATA(YES)	B	Supported
Block protocol: Block data when possible	CURRENTDATA(NO)	B	Supported
Block protocol: Never block data	(Not available)		Not supported
Name of remote database	CURRENTSERVER(location name)	B	Supported as a BIND PLAN option
Date format of statement	DATE	P	Supported
Protocol for remote access	DBPROTOCOL	B	Not supported
Maximum decimal precision: 15	DEC(15)	P	Supported
Maximum decimal precision: 31	DEC(31)	P	Supported
Defer preparation of dynamic SQL	DEFER(PREPARE)	B	Supported
Do not defer preparation of dynamic SQL	NODEFER(PREPARE)	B	Supported
Dynamic SQL Authorization	DYNAMICRULES	B	Supported
Encoding scheme for static SQL statements	ENCODING	B	Not supported
Explain option	EXPLAIN	B	Supported
Immediately write group bufferpool-dependent page sets or partitions in a data sharing environment	IMMEDWRITE	B	Supported
Package isolation level: CS	ISOLATION(CS)	B	Supported

Table 207. Program preparation options for packages (continued)

Generic option description	Equivalent for Requesting DB2	Bind or Precompile Option	DB2 Server Support
Package isolation level: RR	ISOLATION(RR)	B	Supported
Package isolation level: RS	ISOLATION(RS)	B	Supported
Package isolation level: UR	ISOLATION(UR)	B	Supported
Keep prepared statements after commit points	KEEPDYNAMIC	B	Supported
Consistency token	LEVEL	P	Supported
Package name	MEMBER	B	Supported
Package owner	OWNER	B	Supported
Schema name list for user-defined functions, distinct types, and stored procedures	PATH	B	Supported
Statement decimal delimiter	PERIOD/COMMA	P	Supported
Default qualifier	QUALIFIER	B	Supported
Use access path hints	OPTHINT	B	Supported
Lock release option	RELEASE	B	Supported
Choose access path at each run time	REOPT(ALWAYS)	B	Supported
Choose access path at bind time only	REOPT(NONE)	B	Supported
Choose and cache access path at only the first run or open time	REOPT(ONCE)	B	Supported
Creation control: create a package despite errors	SQLERROR(CONTINUE)	B	Supported
Creation control: create no package if there are errors	SQLERROR(NO PACKAGE)	B	Supported
Creation control: create no package	(Not available)		Supported
Time format of statement	TIME	P	Supported
Existence checking: full	VALIDATE(BIND)	B	Supported
Existence checking: deferred	VALIDATE(RUN)	B	Supported
Package version	VERSION	P	Supported
Default character subtype: system default	(Not available)		Supported
Default character subtype: BIT	(Not available)		Not supported
Default character subtype: SBCS	(Not available)		Not supported
Default character subtype: DBCS	(Not available)		Not supported
Default character CCSID: SBCS	(Not available)		Not supported
Default character CCSID: Mixed	(Not available)		Not supported
Default character CCSID: Graphic	(Not available)		Not supported
Package label	(Not available)		Ignored when received; no error is returned
Privilege inheritance: retain	default		Supported

Table 207. Program preparation options for packages (continued)

Generic option description	Equivalent for Requesting DB2	Bind or Precompile Option	DB2 Server Support
Privilege inheritance: revoke	(Not available)		Not supported

Appendix J. DB2-supplied stored procedures

DB2 provides several stored procedures that you can call in your application programs to perform a number of utility and application programming functions. Those stored procedures are:

- The utilities stored procedure for EBCDIC input (DSNUTILS)

This stored procedure lets you invoke utilities from a local or remote client program. See Appendix B of *DB2 Utility Guide and Reference* for information.

- The utilities stored procedure for Unicode input (DSNUTILU)

This stored procedure lets you invoke utilities from a local or remote client program that generates Unicode utility control statements. See Appendix B of *DB2 Utility Guide and Reference* for information.

- The DB2 Universal Database Control Center (Control Center) table space and index information stored procedure (DSNACCQC)

This stored procedure helps you determine when utilities should be run on your databases. This stored procedure is designed primarily for use by the Control Center but can be invoked from any client program. See Appendix B of Appendix B of *DB2 Utility Guide and Reference* for information.

- The Control Center partition information stored procedure (DSNACCAV)

This stored procedure helps you determine when utilities should be run on your partitioned table spaces. This stored procedure is designed primarily for use by the Control Center but can be invoked from any client program. See Appendix B of *DB2 Utility Guide and Reference* for information.

- The real-time statistics stored procedure (DSNACCOR)

This stored procedure queries the DB2 real-time statistics tables to help you determine when you should run COPY, REORG, or RUNSTATS, or enlarge your DB2 data sets. See Appendix B of *DB2 Utility Guide and Reference* for more information.

- The WLM environment refresh stored procedure (WLM_REFRESH)

This stored procedure lets you refresh a WLM environment from a remote workstation. See “WLM environment refresh stored procedure (WLM_REFRESH)” on page 1129 for information.

- The CICS transaction invocation stored procedure (DSNACICS)

This stored procedure lets you invoke CICS transactions from a remote workstation. See “The CICS transaction invocation stored procedure (DSNACICS)” on page 1131 for more information.

- The SYSIBM.USERNAMES encryption stored procedure (DSNLEUSR)

This stored procedure lets you store encrypted values in the NEWAUTHID and PASSWORD fields of the SYSIBM.USERNAMES catalog table. See Appendix I of *DB2 Administration Guide* for more information.

- The IMS transactions stored procedure (DSNAIMS)

This stored procedure allows DB2 to invoke IMS transactions and commands easily, without maintaining their own connections to IMS. See “IMS transactions stored procedure (DSNAIMS)” on page 1139 for more information.

- The EXPLAIN stored procedure (DSN8EXP)

This stored procedure allows a user to perform an EXPLAIN on an SQL statement without having the authorization to execute that SQL statement. See “The DB2 EXPLAIN stored procedure” on page 1143 for more information.

- The MQ XML stored procedures
- All of the MQ XML stored procedures have been deprecated.
- These stored procedures perform the following functions:

Table 208. MQ XML stored procedures

Stored procedure name	Function	For information, see:
DXXMQINSERT	Returns a message that contains an XML document from an MQ message queue, decomposes the document, and stores the data in DB2 tables that are specified by an enabled XML collection.	“Deprecated: Store an XML document from an MQ message queue in DB2 tables (DXXMQINSERT)” on page 1146
DXXMQSHRED	Returns a message that contains an XML document from an MQ message queue, decomposes the document, and stores the data in DB2 tables that are specified in a document access definition (DAD) file. DXXMQSHRED does not require an enabled XML collection.	“Deprecated: Store an XML document from an MQ message queue in DB2 tables (DXXMQSHRED)” on page 1148
DXXMQINSERTCLOB	Returns a message that contains an XML document from an MQ message queue, decomposes the document, and stores the data in DB2 tables that are specified by an enabled XML collection. DXXMQINSERTCLOB is intended for an XML document with a length of up to 1MB.	“Deprecated: Store a large XML document from an MQ message queue in DB2 tables (DXXMQINSERTCLOB)” on page 1151
DXXMQSHREDCLOB	Returns a message that contains an XML document from an MQ message queue, decomposes the document, and stores the data in DB2 tables that are specified in a document access definition (DAD) file. DXXMQSHREDCLOB does not require an enabled XML collection. DXXMQSHREDCLOB is intended for an XML document with a length of up to 1MB.	“Deprecated: Store a large XML document from an MQ message queue in DB2 tables (DXXMQSHREDCLOB)” on page 1153
DXXMQINSERTALL	Returns messages that contains XML documents from an MQ message queue, decomposes the documents, and stores the data in DB2 tables that are specified by an enabled XML collection. DXXMQINSERTALL is intended for XML documents with a length of up to 3KB.	“Deprecated: Store XML documents from an MQ message queue in DB2 tables (DXXMQINSERTALL)” on page 1156
DXXMQSHREDALL	Returns messages that contain XML documents from an MQ message queue, decomposes the documents, and stores the data in DB2 tables that are specified in a document access definition (DAD) file. DXXMQSHREDALL does not require an enabled XML collection. DXXMQSHREDALL is intended for XML documents with a length of up to 3KB.	“Deprecated: Store XML documents from an MQ message queue in DB2 tables (DXXMQSHREDALL)” on page 1158
DXXMQSHREDALLCLOB	Returns messages that contain XML documents from an MQ message queue, decomposes the documents, and stores the data in DB2 tables that are specified in a document access definition (DAD) file. DXXMQSHREDALLCLOB does not require an enabled XML collection. DXXMQSHREDALLCLOB is intended for XML documents with a length of up to 1MB.	“Deprecated: Store large XML documents from an MQ message queue in DB2 tables (DXXMQSHREDALLCLOB)” on page 1161

Table 208. MQ XML stored procedures (continued)

#	Stored procedure name	Function	For information, see:
#	DXXMQINSERTALLCLOB	Returns messages that contains XML documents from an MQ message queue, decomposes the documents, and stores the data in DB2 tables that are specified by an enabled XML collection. DXXMQINSERTALLCLOB is intended for XML documents with a length of up to 1MB.	“Deprecated: Store large XML documents from an MQ message queue in DB2 tables (DXXMQINSERTALLCLOB)” on page 1163
#	DXXMQGEN	Constructs XML documents from data that is stored in DB2 tables that are specified in a document access definition (DAD) file, and sends the XML documents to an MQ message queue. DXXMQGEN is intended for XML documents with a length of up to 3KB.	“Deprecated: Send XML documents to an MQ message queue (DXXMQGEN)” on page 1165
#	DXXMQRETRIEVE	Constructs XML documents from data that is stored in DB2 tables that are specified in an enabled XML collection, and sends the XML documents to an MQ message queue. DXXMQRETRIEVE is intended for XML documents with a length of up to 3KB.	“Deprecated: Send XML documents to an MQ message queue (DXXMQRETRIEVE)” on page 1169
#	DXXMQGENCLOB	Constructs XML documents from data that is stored in DB2 tables that are specified in a document access definition (DAD) file, and sends the XML documents to an MQ message queue. DXXMQGENCLOB is intended for XML documents with a length of up to 32KB.	“Deprecated: Send large XML documents to an MQ message queue (DXXMQGENCLOB)” on page 1173
#	DXXMQRETRIEVECLOB	Constructs XML documents from data that is stored in DB2 tables that are specified in an enabled XML collection, and sends the XML documents to an MQ message queue. DXXMQRETRIEVECLOB is intended for XML documents with a length of up to 32KB.	“Deprecated: Send XML documents to an MQ message queue (DXXMQRETRIEVECLOB)” on page 1176
#			
#			

WLM environment refresh stored procedure (WLM_REFRESH)

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.

Environment for WLM_REFRESH

WLM_REFRESH runs in a WLM-established stored procedures address space. The load module for WLM_REFRESH, DSNTWR, must reside in an APF-authorized library.

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.

See Part 3 (Volume 1) *DB2 Administration Guide* for information about authorizing access to SAF resource profiles. See *z/OS MVS Planning: Operations* for more information about permitting access to the extended MCS console.

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.

```

CALL WLM_REFRESH (—WLM-environment, —ssid—, —status-message, —return-code—)

```

The diagram shows the syntax for the CALL statement. The parameters are: WLM-environment, ssid, status-message, and return-code. The ssid parameter has a NULL option. The diagram is enclosed in a box with arrows at the ends.

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 <i>ssid.WLM_REFRESH.wlm-environment</i> 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 <i>WLM-environment</i> parameter value is null, blank, or contains invalid characters. • The <i>ssid</i> 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 DSNTEJ6W, which is in data set DSN810.SDSNSAMP.

The CICS transaction invocation stored procedure (DSNACICS)

The CICS transaction invocation stored procedure (DSNACICS) invokes CICS server programs. 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 UDB 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 for DSNACICS

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 required for DSNACICS

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:

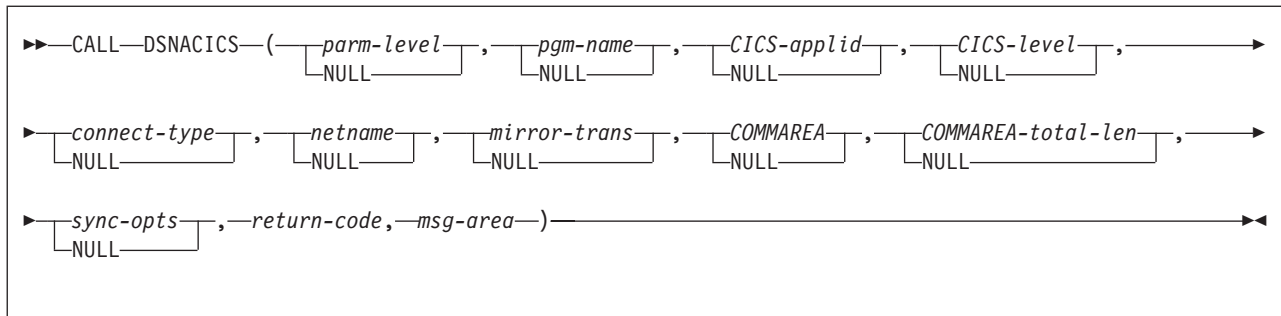
- The EXECUTE privilege on stored procedure DSNACICS
- Ownership of the stored procedure
- SYSADM authority

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. See Part 2 of *DB2 Installation Guide*.

The DSNACICS caller also needs authorization from an external security system, such as RACF, to use CICS resources. See Part 2 of *DB2 Installation Guide*.

DSNACICS syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DSNACICS. 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.



DSNACICS 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 UDB 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).

DSNACICX 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*.

General considerations for DSNACICX

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 DSNACICX 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 DSNACICX 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 DSNACICX 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 for DSNACICX

At invocation, registers are set as described in Table 209.

Table 209. 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.

Table 210 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 210. 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>
XPL_RETCODE	48	4-byte integer	Return code from the exit routine	<i>return-code</i>
XPL_MSGLEN	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

Table 210. Contents of the XPL exit parameter list (continued)

Name	Hex offset	Data type	Description	Corresponding DSNACICS parameter
Note:				
1. The area that this field points to is specified by DSNACICS parameter <i>COMMAREA</i> . 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 <i>COMMAREA</i> parameter.				
3. Although the total length of <i>msg-area</i> is 500 bytes, DSNACICX can use only 256 bytes of that area.				

Example of DSNACICS invocation

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;

DECLARE 1 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 BIN FIXED(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.
*****/

```

```

/*****
PARAM_LEVEL = 1;
IND_PARAM_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 CICSPGM1.      */
/*****
EXEC SQL
    CALL SYSPROC.DSNACICS(:PARAM_LEVEL           :IND_PARAM_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_RETCODE,
                        :MSG_AREA                  :IND_MSG_AREA);

```

DSNACICS 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.

DSNACICS 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. The list of supported commands is documented in CICS Transaction Server for z/OS Application Programming Reference.

DSNACICS 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.

IMS transactions stored procedure (DSNAIMS)

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 for DSNAIMS

DSNAIMS runs in a WLM-established stored procedures address space. DSNAIMS
requires DB2 with RRSAPF 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 required for DSNAIMS

To set up and run DSNAIMS, you must be authorized to 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. Refer to IMS Open Transaction Manager
Access Guide and Reference for an explanation of the C/I initialization.

DSNAIMS syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking
DSNAIMS.

```

# ►►CALL—SYSPROC.DSNAIMS—(—dsnaims-function—, —dsnaims-2pc—, —xcf-group-name—, —————►
#                                     NULL
#
# ►—xcf-ims-name—, —racf-userid—, —racf-groupid—, —ims-lterm—, —ims-modname—, —————►
#                                     NULL          NULL          NULL
#
# ►—ims-tran-name—, —ims-data-in—, —ims-data-out—, —otma-tpipe-name—, —————►
#   NULL          NULL          NULL          NULL
#
# ►—otma-dru-name—, —user-data-in—, —user-data-out—, —status-message—, —return-code—)►
#   NULL          NULL
#
#
#

```

DSNAIMS option descriptions

dsnaims-function

A string that indicates whether the transaction is send-only, receive-only, or send-and-receive:

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 DSAIMS. 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. *racf_groupid* 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.
#
# 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.
#
# 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 of DSNAIMS invocation

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 SYSPROCC.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.

Example:

```
# CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMS7GRP", "IMS7TMEM", ...)
# CALL SYSPROC.DSNAIMSB("SENDRECV", "N", "IMS8GRP", "IMS8TMEM", ...)
```

The DB2 EXPLAIN stored procedure

The information under this heading is Product-sensitive Programming Interface and Associated Guidance Information, as defined in “Notices” on page 1183.

The DSNAEXP stored procedure is a sample stored procedure that lets users perform an EXPLAIN on an SQL statement without having the authorization to execute that SQL statement.

Note: The DSNAEXP stored procedure replaces the DSN8EXP stored procedure for DB2 Version 8 new function mode. In DB2 Version 8 compatibility mode you must use the DSN8EXP stored procedure.

Environment

DSNAEXP must run in a WLM-established stored procedure address space.

Before you can invoke DSNAEXP, table *sqlid*.PLAN_TABLE must exist. *sqlid* is the value that you specify for the *sqlid* input parameter when you call DSNAEXP.

Job DSNTESC in DSN8810.SDSNSAMP contains a sample CREATE TABLE statement for the PLAN_TABLE.

Authorization required

To execute the CALL DSN8.DSNAEXP 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 DSNAEXP
- Ownership of the package
- PACKADM authority for the package collection
- SYSADM authority

In addition:

- The SQL authorization ID of the process in which DSNAEXP is called must have the authority to execute SET CURRENT SQLID=*sqlid*.
- The SQL authorization ID of the process must also have one of the following characteristics:
 - Be the owner of a plan table named PLAN_TABLE
 - Have an alias on a plan table named *owner*.PLAN_TABLE and have SELECT and INSERT privileges on the table

DSNAEXP syntax diagram

The following syntax diagram shows the CALL statement for invoking DSNAEXP. Because the linkage convention for DSNAEXP is GENERAL, you cannot pass null values to the stored procedure.

```
CALL DSNAEXP (—sqlid, —queryno, —sql-statement, —parse, —qualifier, —sqlcode, —sqlstate, —  
—error-message—)
```

DSNAEXP option descriptions

sqlid

Specifies:

- The authorization ID under which the EXPLAIN statement is to be executed
- The qualifier for the table into which EXPLAIN output is written:
sqlid.PLAN_TABLE
- The implicit qualifier for unqualified objects in the SQL statement on which EXPLAIN is executed, if the value of *parse* is 'N'

sqlid is an input parameter of type CHAR(8).

queryno

Specifies a number that is to be used to identify *sql-statement* in the EXPLAIN output. This number appears in the QUERYNO column in the PLAN_TABLE. *queryno* is an input parameter of type INTEGER.

sql-statement

Specifies the SQL statement on which EXPLAIN is to be executed. *sql-statement* is an input parameter of type CLOB(2M).

parse

Specifies whether DSNAEXP adds a qualifier for unqualified table or view

names in the input SQL statement. Valid values are 'Y' and 'N'. If the value of
parse is 'Y', *qualifier* must contain a valid SQL qualifier name.

If *sql-statement* is insert-within-select and common table expressions, you need
to disable the parsing functionality, and add the qualifier manually.

parse is an input parameter of type CHAR(1).

qualifier
Specifies the qualifier that DSNAEXP adds to unqualified table or view names
in the input SQL statement. If the value of *parse* is 'N', *qualifier* is ignored.

If the statement on which EXPLAIN is run contains an INSERT within a
SELECT or a common table expression, *parse* must be 'N', and table and view
qualifiers must be explicitly specified.

qualifier is an input parameter of type CHAR(8).

sqlcode
Contains the SQLCODE from execution of the EXPLAIN statement. *sqlcode* is
an output parameter of type INTEGER.

sqlstate
Contains the SQLSTATE from execution of the EXPLAIN statement. *sqlstate* is
an output parameter of type CHAR(5).

error-message
Contains information about DSNAEXP execution. If the SQLCODE from
execution of the EXPLAIN statement is not 0, *error-message* contains the error
message for the SQLCODE. *error-message* is an output parameter of type
VARCHAR(960).

Example of DSNAEXP invocation

The following C example shows how to call DSNAEXP to execute an EXPLAIN
statement.

```

# EXEC SQL BEGIN DECLARE SECTION;
# char      hvsqlid[9];          /* Qualifier for PLAN_TABLE */
# long int   hvqueryno;          /* QUERYNO to give the SQL */
# struct {
#     short int hvsql_stmt_len;   /* Input SQL statement length */
#     char      hvsql_stmt_txt SQL TYPE IS CLOB 2M;
#     /* SQL statement text */
#     } hvsql_stmt ;             /* SQL statement to EXPLAIN */
# char      hvparse[1];          /* Qualify object names */
# /* indicator */
# char      hvqualifier[9];       /* Qualifier for unqualified */
# /* objects */
# long int   hvsqlcode;          /* SQLCODE from CALL DSNAEXP */
# char      hvsqlstate[6];       /* SQLSTATE from CALL DSNAEXP */
# struct {
#     short int hvmsg_len;        /* Error message length */
#     char      hvmsg_text[961];  /* Error message text */
#     } hvmsg;                   /* Error message from failed */
# /* CALL DSNAEXP */
# EXEC SQL END DECLARE SECTION;
# short int i;
# :
# :
# /* Set the input parameters */
# strcpy(hvsqlid,"ADMFO01 ");
# hvqueryno=320;
# strcpy(hvsql_stmt.hvsql_stmt_txt,"SELECT CURRENT DATE FROM SYSDDUMMY1");
# hvsql_stmt.hvsql_stmt_len=strlen(hvsql_stmt.hvsql_stmt_txt);
# hvparse[0]='Y';
# strcpy(hvqualifier,"SYSIBM");

```

```

#          /* Initialize the output parameters */
#          hvsqlcode=0;
#          for (i = 0; i < 5; i++) hvsqlstate[i] = '0';
#          hvsqlstate[5]='\0';
#          hvmsg.hvmsg_len=0;
#          for (i = 0; i < 960; i++) hvmsg.hvmsg_text[i] = ' ';
#          hvmsg.hvmsg_text[960] = '\0';
#          /* Call DSNAXP to do EXPLAIN and put output in ADMF001.PLAN_TABLE */
#          EXEC SQL
#              CALL DSN8.DSNAEXP(:hvsqlid,
#                               :hvqueryno,
#                               :hvsql_stmt,
#                               :hvparse,
#                               :hvqualifier,
#                               :hvsqlcode,
#                               :hvsqlstate,
#                               :hvmsg);
#

```

DSNAEXP output

```

#          If DSNAEXP executes successfully, sqlid.PLAN_TABLE contains the EXPLAIN
#          output. A user with SELECT authority on sqlid.PLAN_TABLE can obtain the results
#          of the EXPLAIN that was executed by DSNAEXP by executing this query:
#          SELECT * FROM sqlid.PLAN_TABLE WHERE QUERYNO='queryno';

#          If DSNAEXP does not execute successfully, sqlcode, sqlstate, and error-message
#          contain error information.

```

Deprecated: Store an XML document from an MQ message queue in DB2 tables (DXXMQINSERT)

```

#          Restriction: DXXMQINSERT has been deprecated.

```

The DXXMQINSERT stored procedure returns a message that contains an XML document from an MQ message queue, decomposes the document, and stores the data in DB2 tables that are specified by an enabled XML collection.

Use DXXMQINSERT for an XML document with a length of up to 3KB.

There are two versions of DXXMQINSERT:

- A single-phase commit version, with schema name DMQXML1C.
- A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQINSERT

DXXMQINSERT runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

DXXMQINSERT requires that WebSphere MQ and XML Extender are installed.

See *DB2 Installation Guide* for installation instructions.

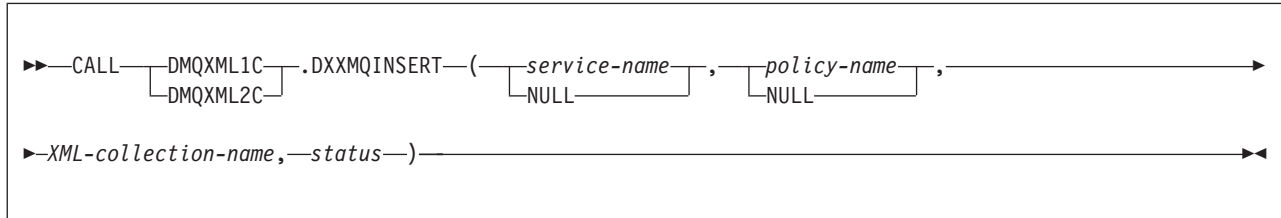
Authorization required for DXXMQINSERT

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 DXXMQINSERT
- Ownership of the stored procedure
- SYSADM authority

DXXMQINSERT syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DXXMQINSERT. Because the linkage convention for DXXMQINSERT 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.



DXXMQINSERT option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination from which the message is to be retrieved. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the message. The service policy is defined in the DSNAMT repository file. If *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.

XML-collection-name

Specifies the name of the XML collection that specifies the DB2 tables into which the decomposed XML document is to be inserted. *XML-collection-name* is an input parameter of type VARCHAR(80).

status

Contains information that indicates whether DXXMQINSERT ran successfully. The format of *status* is *dxx-rc:sqlcode:mq-num-msgs*, where:

- *dxx-rc* is the return code from accessing XML Extender. *dxx-rc* values are defined in *dxxrc.h*.
- *sqlcode* is 0 if DXXMQINSERT ran successfully, or the SQLCODE from the most recent unsuccessful SQL statement if DXXMQINSERT ran unsuccessfully.
- *mq-num-msgs* is the number of messages that were successfully sent to the MQ message queue.

status is an output parameter of type CHAR(20).

Example of DXXMQINSERT invocation

The following example calls the single-phase commit version of DXXMQINSERT to retrieve an XML document from an MQ message queue and decompose it into

DB2 tables that are specified by enabled collection sales_ord. For a complete example of DXXMQINSERT invocation, see DSN8QXSI in DSN810.SDSNSAMP.

```
#include "dxx.h"
#include "dxxrc.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char      serviceName[48];    /* WebSphere MQ service name */
char      policyName[48];     /* WebSphere MQ policy name */
char      collectionName[30]; /* XML collection name */
char      status[20];         /* Status of DXXMQINSERT call */
/* DXXMQINSERT is GENERAL WITH NULLS, so parameters need indicators */
short     serviceName_ind;    /* Indicator var for serviceName */
short     policyName_ind;     /* Indicator var for policyName */
short     collectionName_ind; /* Indicator var for collectionName */
short     status_ind;         /* Indicator var for status */
EXEC SQL END DECLARE SECTION;
/* Initialize status fields */
int dxx_rc=0;
int dxx_sql=0;
int dxx_mq=0;
/* Get the service name and policy name for the MQ message queue */
strcpy(serviceName,"DB2.DEFAULT.SERVICE");
strcpy(policyName,"DB2.DEFAULT.POLICY");
/* Get the XML collection name */
strcpy(collectionName,"sales_ord");
/* Initialize the output variable */
status[0] = '\0';
/* Set the indicators to 0 for the parameters that have non-null */
/* values */
collectionName_ind = 0;
serviceName_ind = 0;
policyName_ind = 0;
/* Initialize the indicator for the output parameter */
status_ind = -1;
/* Call the store procedure */
EXEC SQL CALL DMQXML1C.DXXMQINSERT(:serviceName:serviceName_ind,
:policyName:policyName_ind,
:dadFileName:dadFileName_ind,
:overrideType:ovtype_ind,
:override:ov_ind,
:max_row:maxrow_ind,
:num_row:numrow_ind,
:status:status_ind);
printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
/* Get the status fields from the status parameter and print them */
sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);
```

DXXMQINSERT output

If DXXMQINSERT executes successfully, the *mq-num-msgs* field of the *status* parameter is set to 1, to indicate that a message was retrieved from the MQ message queue and inserted into DB2 tables. If DXXMQINSERT does not execute successfully, the contents of the *status* parameter indicate the problem.

Deprecated: Store an XML document from an MQ message queue in DB2 tables (DXXMQSHRED)

Restriction: DXXMQSHRED has been deprecated.

The DXXMQSHRED stored procedure returns a message that contains an XML document from an MQ message queue, decomposes the document, and stores the data in DB2 tables that are specified in a document access definition (DAD) file. DXXMQSHRED does not require an enabled XML collection.

Use DXXMQSHRED for an XML document with a length of up to 3KB.

There are two versions of DXXMQSHRED:

- # • A single-phase commit version, with schema name DMQXML1C.
- # • A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQSHRED

DXXMQSHRED runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

DXXMQSHRED requires that WebSphere MQ and XML Extender are installed.

See *DB2 Installation Guide* for installation instructions.

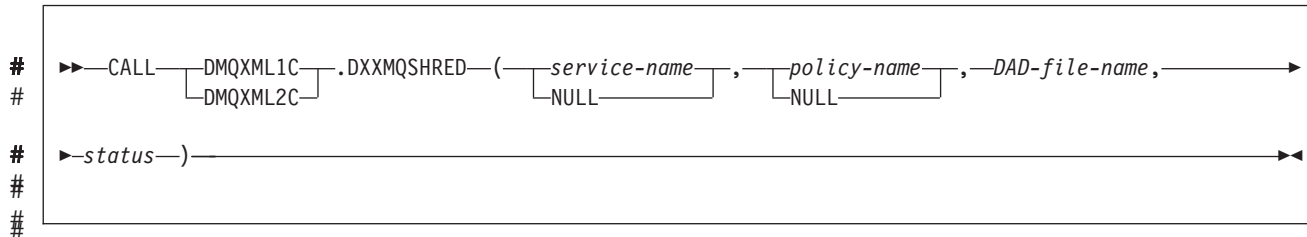
Authorization required for DXXMQSHRED

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 DXXMQSHRED
- # • Ownership of the stored procedure
- # • SYSADM authority

DXXMQSHRED syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DXXMQSHRED. Because the linkage convention for DXXMQSHRED 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.



DXXMQSHRED option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination from which the message is to be retrieved. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the message. The service policy is defined in the DSNAMT repository file. If

```

#           policy-name is not listed in the DSNAMT repository file, or policy-name is not
#           specified, DB2.DEFAULT.POLICY is used.

#           policy-name is an input parameter of type VARCHAR(48). policy-name cannot be
#           blank, a null string, or have trailing blanks.

# DAD-file-name
#           Specifies the name of the (DAD) file that maps the XML document to DB2
#           tables. DAD-file-name must be specified, and must be the name of a valid DAD
#           file that exists on the system on which DXXMQSHRED runs. DAD-file-name is
#           an input parameter of type VARCHAR(80).

# status
#           Contains information that indicates whether DXXMQSHRED ran successfully.
#           The format of status is dxx-rc:sqlcode:mq-num-msgs, where:
#           • dxx-rc is the return code from accessing XML Extender. dxx-rc values are
#             defined in dxxrc.h.
#           • sqlcode is 0 if DXXMQSHRED ran successfully, or the SQLCODE from the
#             most recent unsuccessful SQL statement if DXXMQSHRED ran
#             unsuccessfully.
#           • mq-num-msgs is the number of messages that were successfully sent to the
#             MQ message queue.

#           status is an output parameter of type CHAR(20).

```

Example of DXXMQSHRED invocation

```

#           The following example calls the single-phase commit version of DXXMQSHRED to
#           retrieve an XML document from an MQ message queue and decompose it into
#           DB2 tables that are specified by DAD file /tmp/neworder2.dad. For a complete
#           example of DXXMQSHRED invocation, see DSN8QXSI in DSN810.SDSNSAMP.

#           #include "dxx.h"
#           #include "dxxrc.h"
#           EXEC SQL INCLUDE SQLCA;
#           EXEC SQL BEGIN DECLARE SECTION;
#           char      serviceName[48];      /* WebSphere MQ service name */
#           char      policyName[48];      /* WebSphere MQ policy name */
#           char      dadFileName[30];      /* DAD file name */
#           char      status[20];          /* Status of DXXMQSHRED call */
#           /* DXXMQSHRED is GENERAL WITH NULLS, so parameters need indicators */
#           short     serviceName_ind;      /* Indicator var for serviceName */
#           short     policyName_ind;      /* Indicator var for policyName */
#           short     dadFileName_ind;      /* Indicator var for dadFileName */
#           short     status_ind;          /* Indicator var for status */
#           EXEC SQL END DECLARE SECTION;
#           /* Initialize status fields */
#           int dxx_rc=0;
#           int dxx_sql=0;
#           int dxx_mq=0;
#           /* Get the service name and policy name for the MQ message queue */
#           strcpy(serviceName,"DB2.DEFAULT.SERVICE");
#           strcpy(policyName,"DB2.DEFAULT.POLICY");
#           /* Get the DAD file name */
#           strcpy(dadFileName,"/tmp/neworder2.dad");
#           /* Initialize the output variable */
#           status[0] = '\0';
#           /* Set the indicators to 0 for the parameters that have non-null */
#           /* values */
#           collectionName_ind = 0;
#           serviceName_ind = 0;
#           policyName_ind = 0;
#           /* Initialize the indicator for the output parameter */
#           status_ind = -1;

```



```
#          /* Call the store procedure */
#          EXEC SQL CALL DMQXML1C.DXXMQSHRED(:serviceName:serviceName_ind,
#              :policyName:policyName_ind,
#              :dadFileName:dadFileName_ind,
#              :status:status_ind);
#          printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
#          /* Get the status fields from the status parameter and print them */
#          sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
#          printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);
```

DXXMQSHRED output

```
#          If DXXMQSHRED executes successfully, the mq-num-msgs field of the status
#          parameter is set to 1, to indicate that a message was retrieved from the MQ
#          message queue and inserted into DB2 tables. If DXXMQSHRED does not execute
#          successfully, the contents of the status parameter indicate the problem.
```

Depreciated: Store a large XML document from an MQ message queue in DB2 tables (DXXMQINSERTCLOB)

```
#          Restriction: DXXMQINSERTCLOB has been deprecated.

#          The DXXMQINSERTCLOB stored procedure returns a message that contains an
#          XML document from an MQ message queue, decomposes the document, and
#          stores the data in DB2 tables that are specified by an enabled XML collection.

#          Use DXXMQINSERTCLOB for an XML document with a length of up to 1MB.

#          There are two versions of DXXMQINSERTCLOB:
#          • A single-phase commit version, with schema name DMQXML1C.
#          • A two-phase commit version, with schema name DMQXML2C.
```

Environment for DXXMQINSERTCLOB

```
#          DXXMQINSERTCLOB runs in a WLM-established stored procedure address space
#          and uses the Resource Recovery Services attachment facility to connect to DB2.

#          DXXMQINSERTCLOB requires that WebSphere MQ and XML Extender are
#          installed.

#          See DB2 Installation Guide for installation instructions.
```

Authorization required for DXXMQINSERTCLOB

```
#          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 DXXMQINSERTCLOB
#          • Ownership of the stored procedure
#          • SYSADM authority
```

DXXMQINSERTCLOB syntax diagram

```
#          The following syntax diagram shows the SQL CALL statement for invoking
#          DXXMQINSERTCLOB. Because the linkage convention for DXXMQINSERTCLOB
#          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.
```



```

# ──CALL──┬──DMQXML1C──┬──.DXXMQINSERTCLOB──(──┬──service-name──┬──,──┬──policy-name──┬──,──┬──
# ──┬──DMQXML2C──┴──┬──NULL──┴──┬──NULL──┴──┬──
# ──XML-collection-name,──status──)──┬──
#
#
#

```

DXXMQINSERTCLOB option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination from which the message is to be retrieved. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the message. The service policy is defined in the DSNAMT repository file. If *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.

XML-collection-name

Specifies the name of the XML collection that specifies the DB2 tables into which the decomposed XML document is to be inserted. *XML-collection-name* must be specified, and must be the name of an enabled collection that exists on the system on which DXXMQINSERTCLOB runs. *XML-collection-name* is an input parameter of type VARCHAR(80).

status

Contains information that indicates whether DXXMQINSERTCLOB ran successfully. The format of *status* is *dxx-rc:sqlcode:mq-num-msgs*, where:

- *dxx-rc* is the return code from accessing XML Extender. *dxx-rc* values are defined in dxxrc.h.
- *sqlcode* is 0 if DXXMQINSERTCLOB ran successfully, or the SQLCODE from the most recent unsuccessful SQL statement if DXXMQINSERTCLOB ran unsuccessfully.
- *mq-num-msgs* is the number of messages that were successfully sent to the MQ message queue.

status is an output parameter of type CHAR(20).

Example of DXXMQINSERTCLOB invocation

The following example calls the single-phase commit version of DXXMQINSERTCLOB to retrieve all XML documents from an MQ message queue and decompose them into DB2 tables that are specified by enabled collection sales_ord. For a complete example of DXXMQINSERTCLOB invocation, see DSN8QXSI in DSN810.SDSNSAMP.

```

#include "dxx.h"
#include "dxxrc.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char      serviceName[48];    /* WebSphere MQ service name */

```

```

#          char      policyName[48];          /* WebSphere MQ policy name          */
#          char      collectionName[30];      /* XML collection name              */
#          char      status[20];              /* Status of DXXMQINSERTCLOB call   */
#          /* DXXMQINSERTCLOB is GENERAL WITH NULLS, so parameters need indicators */
#          short     serviceName_ind;         /* Indicator var for serviceName     */
#          short     policyName_ind;         /* Indicator var for policyName      */
#          short     collectionName_ind;      /* Indicator var for collectionName  */
#          short     status_ind;              /* Indicator var for status          */
#          EXEC SQL END DECLARE SECTION;
#          /* Initialize status fields */
#          int dxx_rc=0;
#          int dxx_sql=0;
#          int dxx_mq=0;
#          /* Get the service name and policy name for the MQ message queue */
#          strcpy(serviceName,"DB2.DEFAULT.SERVICE");
#          strcpy(policyName,"DB2.DEFAULT.POLICY");
#          /* Get the XML collection name */
#          strcpy(collectionName,"sales_ord");
#          /* Initialize the output variable */
#          status[0] = '\0';
#          /* Set the indicators to 0 for the parameters that have non-null */
#          /* values */
#          collectionName_ind = 0;
#          serviceName_ind = 0;
#          policyName_ind = 0;
#          /* Initialize the indicator for the output parameter */
#          status_ind = -1;
#          /* Call the store procedure */
#          EXEC SQL CALL DMQXML1C.DXXMQINSERTCLOB(:serviceName:serviceName_ind,
#          :policyName:policyName_ind,
#          :collectionName:collectionName_ind,
#          :status:status_ind);
#          printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
#          /* Get the status fields from the status parameter and print them */
#          sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
#          printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);

```

DXXMQINSERTCLOB output

If DXXMQINSERTCLOB executes successfully, the *mq-num-msgs* field of the *status* parameter is set to 1, to indicate that a message was retrieved from the MQ message queue and inserted into DB2 tables. If DXXMQINSERTCLOB does not execute successfully, the contents of the *status* parameter indicate the problem.

Deprecated: Store a large XML document from an MQ message queue in DB2 tables (DXXMQSHREDCLOB)

Restriction: DXXMQSHREDCLOB has been deprecated.

The DXXMQSHREDCLOB stored procedure returns a message that contains an XML document from an MQ message queue, decomposes the document, and stores the data in DB2 tables that are specified in a document access definition (DAD) file. DXXMQSHREDCLOB does not require an enabled XML collection.

Use DXXMQSHREDCLOB for an XML document with a length of up to 1MB.

There are two versions of DXXMQSHREDCLOB:

- A single-phase commit version, with schema name DMQXML1C.
- A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQSHREDCLOB

DXXMQSHREDCLOB runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

DXXMQSHREDCLOB requires that WebSphere MQ and XML Extender are installed.

See *DB2 Installation Guide* for installation instructions.

Authorization required for DXXMQSHREDCLOB

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 DXXMQSHREDCLOB
- Ownership of the stored procedure
- SYSADM authority

DXXMQSHREDCLOB syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DXXMQSHREDCLOB. Because the linkage convention for DXXMQSHREDCLOB 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.

```
CALL (DMQXML1C | DMQXML2C) .DXXMQSHREDCLOB ( (service-name | NULL), (policy-name | NULL), DAD-file-name,
status )
```

DXXMQSHREDCLOB option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination from which the message is to be retrieved. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the message. The service policy is defined in the DSNAMT repository file. If *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.

DAD-file-name

Specifies the name of the (DAD) file that maps the XML document to DB2 tables. *DAD-file-name* is an input parameter of type VARCHAR(80).

```
#          status
#          Contains information that indicates whether DXXMQSHREDCLOB ran
#          successfully. The format of status is dxx-rc:sqlcode:mq-num-msgs, where:
#          • dxx-rc is the return code from accessing XML Extender. dxx-rc values are
#            defined in dxxrc.h.
#          • sqlcode is 0 if DXXMQSHREDCLOB ran successfully, or the SQLCODE from
#            the most recent unsuccessful SQL statement if DXXMQSHREDCLOB ran
#            unsuccessfully.
#          • mq-num-msgs is the number of messages that were successfully sent to the
#            MQ message queue.
#          status is an output parameter of type CHAR(20).
```

```
# The following example calls the single-phase commit version of
# DXXMQSHREDCLOB to retrieve an XML document from an MQ message queue
# and decompose it into DB2 tables that are specified by DAD file
# /tmp/neworder2.dad. For a complete example of DXXMQSHREDCLOB
# invocation, see DSN8OXSI in DSN810.SDSNSAMP.
```

```

# #include "dxx.h"
# #include "dxxrc.h"
# EXEC SQL INCLUDE SQLCA;
# EXEC SQL BEGIN DECLARE SECTION;
# char      serviceName[48];      /* WebSphere MQ service name */
# char      policyName[48];      /* WebSphere MQ policy name */
# char      dadFileName[30];      /* DAD file name */
# char      status[20];          /* Status of DXXMQSHREDCLOB call */
# /* DXXMQSHREDCLOB is GENERAL WITH NULLS, so parameters need indicators */
# short     serviceName_ind;      /* Indicator var for serviceName */
# short     policyName_ind;      /* Indicator var for policyName */
# short     dadFileName_ind;      /* Indicator var for dadFileName */
# short     status_ind;          /* Indicator var for status */
# EXEC SQL END DECLARE SECTION;
# /* Initialize status fields */
# int dxx_rc=0;
# int dxx_sql=0;
# int dxx_mq=0;
# /* Get the service name and policy name for the MQ message queue */
# strcpy(serviceName,"DB2.DEFAULT.SERVICE");
# strcpy(policyName,"DB2.DEFAULT.POLICY");
# /* Get the DAD file name */
# strcpy(dadFileName,"/tmp/neworder2.dad");
# /* Initialize the output variable */
# status[0] = '\0';
# /* Set the indicators to 0 for the parameters that have non-null */
# /* values */
# collectionName_ind = 0;
# serviceName_ind = 0;
# policyName_ind = 0;
# /* Initialize the indicator for the output parameter */
# status_ind = -1;
# /* Call the store procedure */
# EXEC SQL CALL DMQXML1C.DXXMQSHREDCLOB(:serviceName:serviceName_ind,
# :policyName:policyName_ind,
# :dadFileName:dadFileName_ind,
# :status:status_ind);
# printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
# /* Get the status fields from the status parameter and print them */
# sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
# printf("Status fields: dxx rc=%d dxx sql=%d dxx mq=%d\n",dxx_rc,dxx_sql,dxx_mq);

```

DXXMQSHREDCLOB output

If DXXMQSHREDCLOB executes successfully, the *mq-num-msgs* field of the *status* parameter is set to 1, to indicate that a message was retrieved from the MQ message queue and inserted into DB2 tables. If DXXMQSHREDCLOB does not execute successfully, the contents of the *status* parameter indicate the problem.

Deprecated: Store XML documents from an MQ message queue in DB2 tables (DXXMQINSERTALL)

Restriction: DXXMQINSERTALL has been deprecated.

The DXXMQINSERTALL stored procedure returns messages that contains XML documents from an MQ message queue, decomposes the documents, and stores the data in DB2 tables that are specified by an enabled XML collection.

Use DXXMQINSERTALL for XML documents with a length of up to 3KB.

There are two versions of DXXMQINSERTALL:

- A single-phase commit version, with schema name DMQXML1C.
- A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQINSERTALL

DXXMQINSERTALL runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

DXXMQINSERTALL requires that WebSphere MQ and XML Extender are installed.

See *DB2 Installation Guide* for installation instructions.

Authorization required for DXXMQINSERTALL

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 DXXMQINSERTALL
- Ownership of the stored procedure
- SYSADM authority

DXXMQINSERTALL syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DXXMQINSERTALL. Because the linkage convention for DXXMQINSERTALL 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.

```
CALL (DMQXML1C | DMQXML2C).DXXMQINSERTALL ( (service-name | NULL), (policy-name | NULL),  
XML-collection-name, status )
```

DXXMQINSERTALL option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination from which the messages are to be retrieved. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the messages. The service policy is defined in the DSNAMT repository file. If *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.

XML-collection-name

Specifies the name of the XML collection that specifies the DB2 tables into which the decomposed XML documents are to be inserted. *XML-collection-name* must be specified, and must be the name of a valid XML collection that exists on the system on which DXXMQINSERTALL runs. *XML-collection-name* is an input parameter of type VARCHAR(80).

status

Contains information that indicates whether DXXMQINSERTALL ran successfully. The format of *status* is *dxx-rc:sqlcode:mq-num-msgs*, where:

- *dxx-rc* is the return code from accessing XML Extender. *dxx-rc* values are defined in dxxrc.h.
- *sqlcode* is 0 if DXXMQINSERTALL ran successfully, or the SQLCODE from the most recent unsuccessful SQL statement if DXXMQINSERTALL ran unsuccessfully.
- *mq-num-msgs* is the number of messages that were successfully sent to the MQ message queue.

status is an output parameter of type CHAR(20).

Example of DXXMQINSERTALL invocation

The following example calls the single-phase commit version of DXXMQINSERTALL to retrieve all XML documents from an MQ message queue and decompose them into DB2 tables that are specified by enabled collection sales_ord. For a complete example of DXXMQINSERTALL invocation, see DSN8QXSI in DSN810.SDSNSAMP.

```
#include "dxx.h"
#include "dxxrc.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char      serviceName[48];    /* WebSphere MQ service name */
char      policyName[48];    /* WebSphere MQ policy name */
char      collectionName[30]; /* XML collection name */
char      status[20];        /* Status of DXXMQINSERTALL call */
/* DXXMQINSERTALL is GENERAL WITH NULLS, so parameters need indicators */
short     serviceName_ind;    /* Indicator var for serviceName */
short     policyName_ind;     /* Indicator var for policyName */
short     collectionName_ind; /* Indicator var for collectionName */
short     status_ind;         /* Indicator var for status */
EXEC SQL END DECLARE SECTION;
```



```

#          /* Initialize status fields                                     */
#          int dxx_rc=0;
#          int dxx_sql=0;
#          int dxx_mq=0;
#          /* Get the service name and policy name for the MQ message queue */
#          strcpy(serviceName,"DB2.DEFAULT.SERVICE");
#          strcpy(policyName,"DB2.DEFAULT.POLICY");
#          /* Get the XML collection name                                     */
#          strcpy(collectionName,"sales_ord");
#          /* Initialize the output variable                               */
#          status[0] = '\0';
#          /* Set the indicators to 0 for the parameters that have non-null */
#          /* values                                                         */
#          collectionName_ind = 0;
#          serviceName_ind = 0;
#          policyName_ind = 0;
#          /* Initialize the indicators for the output parameter             */
#          status_ind = -1;
#          /* Call the store procedure */
#          EXEC SQL CALL DMQXML1C.DXXMQINSERTALL(:serviceName:serviceName_ind,
#          :policyName:policyName_ind,
#          :collectionName:collectionName_ind,
#          :status:status_ind);
#          printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
#          /* Get the status fields from the status parameter and print them */
#          sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
#          printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);

```

DXXMQINSERTALL output

```

#          If DXXMQINSERTALL executes successfully, the mq-num-msgs field of the status
#          parameter is set to the number of messages that were retrieved from the MQ
#          message queue and decomposed. If DXXMQINSERTALL does not execute
#          successfully, the contents of the status parameter indicate the problem.

```

Depreciated: Store XML documents from an MQ message queue in DB2 tables (DXXMQSHREDALL)

```

#          Restriction: DXXMQSHREDALL has been deprecated.

#          The DXXMQSHREDALL stored procedure returns messages that contain XML
#          documents from an MQ message queue, decomposes the documents, and stores
#          the data in DB2 tables that are specified in a document access definition (DAD)
#          file. DXXMQSHREDALL does not require an enabled XML collection.

#          Use DXXMQSHREDALL for XML documents with a length of up to 3KB.

#          There are two versions of DXXMQSHREDALL:
#          • A single-phase commit version, with schema name DMQXML1C.
#          • A two-phase commit version, with schema name DMQXML2C.

```

Environment for DXXMQSHREDALL

```

#          DXXMQSHREDALL runs in a WLM-established stored procedure address space
#          and uses the Resource Recovery Services attachment facility to connect to DB2.

#          DXXMQSHREDALL requires that WebSphere MQ and XML Extender are installed.

#          See DB2 Installation Guide for installation instructions.

```

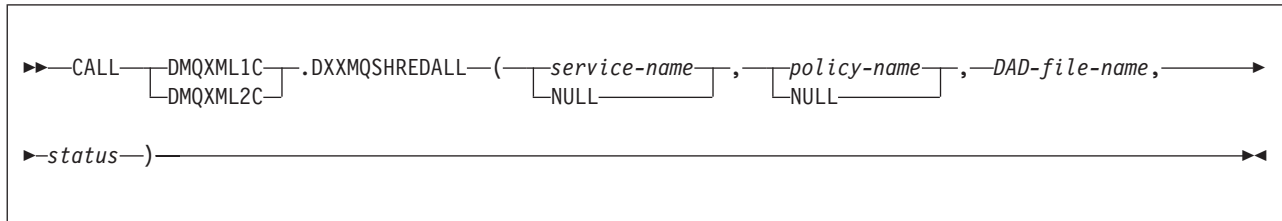
Authorization required for DXXMQSHREDALL

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 DXXMQSHREDALL
- Ownership of the stored procedure
- SYSADM authority

DXXMQSHREDALL syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DXXMQSHREDALL. Because the linkage convention for DXXMQSHREDALL 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.



DXXMQSHREDALL option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination from which messages are to be retrieved. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the messages. The service policy is defined in the DSNAMT repository file. If *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.

DAD-file-name

Specifies the name of the (DAD) file that maps the XML document to DB2 tables. *DAD-file-name* must be specified, and must be the name of a valid DAD file that exists on the system on which DXXMQSHREDALL runs. *DAD-file-name* is an input parameter of type VARCHAR(80).

status

Contains information that indicates whether DXXMQSHREDALL ran successfully. The format of *status* is *dxx-rc:sqlcode:mq-num-msgs*, where:

- *dxx-rc* is the return code from accessing XML Extender. *dxx-rc* values are defined in dxxrc.h.
- *sqlcode* is 0 if DXXMQSHREDALL ran successfully, or the SQLCODE from the most recent unsuccessful SQL statement if DXXMQSHREDALL ran unsuccessfully.


```
#
#      • mq-num-msgs is the number of messages that were successfully sent to the
#        MQ message queue.
#
#      status is an output parameter of type CHAR(20).
```

Example of DXXMQSHREDALL invocation

```
#
#      The following example calls the single-phase commit version of
#      DXXMQSHREDALL to retrieve XML documents from an MQ message queue and
#      decompose them into DB2 tables that are specified by DAD file
#      /tmp/neworder2.dad. For a complete example of DXXMQSHREDALL invocation,
#      see DSN8QXSI in DSN810.SDSNSAMP.
```

```
#      #include "dxx.h"
#      #include "dxxrc.h"
#      EXEC SQL INCLUDE SQLCA;
#      EXEC SQL BEGIN DECLARE SECTION;
#      char      serviceName[48];      /* WebSphere MQ service name      */
#      char      policyName[48];      /* WebSphere MQ policy name      */
#      char      dadFileName[30];      /* DAD file name                  */
#      char      status[20];          /* Status of DXXMQSHREDALL call   */
#      /* DXXMQSHREDALL is GENERAL WITH NULLS, so parameters need indicators */
#      short     serviceName_ind;      /* Indicator var for serviceName   */
#      short     policyName_ind;      /* Indicator var for policyName    */
#      short     dadFileName_ind;      /* Indicator var for dadFileName   */
#      short     status_ind;          /* Indicator var for status        */
#      EXEC SQL END DECLARE SECTION;
#      /* Initialize status fields */
#      int dxx_rc=0;
#      int dxx_sql=0;
#      int dxx_mq=0;
#      /* Get the service name and policy name for the MQ message queue */
#      strcpy(serviceName,"DB2.DEFAULT.SERVICE");
#      strcpy(policyName,"DB2.DEFAULT.POLICY");
#      /* Get the DAD file name */
#      strcpy(dadFileName,"/tmp/neworder2.dad");
#      /* Initialize the output variable */
#      status[0] = '\0';
#      /* Set the indicators to 0 for the parameters that have non-null */
#      /* values */
#      collectionName_ind = 0;
#      serviceName_ind = 0;
#      policyName_ind = 0;
#      /* Initialize the indicator for the output parameter */
#      status_ind = -1;
#      /* Call the store procedure */
#      EXEC SQL CALL DMQXML1C.DXXMQSHREDALL(:serviceName:serviceName_ind,
#      :policyName:policyName_ind,
#      :dadFileName:dadFileName_ind,
#      :status:status_ind);
#      printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
#      /* Get the status fields from the status parameter and print them */
#      sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
#      printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);
```

DXXMQSHREDALL output

```
#
#      If DXXMQSHREDALL executes successfully, the mq-num-msgs field of the status
#      parameter is set to the number of messages that were retrieved from the MQ
#      message queue and inserted into DB2 tables. If DXXMQSHREDALL does not
#      execute successfully, the contents of the status parameter indicate the problem.
```

Depreciated: Store large XML documents from an MQ message queue # in DB2 tables (DXXMQSHREDALLCLOB)

Restriction: DXXMQSHREDALLCLOB has been deprecated.

The DXXMQSHREDALLCLOB stored procedure returns messages that contain
XML documents from an MQ message queue, decomposes the documents, and
stores the data in DB2 tables that are specified in a document access definition
(DAD) file. DXXMQSHREDALLCLOB does not require an enabled XML collection.

Use DXXMQSHREDALLCLOB for XML documents with a length of up to 1MB.

There are two versions of DXXMQSHREDALLCLOB:

- # • A single-phase commit version, with schema name DMQXML1C.
- # • A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQSHREDALLCLOB

DXXMQSHREDALLCLOB runs in a WLM-established stored procedure address
space and uses the Resource Recovery Services attachment facility to connect to
DB2.

DXXMQSHREDALLCLOB requires that WebSphere MQ and XML Extender are
installed.

See *DB2 Installation Guide* for installation instructions.

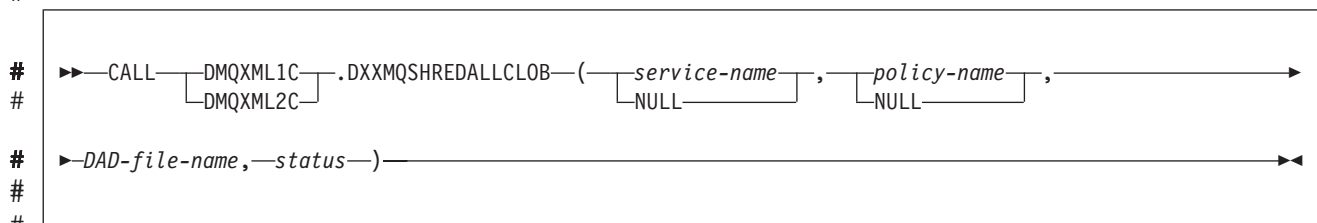
Authorization required for DXXMQSHREDALLCLOB

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 DXXMQSHREDALLCLOB
- # • Ownership of the stored procedure
- # • SYSADM authority

DXXMQSHREDALLCLOB syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking
DXXMQSHREDALLCLOB. Because the linkage convention for
DXXMQSHREDALLCLOB 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.



DXXMQSHREDALLCLOB option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination from

which messages are to be retrieved. The service point is defined in the
DSNAMT repository file. If *service-name* is not listed in the DSNAMT
repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot
be blank, a null string, or have trailing blanks.

policy-name
Specifies the WebSphere MQ AMI service policy that is used to handle the
messages. The service policy is defined in the DSNAMT repository file. If
policy-name is not listed in the DSNAMT repository file, or *policy-name* is not
specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be
blank, a null string, or have trailing blanks.

DAD-file-name
Specifies the name of the (DAD) file that maps the XML document to DB2
tables. *DAD-file-name* must be specified, and must be the name of a valid DAD
file that exists on the system on which DXXMQSHREDALLCLOB runs.
DAD-file-name is an input parameter of type VARCHAR(80).

status
Contains information that indicates whether DXXMQSHREDALLCLOB ran
successfully. The format of *status* is *dxx-rc:sqlcode:mq-num-msgs*, where:

- # • *dxx-rc* is the return code from accessing XML Extender. *dxx-rc* values are
defined in dxxrc.h.
- # • *sqlcode* is 0 if DXXMQSHREDALLCLOB ran successfully, or the SQLCODE
from the most recent unsuccessful SQL statement if
DXXMQSHREDALLCLOB ran unsuccessfully.
- # • *mq-num-msgs* is the number of messages that were successfully sent to the
MQ message queue.

status is an output parameter of type CHAR(20).

Example of DXXMQSHREDALLCLOB invocation

The following example calls the single-phase commit version of
DXXMQSHREDALLCLOB to retrieve XML documents from an MQ message queue
and decompose them into DB2 tables that are specified by DAD file
/tmp/neworder2.dad. For a complete example of DXXMQSHREDALLCLOB
invocation, see DSN8QXSI in DSN810.SDSNSAMP.

```
#include "dxx.h"
#include "dxxrc.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char      serviceName[48];    /* WebSphere MQ service name */
char      policyName[48];    /* WebSphere MQ policy name */
char      dadFileName[30];   /* DAD file name */
char      status[20];        /* Status of DXXMQSHREDALLCLOB call */
/* DXXMQSHREDALLCLOB is GENERAL WITH NULLS, so parameters need indicators */
short     serviceName_ind;   /* Indicator var for serviceName */
short     policyName_ind;    /* Indicator var for policyName */
short     dadFileName_ind;   /* Indicator var for dadFileName */
short     status_ind;        /* Indicator var for status */
EXEC SQL END DECLARE SECTION;
/* Initialize status fields */
int dxx_rc=0;
int dxx_sql=0;
int dxx_mq=0;
/* Get the service name and policy name for the MQ message queue */
strcpy(serviceName,"DB2.DEFAULT.SERVICE");
```

```

# strcpy(policyName,"DB2.DEFAULT.POLICY");
# /* Get the DAD file name */
# strcpy(dadFileName,"/tmp/neworder2.dad");
# /* Initialize the output variable */
# status[0] = '\0';
# /* Set the indicators to 0 for the parameters that have non-null */
# /* values */
# collectionName_ind = 0;
# serviceName_ind = 0;
# policyName_ind = 0;
# /* Initialize the indicator for the output parameter */
# status_ind = -1;
# /* Call the store procedure */
# EXEC SQL CALL DMQXML1C.DXXMQSHREDALLCLOB(:serviceName:serviceName_ind,
# :policyName:policyName_ind,
# :dadFileName:dadFileName_ind,
# :status:status_ind);
# printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
# /* Get the status fields from the status parameter and print them */
# sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
# printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);

```

DXXMQSHREDALLCLOB output

If DXXMQSHREDALLCLOB executes successfully, the *mq-num-msgs* field of the *status* parameter is set to the number of messages that were retrieved from the MQ message queue and inserted into DB2 tables. If DXXMQSHREDALLCLOB does not execute successfully, the contents of the *status* parameter indicate the problem.

Depreciated: Store large XML documents from an MQ message queue in DB2 tables (DXXMQINSERTALLCLOB)

```

# Restriction: DXXMQINSERTALLCLOB has been deprecated.

# The DXXMQINSERTALLCLOB stored procedure returns messages that contains
# XML documents from an MQ message queue, decomposes the documents, and
# stores the data in DB2 tables that are specified by an enabled XML collection.

# Use DXXMQINSERTALLCLOB for XML documents with a length of up to 1MB.

# There are two versions of DXXMQINSERTALLCLOB:
# • A single-phase commit version, with schema name DMQXML1C.
# • A two-phase commit version, with schema name DMQXML2C.

```

Environment for DXXMQINSERTALLCLOB

```

# DXXMQINSERTALLCLOB runs in a WLM-established stored procedure address
# space and uses the Resource Recovery Services attachment facility to connect to
# DB2.

# DXXMQINSERTALLCLOB requires that WebSphere MQ and XML Extender are
# installed.

# See DB2 Installation Guide for installation instructions.

```

Authorization required for DXXMQINSERTALLCLOB

```

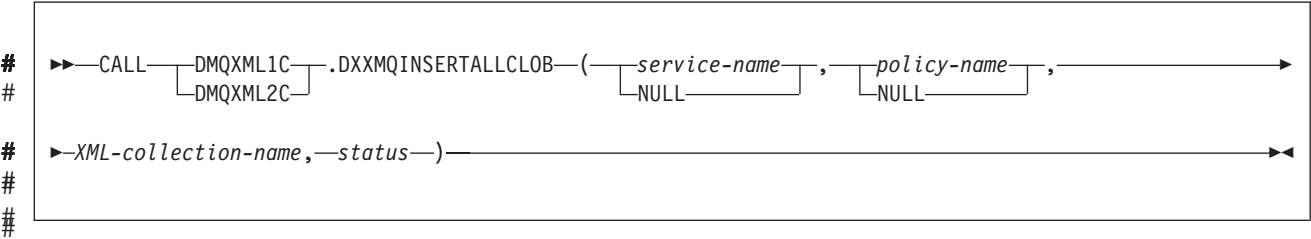
# To execute the CALL statement, the owner of the package or plan that contains the
# CALL statement must have one or more of the following privileges:
# • The EXECUTE privilege on stored procedure DXXMQINSERTALLCLOB

```

- Ownership of the stored procedure
- SYSADM authority

DXXMQINSERTALLCLOB syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DXXMQINSERTALLCLOB. Because the linkage convention for DXXMQINSERTALLCLOB 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.



DXXMQINSERTALLCLOB option descriptions

- service-name*
- Specifies the service point that is the logical WebSphere MQ destination from which the messages are to be retrieved. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.
- service-name* is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.
- policy-name*
- Specifies the WebSphere MQ AMI service policy that is used to handle the messages. The service policy is defined in the DSNAMT repository file. If *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.
- policy-name* is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.
- XML-collection-name*
- Specifies the name of the XML collection that specifies the DB2 tables into which the decomposed XML documents are to be inserted. *XML-collection-name* must be specified, and must be the name of a valid XML collection that exists on the system on which DXXMQINSERTALLCLOB runs. *XML-collection-name* is an input parameter of type VARCHAR(80).
- status*
- Contains information that indicates whether DXXMQINSERTALLCLOB ran successfully. The format of *status* is *dxx-rc:sqlcode:mq-num-msgs*, where:
- *dxx-rc* is the return code from accessing XML Extender. *dxx-rc* values are defined in dxxrc.h.
 - *sqlcode* is 0 if DXXMQINSERTALLCLOB ran successfully, or the SQLCODE from the most recent unsuccessful SQL statement if DXXMQINSERTALLCLOB ran unsuccessfully.
 - *mq-num-msgs* is the number of messages that were successfully sent to the MQ message queue.

status is an output parameter of type CHAR(20).

Example of DXXMQINSERTALLCLOB invocation

The following example calls the single-phase commit version of DXXMQINSERTALLCLOB to retrieve all XML documents from an MQ message queue and decompose them into DB2 tables that are specified by enabled collection *sales_ord*. For a complete example of DXXMQINSERTALLCLOB invocation, see DSN8QXSI in DSN810.SDSNSAMP.

```
#include "dxx.h"
#include "dxxrc.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char      serviceName[48]; /* WebSphere MQ service name */
char      policyName[48]; /* WebSphere MQ policy name */
char      collectionName[30]; /* XML collection name */
char      status[20]; /* Status of DXXMQINSERTALLCLOB call */
/* DXXMQINSERTALLCLOB is GENERAL WITH NULLS, so parameters need indicators */
short     serviceName_ind; /* Indicator var for serviceName */
short     policyName_ind; /* Indicator var for policyName */
short     collectionName_ind; /* Indicator var for collectionName */
short     status_ind; /* Indicator var for status */
EXEC SQL END DECLARE SECTION;
/* Initialize status fields */
int dxx_rc=0;
int dxx_sql=0;
int dxx_mq=0;
/* Get the service name and policy name for the MQ message queue */
strcpy(serviceName,"DB2.DEFAULT.SERVICE");
strcpy(policyName,"DB2.DEFAULT.POLICY");
/* Get the XML collection name */
strcpy(collectionName,"sales_ord");
/* Initialize the output variable */
status[0] = '\0';
/* Set the indicators to 0 for the parameters that have non-null */
/* values */
collectionName_ind = 0;
serviceName_ind = 0;
policyName_ind = 0;
/* Initialize the indicators for the output parameter */
status_ind = -1;
/* Call the store procedure */
EXEC SQL CALL DMQXML1C.DXXMQINSERTALLCLOB(:serviceName:serviceName_ind,
:policyName:policyName_ind,
:collectionName:collectionName_ind,
:status:status_ind);
printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
/* Get the status fields from the status parameter and print them */
sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);
```

DXXMQINSERTALLCLOB output

If DXXMQINSERTALLCLOB executes successfully, the *mq-num-msgs* field of the *status* parameter is set to the number of messages that were retrieved from the MQ message queue and decomposed. If DXXMQINSERTALLCLOB does not execute successfully, the contents of the *status* parameter indicate the problem.

Deprecated: Send XML documents to an MQ message queue (DXXMQGEN)

Restriction: DXXMQGEN has been deprecated.

policy-name is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.

DAD-file-name

Specifies the name of the (DAD) file that maps the XML documents to DB2 tables. *DAD-file-name* must be specified, and must be the name of a valid DAD file that exists on the system on which DXXMQGEN runs. *DAD-file-name* is an input parameter of type VARCHAR(80).

override-type

Specifies what the *override* parameter does. Possible values are:

NO_OVERRIDE

The *override* parameter does not override the condition in the DAD file. This is the default.

SQL_OVERRIDE

The DAD file uses SQL mapping, and the *override* parameter contains an SQL statement that overrides the SQL statement in the DAD file.

XML_OVERRIDE

The DAD file uses RDB_node mapping, and the *override* parameter contains conditions that override the RDB_node mapping in the DAD file.

override-type is an input parameter of type INTEGER. The integer equivalents of the *override-type* values are defined in the dxx.h file.

override

Specifies a string that overrides the condition in the DAD file. The contents of the string depend on the value of the *override-type* parameter:

- # • If *override-type* is NO_OVERRIDE, *override* contains a null string. This is the default.
- # • If *override-type* is SQL_OVERRIDE, *override* contains a valid SQL statement that overrides the SQL statement in the DAD file.
- # • If *override-type* is XML_OVERRIDE, *override* contains one or more expressions that are separated by AND. Each expression must be enclosed in double quotation marks. This *override* value overrides the RDB_node mapping in the DAD file.

override is an input parameter of type VARCHAR(1024).

max-rows

Specifies the maximum number of XML documents that DXXMQGEN can send to the MQ message queue. The default is 1.

max-rows is an input parameter of type INTEGER.

num-rows

The actual number of XML documents that DXXMQGEN sends to the MQ message queue.

num-rows is an output parameter of type INTEGER.

status

Contains information that indicates whether DXXMQGEN ran successfully. The format of *status* is *dxx-rc:sqlcode:mq-num-msgs*, where:


```
# • dxx-rc is the return code from accessing XML Extender. dxx-rc values are
# defined in dxxrc.h.
#
# • sqlcode is 0 if DXXMQGEN ran successfully, or the SQLCODE from the most
# recent unsuccessful SQL statement if DXXMQGEN ran unsuccessfully.
#
# • mq-num-msgs is the number of messages that were successfully sent to the
# MQ message queue.
```

Example of DXMQGEN invocation

The following example calls the single-phase commit version of DXXMQGEN to generate XML documents and send them to an MQ message queue. The calling program does not override the definitions in the DAD file. For a complete example of DXXMQGEN invocation, see DSN8OXGR in DSN810.SDSNSAMP.

```
#include "dxx.h"
#include "dxxrc.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char      serviceName[48]; /* WebSphere MQ service name */
char      policyName[48]; /* WebSphere MQ policy name */
char      dadFileName[80]; /* DAD file name */
short     overrideType; /* defined in dxx.h */
char      override[2]; /* Override string for DAD */
short     max_row; /* Maximum number of documents*/
short     num_row; /* Actual number of documents */
char      status[20]; /* Status of DXMQGEN call */
/* DXMQGEN is GENERAL WITH NULLS, so parameters need indicators */
short     serviceName_ind; /* Indicator var for serviceName */
short     policyName_ind; /* Indicator var for policyName */
short     dadFileName_ind; /* Indicator var for dadFileName */
short     ovrtype_ind; /* Indicator var for overrideType */
short     ov_ind; /* Indicator var for override */
short     maxrow_ind; /* Indicator var for maxrow */
short     numrow_ind; /* Indicator var for numrow */
short     status_ind; /* Indicator var for status */
EXEC SQL END DECLARE SECTION;
/* Status fields */
int dxx_rc=0;
int dxx_sql=0;
int dxx_mq=0;
/* Get the service name and policy name for the MQ message queue */
strcpy(serviceName,"DB2.DEFAULT.SERVICE");
strcpy(policyName,"DB2.DEFAULT.POLICY");
/* Get the name of the DAD file for the DB2 tables */
strcpy(dadFileName,"tmp/getstart_xcollection.dad");
/* Put null in the override parameter because we are not going */
/* to override the values in the DAD file */
override[0] = '\0';
overrideType = NO_OVERRIDE;
/* Indicate that we do not want to transfer more than 500 */
/* documents */
max_row = 500;
/* Initialize the output variables */
num_row = 0;
status[0] = '\0';
/* Set the indicators to 0 for the parameters that have non-null */
/* values */
dadFileName_ind = 0;
serviceName_ind = 0;
policyName_ind = 0;
maxrow_ind = 0;
/* Initialize the indicators for the output parameters */
numrow_ind = -1;
```

```

#           status_ind = -1;
#           /* Call the store procedure */
#           EXEC SQL CALL DMQXML1C.DXXMQGEN(:serviceName:serviceName_ind,
#           :policyName:policyName_ind,
#           :dadFileName:dadFileName_ind,
#           :overrideType:ovtype_ind,
#           :override:ov_ind,
#           :max_row:maxrow_ind,
#           :num_row:numrow_ind,
#           :status:status_ind);
#           printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
#           /* Get the status fields from the status parameter and print them */
#           sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
#           printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);

```

DXXMQGEN output

If DXXMQGEN executes successfully, the number of documents indicated by the *mq-num-msgs* field of the *status* parameter are extracted from DB2 tables and inserted into the MQ message queue. If DXXMQGEN does not execute successfully, the contents of the *status* parameter indicate the problem.

Depreciated: Send XML documents to an MQ message queue (DXXMQRETRIEVE)

Restriction: DXXMQRETRIEVE has been deprecated.

The DXXMQRETRIEVE stored procedure constructs XML documents from data that is stored in DB2 tables that are specified in an enabled XML collection, and sends the XML documents to an MQ message queue.

Use DXXMQRETRIEVE for XML documents with a length of up to 3KB.

There are two versions of DXXMQRETRIEVE:

- A single-phase commit version, with schema name DMQXML1C.
- A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQRETRIEVE

DXXMQRETRIEVE runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

DXXMQRETRIEVE requires that WebSphere MQ and XML Extender are installed.

See *DB2 Installation Guide* for installation instructions.

Authorization required for DXXMQRETRIEVE

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 DXXMQRETRIEVE
- Ownership of the stored procedure
- SYSADM authority

DXXMQRETRIEVE syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking DXXMQRETRIEVE. Because the linkage convention for DXXMQRETRIEVE 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.
 #

```
#  ►CALL DMQXML1C.DXXMQRETRIEVE( service-name, policy-name,
#  DMQXML2C NULL NULL,
#  ►XML-collection-name, override-type, override, max-rows, num-msgs, status)►
#
#
#
#
```

DXXMQRETRIEVE option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination to
 # which the message is to be sent. The service point is defined in the DSNAMT
 # repository file. If *service-name* is not listed in the DSNAMT repository file, or
 # *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot
 # be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the
 # message. The service policy is defined in the DSNAMT repository file. If
 # *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not
 # specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be
 # blank, a null string, or have trailing blanks.

XML-collection-name

Specifies the name of the XML collection that specifies the DB2 tables from
 # which the XML documents are to be retrieved. *XML-collection-name* must be
 # specified, and must be the name of a valid XML collection that exists on the
 # system on which DXXMQRETRIEVE runs. *XML-collection-name* is an input
 # parameter of type VARCHAR(80).

override-type

Specifies what the *override* parameter does. Possible values are:

NO_OVERRIDE

The *override* parameter does not override the XML collection. This is
 # the default.

SQL_OVERRIDE

The DAD file uses SQL mapping, and the *override* parameter contains
 # an SQL statement that overrides the SQL statement in the XML
 # collection.

XML_OVERRIDE

The DAD file uses RDB_node mapping, and the *override* parameter
 # contains conditions that override the conditions in the XML collection.

override-type is an input parameter of type INTEGER. The integer equivalents
 # of the *override-type* values are defined in the dxx.h file.

override-type

Specifies what the *override* parameter does. Possible values are:

```

# NO_OVERRIDE
# The override parameter does not override the condition in the DAD file.
# This is the default.

# SQL_OVERRIDE
# The DAD file uses SQL mapping, and the override parameter contains
# an SQL statement that overrides the SQL statement in the DAD file.

# XML_OVERRIDE
# The DAD file uses RDB_node mapping, and the override parameter
# contains conditions that override the RDB_node mapping in the DAD
# file.

# override-type is an input parameter of type INTEGER. The integer equivalents
# of the override-type values are defined in the dxx.h file.

# override
# Specifies a string that overrides the condition in the DAD file. The contents of
# the string depend on the value of the override-type parameter:
#
# • If override-type is NO_OVERRIDE, override contains a null string. This is the
#   default.
#
# • If override-type is SQL_OVERRIDE, override contains a valid SQL statement
#   that overrides the SQL statement in the DAD file.
#
# • If override-type is XML_OVERRIDE, override contains one or more expressions
#   that are separated by AND. Each expression must be enclosed in double
#   quotation marks. This override value overrides the RDB_node mapping in the
#   DAD file.

# override is an input parameter of type VARCHAR(1024).

# max-rows
# Specifies the maximum number of XML documents that DXXMQRETRIEVE
# can send to the MQ message queue. The default is 1.

# max-rows is an input parameter of type INTEGER.

# num-rows
# The actual number of XML documents that DXXMQRETRIEVE sends to the
# MQ message queue.

# num-rows is an output parameter of type INTEGER.

# status
# Contains information that indicates whether DXXMQRETRIEVE ran
# successfully. The format of status is dxx-rc:sqlcode:mq-num-msgs, where:
#
# • dxx-rc is the return code from accessing XML Extender. dxx-rc values are
#   defined in dxxrc.h.
#
# • sqlcode is 0 if DXXMQRETRIEVE ran successfully, or the SQLCODE from the
#   most recent unsuccessful SQL statement if DXXMQRETRIEVE ran
#   unsuccessfully.
#
# • mq-num-msgs is the number of messages that were successfully sent to the
#   MQ message queue.

# status is an output parameter of type CHAR(20).

```

Example of DXXMQRETRIEVE invocation

The following example calls the single-phase commit version of DXXMQRETRIEVE to generate XML documents and send them to an MQ message queue. The calling

program does not override the definitions in the DAD file. For a complete example of DXXMQRETRIEVE invocation, see DSN8QXGR in DSN810.SDSNSAMP.

```
#
#
# include "dxx.h"
# include "dxxrc.h"
# EXEC SQL INCLUDE SQLCA;
# EXEC SQL BEGIN DECLARE SECTION;
# char      serviceName[48]; /* WebSphere MQ service name */
# char      policyName[48]; /* WebSphere MQ policy name */
# char      collectionName[80]; /* XML collection name */
# short     overrideType; /* defined in dxx.h */
# char      override[2]; /* Override string for DAD */
# short     max_row; /* Maximum number of documents*/
# short     num_row; /* Actual number of documents */
# char      status[20]; /* Status of DXXMQRETRIEVE call */
# /* DXXMQRETRIEVE is GENERAL WITH NULLS, so parameters need indicators */
# short     serviceName_ind; /* Indicator var for serviceName */
# short     policyName_ind; /* Indicator var for policyName */
# short     collectionName_ind; /* Indicator var for dadFileName */
# short     ovttype_ind; /* Indicator var for overrideType */
# short     ov_ind; /* Indicator var for override */
# short     maxrow_ind; /* Indicator var for maxrow */
# short     numrow_ind; /* Indicator var for numrow */
# short     status_ind; /* Indicator var for status */
# EXEC SQL END DECLARE SECTION;
# /* Status fields */
# int dxx_rc=0;
# int dxx_sql=0;
# int dxx_mq=0;
# /* Get the service name and policy name for the MQ message queue */
# strcpy(serviceName,"DB2.DEFAULT.SERVICE");
# strcpy(policyName,"DB2.DEFAULT.POLICY");
# /* Get the XML collection name */
# strcpy(collectionName,"sales_ord");
# /* Put null in the override parameter because we are not going */
# /* to override the values in the DAD file */
# override[0] = '\0';
# overrideType = NO_OVERRIDE;
# /* Indicate that we do not want to transfer more than 500 */
# /* documents */
# max_row = 500;
# /* Initialize the output variables */
# num_row = 0;
# status[0] = '\0';
# /* Set the indicators to 0 for the parameters that have non-null */
# /* values */
# collectionName_ind = 0;
# serviceName_ind = 0;
# policyName_ind = 0;
# maxrow_ind = 0;
# /* Initialize the indicators for the output parameters */
# numrow_ind = -1;
# status_ind = -1;
# /* Call the store procedure */
# EXEC SQL CALL DMQXML1C.DXXMQRETRIEVE(:serviceName:serviceName_ind,
# :policyName:policyName_ind,
# :collectionName:collectionName_ind,
# :overrideType:ovttype_ind,
# :override:ov_ind,
# :max_row:maxrow_ind,
# :num_row:numrow_ind,
# :status:status_ind);
# printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
# /* Get the status fields from the status parameter and print them */
# sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
# printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);
#
```

DXXMQRETRIEVE output

If DXXMQRETRIEVE executes successfully, the number of documents indicated by

the *mq-num-msgs* field of the *status* parameter are extracted from DB2 tables and

inserted into the MQ message queue. If DXXMQRETRIEVE does not execute

successfully, the contents of the *status* parameter indicate the problem.

Deprecated: Send large XML documents to an MQ message queue

(DXXMQGENCLOB)

Restriction: DXXMQGENCLOB has been deprecated.

#

The DXXMQGENCLOB stored procedure constructs XML documents from data

that is stored in DB2 tables that are specified in a document access definition

(DAD) file, and sends the XML documents to an MQ message queue.

#

Use DXXMQGENCLOB for XML documents with a record length of up to 32KB.

#

There are two versions of DXXMQGENCLOB:

• A single-phase commit version, with schema name DMQXML1C.

• A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQGENCLOB

DXXMQGENCLOB runs in a WLM-established stored procedure address space

and uses the Resource Recovery Services attachment facility to connect to DB2.

#

DXXMQGENCLOB requires that WebSphere MQ and XML Extender are installed.

#

See *DB2 Installation Guide* for installation instructions.

Authorization required for DXXMQGENCLOB

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 DXXMQGENCLOB

• Ownership of the stored procedure

• SYSADM authority

DXXMQGENCLOB syntax diagram

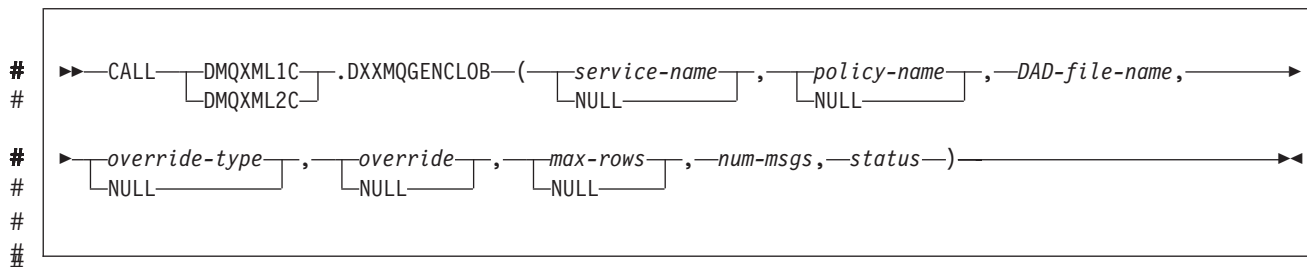
The following syntax diagram shows the SQL CALL statement for invoking

DXXMQGENCLOB. Because the linkage convention for DXXMQGENCLOB 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.



DXXMQGENCLOB option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination to which the message is to be sent. The service point is defined in the DSNAMT repository file. If *service-name* is not listed in the DSNAMT repository file, or *service-name* is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the message. The service policy is defined in the DSNAMT repository file. If *policy-name* is not listed in the DSNAMT repository file, or *policy-name* is not specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be blank, a null string, or have trailing blanks.

DAD-file-name

Specifies the name of the (DAD) file that maps the XML documents to DB2 tables. *DAD-file-name* must be specified, and must be the name of a valid DAD file that exists on the system on which DXXMQGENCLOB runs. *DAD-file-name* is an input parameter of type VARCHAR(80).

override-type

Specifies what the *override* parameter does. Possible values are:

NO_OVERRIDE

The *override* parameter does not override the condition in the DAD file. This is the default.

SQL_OVERRIDE

The DAD file uses SQL mapping, and the *override* parameter contains an SQL statement that overrides the SQL statement in the DAD file.

XML_OVERRIDE

The DAD file uses RDB_node mapping, and the *override* parameter contains conditions that override the RDB_node mapping in the DAD file.

override-type is an input parameter of type INTEGER. The integer equivalents of the *override-type* values are defined in the dxx.h file.

override

Specifies a string that overrides the condition in the DAD file. The contents of the string depend on the value of the *override-type* parameter:

- If *override-type* is NO_OVERRIDE, *override* contains a null string. This is the default.
- If *override-type* is SQL_OVERRIDE, *override* contains a valid SQL statement that overrides the SQL statement in the DAD file.
- If *override-type* is XML_OVERRIDE, *override* contains one or more expressions that are separated by AND. Each expression must be enclosed in double quotation marks. This *override* value overrides the RDB_node mapping in the DAD file.

override is an input parameter of type VARCHAR(1024).


```
#
#      max-rows
#      Specifies the maximum number of XML documents that DXXMQGENCLOB
#      can send to the MQ message queue. The default is 1.
#
#      max-rows is an input parameter of type INTEGER.
#
#      num-rows
#      The actual number of XML documents that DXXMQGENCLOB sends to the
#      MQ message queue.
#
#      num-rows is an output parameter of type INTEGER.
#
#      status
#      Contains information that indicates whether DXXMQGENCLOB ran
#      successfully. The format of status is dxx-rc:sqlcode:mq-num-msgs, where:
#
#      • dxx-rc is the return code from accessing XML Extender. dxx-rc values are
#      defined in dxxrc.h.
#
#      • sqlcode is 0 if DXXMQGENCLOB ran successfully, or the SQLCODE from the
#      most recent unsuccessful SQL statement if DXXMQGENCLOB ran
#      unsuccessfully.
#
#      • mq-num-msgs is the number of messages that were successfully sent to the
#      MQ message queue.
#
#      status is an output parameter of type CHAR(20).
```

Example of DXXMQGENCLOB invocation

The following example calls the two-phase commit version of DXXMQGENCLOB to generate CLOB XML documents and send them to an MQ message queue. The calling program does not override the definitions in the DAD file. For a complete example of DXXMQGENCLOB invocation, see DSN8QXGR in DSN810.SDSNSAMP.

```
#
#      #include "dxx.h"
#      #include "dxxrc.h"
#      EXEC SQL INCLUDE SQLCA;
#      EXEC SQL BEGIN DECLARE SECTION;
#      char      serviceName[48]; /* WebSphere MQ service name */
#      char      policyName[48]; /* WebSphere MQ policy name */
#      char      dadFileName[80]; /* DAD file name */
#      short     overrideType; /* defined in dxx.h */
#      char      override[2]; /* Override string for DAD */
#      short     max_row; /* Maximum number of documents*/
#      short     num_row; /* Actual number of documents */
#      char      status[20]; /* Status of DXXMQGENCLOB call */
#      /* DXXMQGENCLOB is GENERAL WITH NULLS, so parameters need indicators */
#      short     serviceName_ind; /* Indicator var for serviceName */
#      short     policyName_ind; /* Indicator var for policyName */
#      short     dadFileName_ind; /* Indicator var for dadFileName */
#      short     ovtype_ind; /* Indicator var for overrideType */
#      short     ov_ind; /* Indicator var for override */
#      short     maxrow_ind; /* Indicator var for maxrow */
#      short     numrow_ind; /* Indicator var for numrow */
#      short     status_ind; /* Indicator var for status */
#      EXEC SQL END DECLARE SECTION;
#      /* Status fields */
#      int dxx_rc=0;
#      int dxx_sql=0;
#      int dxx_mq=0;
#      /* Get the service name and policy name for the MQ message queue */
#      strcpy(serviceName,"DB2.DEFAULT.SERVICE");
#      strcpy(policyName,"DB2.DEFAULT.POLICY");
#      /* Get the name of the DAD file for the DB2 tables */
#      strcpy(dadFileName,"/tmp/getstart_xcollection.dad");
```



```

#          /* Put null in the override parameter because we are not going */
#          /* to override the values in the DAD file */
#          override[0] = '\0';
#          overrideType = NO_OVERRIDE;
#          /* Indicate that we do not want to transfer more than 500 */
#          /* documents */
#          max_row = 500;
#          /* Initialize the output variables */
#          num_row = 0;
#          status[0] = '\0';
#          /* Set the indicators to 0 for the parameters that have non-null */
#          /* values */
#          dadFileName_ind = 0;
#          serviceName_ind = 0;
#          policyName_ind = 0;
#          maxrow_ind = 0;
#          /* Initialize the indicators for the output parameters */
#          numrow_ind = -1;
#          status_ind = -1;
#          /* Call the store procedure */
#          EXEC SQL CALL DMQXML2C.DXXMQGENCLOB(:serviceName:serviceName_ind,
#          :policyName:policyName_ind,
#          :dadFileName:dadFileName_ind,
#          :overrideType:ovtype_ind,
#          :override:ov_ind,
#          :max_row:maxrow_ind,
#          :num_row:numrow_ind,
#          :status:status_ind);
#          printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
#          /* Get the status fields from the status parameter and print them */
#          sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
#          printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);

```

DXXMQGENCLOB output

If DXXMQGENCLOB executes successfully, the number of documents indicated by the *mq-num-msgs* field of the *status* parameter are extracted from DB2 tables and inserted into the MQ message queue. If DXXMQGENCLOB does not execute successfully, the contents of the *status* parameter indicate the problem.

Deprecated: Send XML documents to an MQ message queue (DXXMQRETRIEVECLOB)

Restriction: DXXMQRETRIEVECLOB has been deprecated.

The DXXMQRETRIEVECLOB stored procedure constructs XML documents from data that is stored in DB2 tables that are specified in an enabled XML collection, and sends the XML documents to an MQ message queue.

Use DXXMQRETRIEVECLOB for XML documents with a length of up to 32KB.

There are two versions of DXXMQRETRIEVECLOB:

- A single-phase commit version, with schema name DMQXML1C.
- A two-phase commit version, with schema name DMQXML2C.

Environment for DXXMQRETRIEVECLOB

DXXMQRETRIEVECLOB runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

DXXMQRETRIEVECLOB requires that WebSphere MQ and XML Extender are
installed.

See *DB2 Installation Guide* for installation instructions.

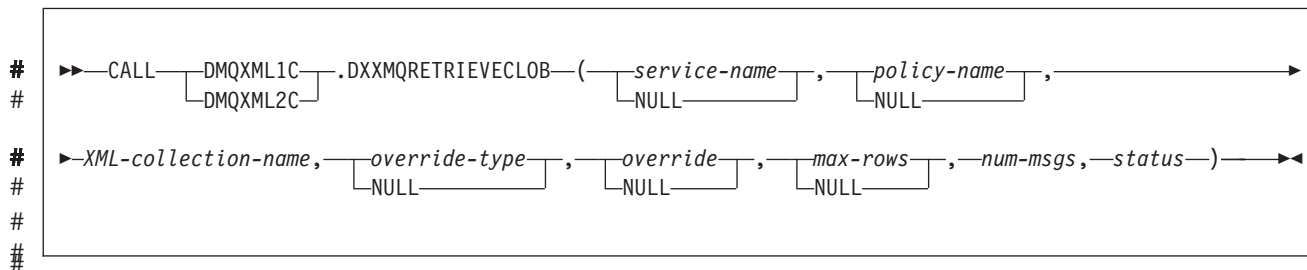
Authorization required for DXXMQRETRIEVECLOB

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 DXXMQRETRIEVECLOB
- # • Ownership of the stored procedure
- # • SYSADM authority

DXXMQRETRIEVECLOB syntax diagram

The following syntax diagram shows the SQL CALL statement for invoking
DXXMQRETRIEVECLOB. Because the linkage convention for
DXXMQRETRIEVECLOB 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.



DXXMQRETRIEVECLOB option descriptions

service-name

Specifies the service point that is the logical WebSphere MQ destination to
which the message is to be sent. The service point is defined in the DSNAMT
repository file. If *service-name* is not listed in the DSNAMT repository file, or
service-name is not specified, DB2.DEFAULT.SERVICE is used.

service-name is an input parameter of type VARCHAR(48). *service-name* cannot
be blank, a null string, or have trailing blanks.

policy-name

Specifies the WebSphere MQ AMI service policy that is used to handle the
message. The service policy is defined in the DSNAMT repository file. If
policy-name is not listed in the DSNAMT repository file, or *policy-name* is not
specified, DB2.DEFAULT.POLICY is used.

policy-name is an input parameter of type VARCHAR(48). *policy-name* cannot be
blank, a null string, or have trailing blanks.

XML-collection-name

Specifies the name of the XML collection that specifies the DB2 tables from
which the XML documents are to be retrieved. *XML-collection-name* must be
specified, and must be the name of a valid XML collection that exists on the
system on which DXXMQRETRIEVECLOB runs. *XML-collection-name* is an
input parameter of type VARCHAR(80).

```

#
#      override-type
#      Specifies what the override parameter does. Possible values are:
#
#      NO_OVERRIDE
#          The override parameter does not override the condition in the DAD file.
#          This is the default.
#
#      SQL_OVERRIDE
#          The DAD file uses SQL mapping, and the override parameter contains
#          an SQL statement that overrides the SQL statement in the DAD file.
#
#      XML_OVERRIDE
#          The DAD file uses RDB_node mapping, and the override parameter
#          contains conditions that override the RDB_node mapping in the DAD
#          file.
#
#      override-type is an input parameter of type INTEGER. The integer equivalents
#      of the override-type values are defined in the dxx.h file.
#
#      override
#      Specifies a string that overrides the condition in the DAD file. The contents of
#      the string depend on the value of the override-type parameter:
#
#      • If override-type is NO_OVERRIDE, override contains a null string. This is the
#        default.
#
#      • If override-type is SQL_OVERRIDE, override contains a valid SQL statement
#        that overrides the SQL statement in the DAD file.
#
#      • If override-type is XML_OVERRIDE, override contains one or more expressions
#        that are separated by AND. Each expression must be enclosed in double
#        quotation marks. This override value overrides the RDB_node mapping in the
#        DAD file.
#
#      override is an input parameter of type VARCHAR(1024).
#
#      max-rows
#      Specifies the maximum number of XML documents that
#      DXXMQRETRIEVECLOB can send to the MQ message queue. The default is 1.
#
#      max-rows is an input parameter of type INTEGER.
#
#      num-rows
#      The actual number of XML documents that DXXMQRETRIEVECLOB sends to
#      the MQ message queue.
#
#      num-rows is an output parameter of type INTEGER.
#
#      status
#      Contains information that indicates whether DXXMQRETRIEVECLOB ran
#      successfully. The format of status is dxx-rc:sqlcode:mq-num-msgs, where:
#
#      • dxx-rc is the return code from accessing XML Extender. dxx-rc values are
#        defined in dxxrc.h.
#
#      • sqlcode is 0 if DXXMQRETRIEVECLOB ran successfully, or the SQLCODE
#        from the most recent unsuccessful SQL statement if
#        DXXMQRETRIEVECLOB ran unsuccessfully.
#
#      • mq-num-msgs is the number of messages that were successfully sent to the
#        MQ message queue.
#
#      status is an output parameter of type CHAR(20).

```

Example of DXXMQRETRIEVECLOB invocation

The following example calls the single-phase commit version of DXXMQRETRIEVECLOB to generate XML documents and send them to an MQ message queue. The calling program does not override the definitions in the DAD file. For a complete example of DXXMQRETRIEVECLOB invocation, see DSN8QXGR in DSN810.SDSNSAMP.

```
#include "dxx.h"
#include "dxxrc.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char      serviceName[48];    /* WebSphere MQ service name */
char      policyName[48];    /* WebSphere MQ policy name */
char      collectionName[80]; /* XML collection name */
short     overrideType;      /* defined in dxx.h */
char      override[2];       /* Override string for DAD */
short     max_row;           /* Maximum number of documents*/
short     num_row;           /* Actual number of documents */
char      status[20];        /* Status of DXXMQRETRIEVECLOB call */
/* DXXMQRETRIEVECLOB is GENERAL WITH NULLS, so parameters need indicators */
short     serviceName_ind;   /* Indicator var for serviceName */
short     policyName_ind;    /* Indicator var for policyName */
short     collectionName_ind; /* Indicator var for dadFileName */
short     ovtype_ind;        /* Indicator var for overrideType */
short     ov_ind;            /* Indicator var for override */
short     maxrow_ind;        /* Indicator var for maxrow */
short     numrow_ind;        /* Indicator var for numrow */
short     status_ind;        /* Indicator var for status */
EXEC SQL END DECLARE SECTION;
/* Status fields */
int dxx_rc=0;
int dxx_sql=0;
int dxx_mq=0;
/* Get the service name and policy name for the MQ message queue */
strcpy(serviceName,"DB2.DEFAULT.SERVICE");
strcpy(policyName,"DB2.DEFAULT.POLICY");
/* Get the XML collection name */
strcpy(collectionName,"sales_ord");
/* Put null in the override parameter because we are not going
/* to override the values in the DAD file */
override[0] = '\0';
overrideType = NO_OVERRIDE;
/* Indicate that we do not want to transfer more than 500
/* documents */
max_row = 500;
/* Initialize the output variables */
num_row = 0;
status[0] = '\0';
/* Set the indicators to 0 for the parameters that have non-null
/* values */
collectionName_ind = 0;
serviceName_ind = 0;
policyName_ind = 0;
maxrow_ind = 0;
/* Initialize the indicators for the output parameters */
numrow_ind = -1;
status_ind = -1;
/* Call the store procedure */
EXEC SQL CALL DMQXML1C.DXXMQRETRIEVECLOB(:serviceName:serviceName_ind,
:policyName:policyName_ind,
:collectionName:collectionName_ind,
:overrideType:ovtype_ind,
:override:ov_ind,
:max_row:maxrow_ind,
:num_row:numrow_ind,
:status:status_ind);
```

```
#      printf("SQLCODE from CALL: %d\n",sqlca.sqlcode);
#      /* Get the status fields from the status parameter and print them */
#      sscanf(status,"%d:%d:%d",&dxx_rc,&dxx_sql,&dxx_mq);
#      printf("Status fields: dxx_rc=%d dxx_sql=%d dxx_mq=%d\n",dxx_rc,dxx_sql,dxx_mq);
```

DXXMQRETRIEVECLOB output

```
#      If DXXMQRETRIEVECLOB executes successfully, the number of documents
#      indicated by the mq-num-msgs field of the status parameter are extracted from DB2
#      tables and inserted into the MQ message queue. If DXXMQRETRIEVECLOB does
#      not execute successfully, the contents of the status parameter indicate the problem.
```

Appendix K. How to use the DB2 library

Titles of books in the library begin with DB2 Universal Database for z/OS Version 8. However, references from one book in the library to another are shortened and do not include the product name, version, and release. Instead, they point directly to the section that holds the information. For a complete list of books in the library, and the sections in each book, see the bibliography at the back of this book.

The most rewarding task associated with a database management system is asking questions of it and getting answers, the task called *end use*. Other tasks are also necessary—defining the parameters of the system, putting the data in place, and so on. The tasks that are associated with DB2 are grouped into the following major categories (but supplemental information relating to all of the following tasks for new releases of DB2 can be found in *DB2 Release Planning Guide*.

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

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

If you want to set up a DB2 subsystem to meet the requirements of the Common Criteria, you need *DB2 Common Criteria Guide*, which contains information that supersedes other information in the DB2 UDB for z/OS library regarding Common Criteria.

End use: End users issue SQL statements to retrieve data. They can also insert, update, or delete data, with SQL statements. They might need an introduction to SQL, detailed instructions for using SPUFI, and an alphabetized reference to the types of SQL statements. This information is found in this book, and *DB2 SQL Reference*.

End users can also issue SQL statements through the DB2 Query Management Facility (QMF) or some other program, and the library for that licensed program might provide all the instruction or reference material they need. For a list of the titles in the DB2 QMF library, see the bibliography at the end of this book.

Application programming: Some users access DB2 without knowing it, using programs that contain SQL statements. DB2 application programmers write those programs. Because they write SQL statements, they need the same resources that end users do.

Application programmers also need instructions on many other topics:

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

- How to write applications in the Java programming language to access DB2 servers

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

If you will be working in a distributed environment, you will need *DB2 Reference for Remote DRDA Requesters and Servers*.

Information about writing applications across operating systems can be found in *IBM DB2 Universal Database SQL Reference for Cross-Platform Development*.

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

- Part 2 (Volume 1) of *DB2 Administration Guide* discusses the decisions that must be made when designing a database and tells how to implement the design by creating and altering DB2 objects, loading data, and adjusting to changes.
- Part 3 (Volume 1) of *DB2 Administration Guide* describes ways of controlling access to the DB2 system and to data within DB2, to audit aspects of DB2 usage, and to answer other security and auditing concerns.
- Part 4 (Volume 1) of *DB2 Administration Guide* describes the steps in normal day-to-day operation and discusses the steps one should take to prepare for recovery in the event of some failure.
- Part 5 (Volume 2) of *DB2 Administration Guide* explains how to monitor the performance of the DB2 system and its parts. It also lists things that can be done to make some parts run faster.

If you will be using the RACF access control module for DB2 authorization checking, you will need *DB2 RACF Access Control Module Guide*.

If you are involved with DB2 only to design the database, or plan operational procedures, you need *DB2 Administration Guide*. If you also want to carry out your own plans by creating DB2 objects, granting privileges, running utility jobs, and so on, you also need:

- *DB2 SQL Reference*, which describes the SQL statements you use to create, alter, and drop objects and grant and revoke privileges
- *DB2 Utility Guide and Reference*, which explains how to run utilities
- *DB2 Command Reference*, which explains how to run commands

If you will be using data sharing, you need *DB2 Data Sharing: Planning and Administration*, which describes how to plan for and implement data sharing.

Additional information about system and database administration can be found in *DB2 Messages* and *DB2 Codes*, which list messages and codes issued by DB2, with explanations and suggested responses.

Diagnosis: Diagnosticians detect and describe errors in the DB2 program. They might also recommend or apply a remedy. The documentation for this task is in *DB2 Diagnosis Guide and Reference*, *DB2 Messages*, and *DB2 Codes*.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is 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.

Programming interface information

This book is intended to help you to write programs that contain SQL statements. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by IBM DB2 Universal Database Server for z/OS (DB2 UDB for z/OS).

General-use Programming Interfaces allow the customer to write programs that obtain the services of DB2 UDB for z/OS.

However, this book also documents Product-sensitive Programming Interface and Associated Guidance Information.

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and

implementation, it is to be expected that programs written to such interfaces may need to be changed 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, either by an introductory statement to a chapter or section or by the following marking:

Product-sensitive Programming Interface Product-sensitive Programming Interface and Associated Guidance Information ... End of Product-sensitive Programming Interface

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

BookManager	IBM Registry
CICS	IMS
CICS Connection	iSeries
CT	Language Environment
DataPropagator	MVS
DB2	MVS/ESA
DB2 Connect	Notes
DB2 Universal Database	OS/390
DFSMSdftp	Parallel Sysplex
DFSMSdss	PR/SM
DFSMSHsm	QMF
Distributed Relational Database Architecture	RACF
DRDA	Redbooks
Enterprise Storage Server	System/390
ES/3090	TotalStorage
eServer	VTAM
FlashCopy	WebSphere
IBM	z/OS

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. 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.

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library.

A

abend. Abnormal end of task.

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with DB2.

abnormal end of task (abend). Termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve during execution.

access method services. The facility that is used to define and reproduce VSAM key-sequenced data sets.

access path. The path that is used to locate data that is specified in SQL statements. An access path can be indexed or sequential.

active log. The portion of the DB2 log to which log records are written as they are generated. The active log always contains the most recent log records, whereas the archive log holds those records that are older and no longer fit on the active log.

active member state. A state of a member of a data sharing group. The cross-system coupling facility identifies each active member with a group and associates the member with a particular task, address space, and z/OS system. A member that is not active has either a failed member state or a quiesced member state.

address space. A range of virtual storage pages that is identified by a number (ASID) and a collection of segment and page tables that map the virtual pages to real pages of the computer's memory.

address space connection. The result of connecting an allied address space to DB2. Each address space that contains a task that is connected to DB2 has exactly one address space connection, even though more than one task control block (TCB) can be present. See also *allied address space* and *task control block*.

| **address space identifier (ASID).** A unique
| system-assigned identifier for and address space.

administrative authority. A set of related privileges that DB2 defines. When you grant one of the administrative authorities to a person's ID, the person has all of the privileges that are associated with that administrative authority.

after trigger. A trigger that is defined with the trigger activation time AFTER.

agent. As used in DB2, the structure that associates all processes that are involved in a DB2 unit of work. An *allied agent* is generally synonymous with an *allied thread*. *System agents* are units of work that process tasks that are independent of the allied agent, such as prefetch processing, deferred writes, and service tasks.

aggregate function. An operation that derives its
result by using values from one or more rows. Contrast
with *scalar function*.

alias. An alternative name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem.

allied address space. An area of storage that is external to DB2 and that is connected to DB2. An allied address space is capable of requesting DB2 services.

allied thread. A thread that originates at the local DB2 subsystem and that can access data at a remote DB2 subsystem.

allocated cursor. A cursor that is defined for stored procedure result sets by using the SQL ALLOCATE CURSOR statement.

already verified. An LU 6.2 security option that allows DB2 to provide the user's verified authorization ID when allocating a conversation. With this option, the user is not validated by the partner DB2 subsystem.

| **ambiguous cursor.** A database cursor that is in a plan
| or package that contains either PREPARE or EXECUTE
| IMMEDIATE SQL statements, and for which the
| following statements are true: the cursor is not defined
| with the FOR READ ONLY clause or the FOR UPDATE
| OF clause; the cursor is not defined on a read-only
| result table; the cursor is not the target of a WHERE
| CURRENT clause on an SQL UPDATE or DELETE
| statement.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

APAR. Authorized program analysis report.

APAR fix corrective service. A temporary correction of an IBM software defect. The correction is temporary,

because it is usually replaced at a later date by a more permanent correction, such as a program temporary fix (PTF).

APF. Authorized program facility.

API. Application programming interface.

APPL. A VTAM® network definition statement that is used to define DB2 to VTAM as an application program that uses SNA LU 6.2 protocols.

application. A program or set of programs that performs a task; for example, a payroll application.

application-directed connection. A connection that an application manages using the SQL CONNECT statement.

application plan. The control structure that is produced during the bind process. DB2 uses the application plan to process SQL statements that it encounters during statement execution.

application process. The unit to which resources and locks are allocated. An application process involves the execution of one or more programs.

application programming interface (API). A functional interface that is supplied by the operating system or by a separately orderable licensed program that allows an application program that is written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester. The component on a remote system that generates DRDA requests for data on behalf of an application. An application requester accesses a DB2 database server using the DRDA application-directed protocol.

application server. The target of a request from a remote application. In the DB2 environment, the application server function is provided by the distributed data facility and is used to access DB2 data from remote applications.

archive log. The portion of the DB2 log that contains log records that have been copied from the active log.

ASCII. An encoding scheme that is used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC* and *Unicode*.

| **ASID.** Address space identifier.

attachment facility. An interface between DB2 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2.

attribute. A characteristic of an entity. For example, in database design, the phone number of an employee is one of that employee's attributes.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges is allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

authorized program analysis report (APAR). A report of a problem that is caused by a suspected defect in a current release of an IBM supplied program.

authorized program facility (APF). A facility that permits the identification of programs that are authorized to use restricted functions.

| **automatic query rewrite.** A process that examines an
| SQL statement that refers to one or more base tables,
| and, if appropriate, rewrites the query so that it
| performs better. This process can also determine
| whether to rewrite a query so that it refers to one or
| more materialized query tables that are derived from
| the source tables.

auxiliary index. An index on an auxiliary table in which each index entry refers to a LOB.

auxiliary table. A table that stores columns outside the table in which they are defined. Contrast with *base table*.

B

backout. The process of undoing uncommitted changes that an application process made. This might be necessary in the event of a failure on the part of an application process, or as a result of a deadlock situation.

backward log recovery. The fourth and final phase of restart processing during which DB2 scans the log in a backward direction to apply UNDO log records for all aborted changes.

base table. (1) A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with *result table* and *temporary table*.

(2) A table containing a LOB column definition. The actual LOB column data is not stored with the base table. The base table contains a row identifier for each row and an indicator column for each of its LOB columns. Contrast with *auxiliary table*.

base table space. A table space that contains base tables.

basic predicate. A predicate that compares two values.

basic sequential access method (BSAM). An access method for storing or retrieving data blocks in a continuous sequence, using either a sequential-access or a direct-access device.

| **batch message processing program.** In IMS, an
| application program that can perform batch-type
| processing online and can access the IMS input and
| output message queues.

before trigger. A trigger that is defined with the trigger activation time BEFORE.

binary integer. A basic data type that can be further classified as small integer or large integer.

binary large object (BLOB). A sequence of bytes in
which the size of the value ranges from 0 bytes to
2 GB-1. Such a string has a CCSID value of 65535.

binary string. A sequence of bytes that is not associated with a CCSID. For example, the BLOB data type is a binary string.

bind. The process by which the output from the SQL precompiler is converted to a usable control structure, often called an access plan, application plan, or package. During this process, access paths to the data are selected and some authorization checking is performed. The types of bind are:

automatic bind. (More correctly, *automatic rebind*) A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.

dynamic bind. A process by which SQL statements are bound as they are entered.

incremental bind. A process by which SQL statements are bound during the execution of an application process.

static bind. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time.

bit data. Data that is character type CHAR or
VARCHAR and has a CCSID value of 65535.

BLOB. Binary large object.

block fetch. A capability in which DB2 can retrieve, or fetch, a large set of rows together. Using block fetch can significantly reduce the number of messages that are being sent across the network. Block fetch applies only to cursors that do not update data.

BMP. Batch Message Processing (IMS). See *batch message processing program*.

bootstrap data set (BSDS). A VSAM data set that contains name and status information for DB2, as well as RBA range specifications, for all active and archive log data sets. It also contains passwords for the DB2 directory and catalog, and lists of conditional restart and checkpoint records.

BSAM. Basic sequential access method.

BSDS. Bootstrap data set.

buffer pool. Main storage that is reserved to satisfy the buffering requirements for one or more table spaces or indexes.

built-in data type. A data type that IBM supplies. Among the built-in data types for DB2 UDB for z/OS are string, numeric, ROWID, and datetime. Contrast with *distinct type*.

built-in function. A function that DB2 supplies. Contrast with *user-defined function*.

business dimension. A category of data, such as products or time periods, that an organization might want to analyze.

C

cache structure. A coupling facility structure that stores data that can be available to all members of a Sysplex. A DB2 data sharing group uses cache structures as group buffer pools.

CAF. Call attachment facility.

call attachment facility (CAF). A DB2 attachment facility for application programs that run in TSO or z/OS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment.

call-level interface (CLI). A callable application programming interface (API) for database access, which is an alternative to using embedded SQL. In contrast to embedded SQL, DB2 ODBC (which is based on the CLI architecture) does not require the user to precompile or bind applications, but instead provides a standard set of functions to process SQL statements and related services at run time.

cascade delete. The way in which DB2 enforces referential constraints when it deletes all descendent rows of a deleted parent row.

CASE expression. An expression that is selected based on the evaluation of one or more conditions.

cast function. A function that is used to convert instances of a (source) data type into instances of a different (target) data type. In general, a cast function has the name of the target data type. It has one single argument whose type is the source data type; its return type is the target data type.

castout. The DB2 process of writing changed pages from a group buffer pool to disk.

castout owner. The DB2 member that is responsible for casting out a particular page set or partition.

catalog. In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table. Any table in the DB2 catalog.

CCSID. Coded character set identifier.

CDB. Communications database.

CDRA. Character Data Representation Architecture.

CEC. Central electronic complex. See *central processor complex*.

central electronic complex (CEC). See *central processor complex*.

central processor (CP). The part of the computer that contains the sequencing and processing facilities for instruction execution, initial program load, and other machine operations.

central processor complex (CPC). A physical collection of hardware (such as an ES/3090™) that consists of main storage, one or more central processors, timers, and channels.

| **CFRM.** Coupling facility resource management.

CFRM policy. A declaration by a z/OS administrator regarding the allocation rules for a coupling facility structure.

character conversion. The process of changing characters from one encoding scheme to another.

Character Data Representation Architecture (CDRA). An architecture that is used to achieve consistent representation, processing, and interchange of string data.

character large object (CLOB). A sequence of bytes representing single-byte characters or a mixture of single- and double-byte characters where the size of the value can be up to 2 GB-1. In general, character large object values are used whenever a character string might exceed the limits of the VARCHAR type.

character set. A defined set of characters.

character string. A sequence of bytes that represent bit data, single-byte characters, or a mixture of single-byte and multibyte characters.

check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain.

check integrity. The condition that exists when each row in a table conforms to the check constraints that are defined on that table. Maintaining check integrity requires DB2 to enforce check constraints on operations that add or change data.

| **check pending.** A state of a table space or partition that prevents its use by some utilities and by some SQL statements because of rows that violate referential constraints, check constraints, or both.

checkpoint. A point at which DB2 records internal status information on the DB2 log; the recovery process uses this information if DB2 abnormally terminates.

| **child lock.** For explicit hierarchical locking, a lock that is held on either a table, page, row, or a large object (LOB). Each child lock has a parent lock. See also *parent lock*.

CI. Control interval.

| **CICS.** Represents (in this publication): CICS Transaction Server for z/OS: Customer Information Control System Transaction Server for z/OS.

CICS attachment facility. A DB2 subcomponent that uses the z/OS subsystem interface (SSI) and cross-storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

CIDE. Control interval definition field.

claim. A notification to DB2 that an object is being accessed. Claims prevent drains from occurring until the claim is released, which usually occurs at a commit point. Contrast with *drain*.

claim class. A specific type of object access that can be one of the following isolation levels:
Cursor stability (CS)
Repeatable read (RR)
Write

claim count. A count of the number of agents that are accessing an object.

class of service. A VTAM term for a list of routes through a network, arranged in an order of preference for their use.

class word. A single word that indicates the nature of a data attribute. For example, the class word PROJ indicates that the attribute identifies a project.

clause. In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

CLI. Call- level interface.

client. See *requester*.

CLIST. Command list. A language for performing TSO tasks.

CLOB. Character large object.

closed application. An application that requires exclusive use of certain statements on certain DB2

objects, so that the objects are managed solely through the application's external interface.

CLPA. Create link pack area.

| **clustering index.** An index that determines how rows
| are physically ordered (*clustered*) in a table space. If a
| clustering index on a partitioned table is not a
| partitioning index, the rows are ordered in cluster
| sequence within each data partition instead of spanning
| partitions. Prior to Version 8 of DB2 UDB for z/OS, the
| partitioning index was required to be the clustering
| index.

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

coded character set identifier (CCSID). A 16-bit number that uniquely identifies a coded representation of graphic characters. It designates an encoding scheme identifier and one or more pairs consisting of a character set identifier and an associated code page identifier.

code page. (1) A set of assignments of characters to code points. In EBCDIC, for example, the character 'A' is assigned code point X'C1' (2) , and character 'B' is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.

code point. In CDRA, a unique bit pattern that represents a character in a code page.

code unit. The fundamental binary width in a
computer architecture that is used for representing
character data, such as 7 bits, 8 bits, 16 bits, or 32 bits.
Depending on the character encoding form that is used,
each code point in a coded character set can be
represented internally by one or more code units.

coexistence. During migration, the period of time in which two releases exist in the same data sharing group.

cold start. A process by which DB2 restarts without processing any log records. Contrast with *warm start*.

collection. A group of packages that have the same qualifier.

column. The vertical component of a table. A column has a name and a particular data type (for example, character, decimal, or integer).

column function. See *aggregate function*.

"come from" checking. An LU 6.2 security option that defines a list of authorization IDs that are allowed to connect to DB2 from a partner LU.

command. A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.

command prefix. A one- to eight-character command identifier. The command prefix distinguishes the command as belonging to an application or subsystem rather than to MVS.

command recognition character (CRC). A character that permits a z/OS console operator or an IMS subsystem user to route DB2 commands to specific DB2 subsystems.

command scope. The scope of command operation in a data sharing group. If a command has *member scope*, the command displays information only from the one member or affects only non-shared resources that are owned locally by that member. If a command has *group scope*, the command displays information from all members, affects non-shared resources that are owned locally by all members, displays information on sharable resources, or affects sharable resources.

commit. The operation that ends a unit of work by releasing locks so that the database changes that are made by that unit of work can be perceived by other processes.

commit point. A point in time when data is considered consistent.

committed phase. The second phase of the multisite update process that requests all participants to commit the effects of the logical unit of work.

common service area (CSA). In z/OS, a part of the common area that contains data areas that are addressable by all address spaces.

communications database (CDB). A set of tables in the DB2 catalog that are used to establish conversations with remote database management systems.

comparison operator. A token (such as =, >, or <) that is used to specify a relationship between two values.

composite key. An ordered set of key columns of the same table.

compression dictionary. The dictionary that controls the process of compression and decompression. This dictionary is created from the data in the table space or table space partition.

concurrency. The shared use of resources by more than one application process at the same time.

conditional restart. A DB2 restart that is directed by a user-defined conditional restart control record (CRCR).

connection. In SNA, the existence of a communication path between two partner LUs that allows information

to be exchanged (for example, two DB2 subsystems that are connected and communicating by way of a conversation).

connection context. In SQLJ, a Java object that represents a connection to a data source.

connection declaration clause. In SQLJ, a statement that declares a connection to a data source.

connection handle. The data object containing information that is associated with a connection that DB2 ODBC manages. This includes general status information, transaction status, and diagnostic information.

connection ID. An identifier that is supplied by the attachment facility and that is associated with a specific address space connection.

consistency token. A timestamp that is used to generate the version identifier for an application. See also *version*.

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with *variable*.

constraint. A rule that limits the values that can be inserted, deleted, or updated in a table. See *referential constraint*, *check constraint*, and *unique constraint*.

context. The application's logical connection to the data source and associated internal DB2 ODBC connection information that allows the application to direct its operations to a data source. A DB2 ODBC context represents a DB2 thread.

contracting conversion. A process that occurs when the length of a converted string is smaller than that of the source string. For example, this process occurs when an EBCDIC mixed-data string that contains DBCS characters is converted to ASCII mixed data; the converted string is shorter because of the removal of the shift codes.

control interval (CI). A fixed-length area or disk in which VSAM stores records and creates distributed free space. Also, in a key-sequenced data set or file, the set of records that an entry in the sequence-set index record points to. The control interval is the unit of information that VSAM transmits to or from disk. A control interval always includes an integral number of physical records.

control interval definition field (CIDF). In VSAM, a field that is located in the 4 bytes at the end of each control interval; it describes the free space, if any, in the control interval.

conversation. Communication, which is based on LU 6.2 or Advanced Program-to-Program Communication (APPC), between an application and a remote

transaction program over an SNA logical unit-to-logical unit (LU-LU) session that allows communication while processing a transaction.

coordinator. The system component that coordinates the commit or rollback of a unit of work that includes work that is done on one or more other systems.

| **copy pool.** A named set of SMS storage groups that
| contains data that is to be copied collectively. A copy
| pool is an SMS construct that lets you define which
| storage groups are to be copied by using FlashCopy®
| functions. HSM determines which volumes belong to a
| copy pool.

| **copy target.** A named set of SMS storage groups that
| are to be used as containers for copy pool volume
| copies. A copy target is an SMS construct that lets you
| define which storage groups are to be used as
| containers for volumes that are copied by using
| FlashCopy functions.

| **copy version.** A point-in-time FlashCopy copy that is
| managed by HSM. Each copy pool has a version
| parameter that specifies how many copy versions are
| maintained on disk.

correlated columns. A relationship between the value of one column and the value of another column.

correlated subquery. A subquery (part of a WHERE or HAVING clause) that is applied to a row or group of rows of a table or view that is named in an outer subselect statement.

correlation ID. An identifier that is associated with a specific thread. In TSO, it is either an authorization ID or the job name.

correlation name. An identifier that designates a table, a view, or individual rows of a table or view within a single SQL statement. It can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

cost category. A category into which DB2 places cost estimates for SQL statements at the time the statement is bound. A cost estimate can be placed in either of the following cost categories:

- A: Indicates that DB2 had enough information to make a cost estimate without using default values.
- B: Indicates that some condition exists for which DB2 was forced to use default values for its estimate.

The cost category is externalized in the COST_CATEGORY column of the DSN_STATEMENT_TABLE when a statement is explained.

coupling facility. A special PR/SM™ LPAR logical partition that runs the coupling facility control program and provides high-speed caching, list processing, and locking functions in a Parallel Sysplex®.

| **coupling facility resource management.** A component of z/OS that provides the services to manage coupling facility resources in a Parallel Sysplex. This management includes the enforcement of CFRM policies to ensure that the coupling facility and structure requirements are satisfied.

CP. Central processor.

CPC. Central processor complex.

C++ member. A data object or function in a structure, union, or class.

C++ member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and to the member functions of objects in its class. Member functions are also called methods.

C++ object. (1) A region of storage. An object is created when a variable is defined or a new function is invoked. (2) An instance of a class.

CRC. Command recognition character.

CRCR. Conditional restart control record. See also *conditional restart*.

create link pack area (CLPA). An option that is used during IPL to initialize the link pack pageable area.

created temporary table. A table that holds temporary data and is defined with the SQL statement CREATE GLOBAL TEMPORARY TABLE. Information about created temporary tables is stored in the DB2 catalog, so this kind of table is persistent and can be shared across application processes. Contrast with *declared temporary table*. See also *temporary table*.

cross-memory linkage. A method for invoking a program in a different address space. The invocation is synchronous with respect to the caller.

cross-system coupling facility (XCF). A component of z/OS that provides functions to support cooperation between authorized programs that run within a Sysplex.

cross-system extended services (XES). A set of z/OS services that allow multiple instances of an application or subsystem, running on different systems in a Sysplex environment, to implement high-performance, high-availability data sharing by using a coupling facility.

CS. Cursor stability.

CSA. Common service area.

CT. Cursor table.

current data. Data within a host structure that is current with (identical to) the data within the base table.

current SQL ID. An ID that, at a single point in time, holds the privileges that are exercised when certain dynamic SQL statements run. The current SQL ID can be a primary authorization ID or a secondary authorization ID.

current status rebuild. The second phase of restart processing during which the status of the subsystem is reconstructed from information on the log.

cursor. A named control structure that an application program uses to point to a single row or multiple rows within some ordered set of rows of a result table. A cursor can be used to retrieve, update, or delete rows from a result table.

cursor sensitivity. The degree to which database updates are visible to the subsequent FETCH statements in a cursor. A cursor can be sensitive to changes that are made with positioned update and delete statements specifying the name of that cursor. A cursor can also be sensitive to changes that are made with searched update or delete statements, or with cursors other than this cursor. These changes can be made by this application process or by another application process.

cursor stability (CS). The isolation level that provides maximum concurrency without the ability to read uncommitted data. With cursor stability, a unit of work holds locks only on its uncommitted changes and on the current row of each of its cursors.

cursor table (CT). The copy of the skeleton cursor table that is used by an executing application process.

cycle. A set of tables that can be ordered so that each table is a descendent of the one before it, and the first table is a descendent of the last table. A self-referencing table is a cycle with a single member.

D

| **DAD.** See *Document access definition*.

| **disk.** A direct-access storage device that records data magnetically.

database. A collection of tables, or a collection of table spaces and index spaces.

database access thread. A thread that accesses data at the local subsystem on behalf of a remote subsystem.

database administrator (DBA). An individual who is responsible for designing, developing, operating, safeguarding, maintaining, and using a database.

| **database alias.** The name of the target server if
| different from the location name. The database alias
| name is used to provide the name of the database
| server as it is known to the network. When a database
| alias name is defined, the location name is used by the
| application to reference the server, but the database
| alias name is used to identify the database server to be
| accessed. Any fully qualified object names within any
| SQL statements are not modified and are sent
| unchanged to the database server.

| **database descriptor (DBD).** An internal representation
| of a DB2 database definition, which reflects the data
| definition that is in the DB2 catalog. The objects that
| are defined in a database descriptor are table spaces,
| tables, indexes, index spaces, relationships, check
| constraints, and triggers. A DBD also contains
| information about accessing tables in the database.

database exception status. An indication that
something is wrong with a database. All members of a
data sharing group must know and share the exception
status of databases.

| **database identifier (DBID).** An internal identifier of
| the database.

database management system (DBMS). A software
system that controls the creation, organization, and
modification of a database and the access to the data
that is stored within it.

database request module (DBRM). A data set
member that is created by the DB2 precompiler and
that contains information about SQL statements.
DBRMs are used in the bind process.

database server. The target of a request from a local
application or an intermediate database server. In the
DB2 environment, the database server function is
provided by the distributed data facility to access DB2
data from local applications, or from a remote database
server that acts as an intermediate database server.

data currency. The state in which data that is retrieved
into a host variable in your program is a copy of data
in the base table.

data definition name (ddname). The name of a data
definition (DD) statement that corresponds to a data
control block containing the same name.

data dictionary. A repository of information about an
organization's application programs, databases, logical
data models, users, and authorizations. A data
dictionary can be manual or automated.

data-driven business rules. Constraints on particular
data values that exist as a result of requirements of the
business.

Data Language/I (DL/I). The IMS data manipulation
language; a common high-level interface between a
user application and IMS.

data mart. A small data warehouse that applies to a
single department or team. See also *data warehouse*.

data mining. The process of collecting critical business
information from a data warehouse, correlating it, and
uncovering associations, patterns, and trends.

data partition. A VSAM data set that is contained
within a partitioned table space.

data-partitioned secondary index (DPSI). A secondary
index that is partitioned. The index is partitioned
according to the underlying data.

data sharing. The ability of two or more DB2
subsystems to directly access and change a single set of
data.

data sharing group. A collection of one or more DB2
subsystems that directly access and change the same
data while maintaining data integrity.

data sharing member. A DB2 subsystem that is
assigned by XCF services to a data sharing group.

data source. A local or remote relational or
non-relational data manager that is capable of
supporting data access via an ODBC driver that
supports the ODBC APIs. In the case of DB2 UDB for
z/OS, the data sources are always relational database
managers.

| **data space.** In releases prior to DB2 UDB for z/OS,
| Version 8, a range of up to 2 GB of contiguous virtual
| storage addresses that a program can directly
| manipulate. Unlike an address space, a data space can
| hold only data; it does not contain common areas,
| system data, or programs.

data type. An attribute of columns, literals, host
variables, special registers, and the results of functions
and expressions.

data warehouse. A system that provides critical
business information to an organization. The data
warehouse system cleanses the data for accuracy and
currency, and then presents the data to decision makers
so that they can interpret and use it effectively and
efficiently.

date. A three-part value that designates a day, month,
and year.

date duration. A decimal integer that represents a
number of years, months, and days.

datetime value. A value of the data type DATE, TIME,
or TIMESTAMP.

DBA. Database administrator.

DBCLOB. Double-byte character large object.

DBCS. Double-byte character set.

DBD. Database descriptor.

DBID. Database identifier.

DBMS. Database management system.

DBRM. Database request module.

DB2 catalog. Tables that are maintained by DB2 and contain descriptions of DB2 objects, such as tables, views, and indexes.

DB2 command. An instruction to the DB2 subsystem that a user enters to start or stop DB2, to display information on current users, to start or stop databases, to display information on the status of databases, and so on.

DB2 for VSE & VM. The IBM DB2 relational database management system for the VSE and VM operating systems.

DB2I. DB2 Interactive.

DB2 Interactive (DB2I). The DB2 facility that provides for the execution of SQL statements, DB2 (operator) commands, programmer commands, and utility invocation.

DB2I Kanji Feature. The tape that contains the panels and jobs that allow a site to display DB2I panels in Kanji.

DB2 PM. DB2 Performance Monitor.

DB2 thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services.

DCLGEN. Declarations generator.

DDF. Distributed data facility.

ddname. Data definition name.

deadlock. Unresolvable contention for the use of a resource, such as a table or an index.

declarations generator (DCLGEN). A subcomponent of DB2 that generates SQL table declarations and COBOL, C, or PL/I data structure declarations that conform to the table. The declarations are generated from DB2 system catalog information. DCLGEN is also a DSN subcommand.

declared temporary table. A table that holds temporary data and is defined with the SQL statement DECLARE GLOBAL TEMPORARY TABLE. Information about declared temporary tables is not stored in the DB2 catalog, so this kind of table is not persistent and

can be used only by the application process that issued the DECLARE statement. Contrast with *created temporary table*. See also *temporary table*.

default value. A predetermined value, attribute, or option that is assumed when no other is explicitly specified.

deferred embedded SQL. SQL statements that are neither fully static nor fully dynamic. Like static statements, they are embedded within an application, but like dynamic statements, they are prepared during the execution of the application.

deferred write. The process of asynchronously writing changed data pages to disk.

degree of parallelism. The number of concurrently executed operations that are initiated to process a query.

delete-connected. A table that is a dependent of table P or a dependent of a table to which delete operations from table P cascade.

delete hole. The location on which a cursor is positioned when a row in a result table is refetched and the row no longer exists on the base table, because another cursor deleted the row between the time the cursor first included the row in the result table and the time the cursor tried to refetch it.

delete rule. The rule that tells DB2 what to do to a dependent row when a parent row is deleted. For each relationship, the rule might be CASCADE, RESTRICT, SET NULL, or NO ACTION.

delete trigger. A trigger that is defined with the triggering SQL operation DELETE.

delimited identifier. A sequence of characters that are enclosed within double quotation marks ("). The sequence must consist of a letter followed by zero or more characters, each of which is a letter, digit, or the underscore character (_).

delimiter token. A string constant, a delimited identifier, an operator symbol, or any of the special characters that are shown in DB2 syntax diagrams.

denormalization. A key step in the task of building a physical relational database design. Denormalization is the intentional duplication of columns in multiple tables, and the consequence is increased data redundancy. Denormalization is sometimes necessary to minimize performance problems. Contrast with *normalization*.

dependent. An object (row, table, or table space) that has at least one parent. The object is also said to be a dependent (row, table, or table space) of its parent. See also *parent row*, *parent table*, *parent table space*.

dependent row. A row that contains a foreign key that matches the value of a primary key in the parent row.

dependent table. A table that is a dependent in at least one referential constraint.

DES-based authenticator. An authenticator that is generated using the DES algorithm.

descendent. An object that is a dependent of an object or is the dependent of a descendent of an object.

descendent row. A row that is dependent on another row, or a row that is a descendent of a dependent row.

descendent table. A table that is a dependent of another table, or a table that is a descendent of a dependent table.

deterministic function. A user-defined function whose result is dependent on the values of the input arguments. That is, successive invocations with the same input values produce the same answer. Sometimes referred to as a *not-variant* function. Contrast this with an *nondeterministic function* (sometimes called a *variant function*), which might not always produce the same result for the same inputs.

DFP. Data Facility Product (in z/OS).

DFSMS. Data Facility Storage Management Subsystem (in z/OS). Also called *Storage Management Subsystem (SMS)*.

| **DFSMSdss™.** The data set services (dss) component of
| DFSMS (in z/OS).

| **DFSMSHsm™.** The hierarchical storage manager (hsm)
| component of DFSMS (in z/OS).

dimension. A data category such as time, products, or markets. The elements of a dimension are referred to as members. Dimensions offer a very concise, intuitive way of organizing and selecting data for retrieval, exploration, and analysis. See also *dimension table*.

dimension table. The representation of a dimension in a star schema. Each row in a dimension table represents all of the attributes for a particular member of the dimension. See also *dimension*, *star schema*, and *star join*.

directory. The DB2 system database that contains internal objects such as database descriptors and skeleton cursor tables.

distinct predicate. In SQL, a predicate that ensures
that two row values are not equal, and that both row
values are not null.

distinct type. A user-defined data type that is internally represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

distributed data. Data that resides on a DBMS other than the local system.

distributed data facility (DDF). A set of DB2 components through which DB2 communicates with another relational database management system.

Distributed Relational Database Architecture (DRDA). A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems. See also *DRDA access*.

DL/I. Data Language/I.

DNS. Domain name server.

| **document access definition (DAD).** Used to define
| the indexing scheme for an XML column or the
| mapping scheme of an XML collection. It can be used
| to enable an XML Extender column of an XML
| collection, which is XML formatted.

domain. The set of valid values for an attribute.

domain name. The name by which TCP/IP applications refer to a TCP/IP host within a TCP/IP network.

domain name server (DNS). A special TCP/IP network server that manages a distributed directory that is used to map TCP/IP host names to IP addresses.

double-byte character large object (DBCLOB). A sequence of bytes representing double-byte characters where the size of the values can be up to 2 GB. In general, DBCLOB values are used whenever a double-byte character string might exceed the limits of the VARCHAR type.

double-byte character set (DBCS). A set of characters, which are used by national languages such as Japanese and Chinese, that have more symbols than can be represented by a single byte. Each character is 2 bytes in length. Contrast with *single-byte character set* and *multibyte character set*.

double-precision floating point number. A 64-bit approximate representation of a real number.

downstream. The set of nodes in the syncpoint tree that is connected to the local DBMS as a participant in the execution of a two-phase commit.

| **DPSI.** Data-partitioned secondary index.

drain. The act of acquiring a locked resource by quiescing access to that object.

drain lock. A lock on a claim class that prevents a claim from occurring.

DRDA. Distributed Relational Database Architecture.

DRDA access. An open method of accessing distributed data that you can use to connect to another database server to execute packages that were previously bound at the server location. You use the SQL CONNECT statement or an SQL statement with a three-part name to identify the server. Contrast with *private protocol access*.

DSN. (1) The default DB2 subsystem name. (2) The name of the TSO command processor of DB2. (3) The first three characters of DB2 module and macro names.

duration. A number that represents an interval of time. See also *date duration*, *labeled duration*, and *time duration*.

| **dynamic cursor.** A named control structure that an application program uses to change the size of the result table and the order of its rows after the cursor is opened. Contrast with *static cursor*.

dynamic dump. A dump that is issued during the execution of a program, usually under the control of that program.

dynamic SQL. SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

| **dynamic statement cache pool.** A cache, located above the 2-GB storage line, that holds dynamic statements.

E

EA-enabled table space. A table space or index space that is enabled for extended addressability and that contains individual partitions (or pieces, for LOB table spaces) that are greater than 4 GB.

| **EB.** See *exabyte*.

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the z/OS, VM, VSE, and iSeries™ environments. Contrast with *ASCII* and *Unicode*.

e-business. The transformation of key business processes through the use of Internet technologies.

| **EDM pool.** A pool of main storage that is used for database descriptors, application plans, authorization cache, application packages.

EID. Event identifier.

embedded SQL. SQL statements that are coded within an application program. See *static SQL*.

enclave. In Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is similar to a program or run unit.

encoding scheme. A set of rules to represent character data (ASCII, EBCDIC, or Unicode).

entity. A significant object of interest to an organization.

enumerated list. A set of DB2 objects that are defined with a LISTDEF utility control statement in which pattern-matching characters (*, %, _ or ?) are not used.

environment. A collection of names of logical and physical resources that are used to support the performance of a function.

environment handle. In DB2 ODBC, the data object that contains global information regarding the state of the application. An environment handle must be allocated before a connection handle can be allocated. Only one environment handle can be allocated per application.

EOM. End of memory.

EOT. End of task.

equijoin. A join operation in which the join-condition has the form *expression = expression*.

error page range. A range of pages that are considered to be physically damaged. DB2 does not allow users to access any pages that fall within this range.

escape character. The symbol that is used to enclose an SQL delimited identifier. The escape character is the double quotation mark ("), except in COBOL applications, where the user assigns the symbol, which is either a double quotation mark or an apostrophe (').

ESDS. Entry sequenced data set.

ESMT. External subsystem module table (in IMS).

EUR. IBM European Standards.

| **exabyte.** For processor, real and virtual storage capacities and channel volume:
| 1 152 921 504 606 846 976 bytes or 2⁶⁰.

exception table. A table that holds rows that violate referential constraints or check constraints that the CHECK DATA utility finds.

exclusive lock. A lock that prevents concurrently executing application processes from reading or changing data. Contrast with *share lock*.

executable statement. An SQL statement that can be embedded in an application program, dynamically prepared and executed, or issued interactively.

execution context. In SQLJ, a Java object that can be used to control the execution of SQL statements.

exit routine. A user-written (or IBM-provided default) program that receives control from DB2 to perform specific functions. Exit routines run as extensions of DB2.

expanding conversion. A process that occurs when the length of a converted string is greater than that of the source string. For example, this process occurs when an ASCII mixed-data string that contains DBCS characters is converted to an EBCDIC mixed-data string; the converted string is longer because of the addition of shift codes.

explicit hierarchical locking. Locking that is used to make the parent-child relationship between resources known to IRLM. This kind of locking avoids global locking overhead when no inter-DB2 interest exists on a resource.

exposed name. A correlation name or a table or view name for which a correlation name is not specified. Names that are specified in a FROM clause are exposed or non-exposed.

expression. An operand or a collection of operators and operands that yields a single value.

extended recovery facility (XRF). A facility that minimizes the effect of failures in z/OS, VTAM, the host processor, or high-availability applications during sessions between high-availability applications and designated terminals. This facility provides an alternative subsystem to take over sessions from the failing subsystem.

Extensible Markup Language (XML). A standard metalanguage for defining markup languages that is a subset of Standardized General Markup Language (SGML). The less complex nature of XML makes it easier to write applications that handle document types, to author and manage structured information, and to transmit and share structured information across diverse computing environments.

external function. A function for which the body is written in a programming language that takes scalar argument values and produces a scalar result for each invocation. Contrast with *sourced function*, *built-in function*, and *SQL function*.

external procedure. A user-written application program that can be invoked with the SQL CALL statement, which is written in a programming language. Contrast with *SQL procedure*.

external routine. A user-defined function or stored procedure that is based on code that is written in an external programming language.

external subsystem module table (ESMT). In IMS, the table that specifies which attachment modules must be loaded.

F

failed member state. A state of a member of a data sharing group. When a member fails, the XCF permanently records the failed member state. This state usually means that the member's task, address space, or z/OS system terminated before the state changed from active to quiesced.

fallback. The process of returning to a previous release of DB2 after attempting or completing migration to a current release.

false global lock contention. A contention indication from the coupling facility when multiple lock names are hashed to the same indicator and when no real contention exists.

fan set. A direct physical access path to data, which is provided by an index, hash, or link; a fan set is the means by which the data manager supports the ordering of data.

federated database. The combination of a DB2 Universal Database server (in Linux, UNIX, and Windows environments) and multiple data sources to which the server sends queries. In a federated database system, a client application can use a single SQL statement to join data that is distributed across multiple database management systems and can view the data as if it were local.

fetch orientation. The specification of the desired placement of the cursor as part of a FETCH statement (for example, BEFORE, AFTER, NEXT, PRIOR, CURRENT, FIRST, LAST, ABSOLUTE, and RELATIVE).

field procedure. A user-written exit routine that is designed to receive a single value and transform (encode or decode) it in any way the user can specify.

filter factor. A number between zero and one that estimates the proportion of rows in a table for which a predicate is true.

fixed-length string. A character or graphic string whose length is specified and cannot be changed. Contrast with *varying-length string*.

FlashCopy. A function on the IBM Enterprise Storage Server® that can create a point-in-time copy of data while an application is running.

foreign key. A column or set of columns in a dependent table of a constraint relationship. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table.

Each foreign key value must either match a parent key value in the related parent table or be null.

| **forest.** An ordered set of subtrees of XML nodes.

forget. In a two-phase commit operation, (1) the vote that is sent to the prepare phase when the participant has not modified any data. The forget vote allows a participant to release locks and forget about the logical unit of work. This is also referred to as the read-only vote. (2) The response to the *committed* request in the second phase of the operation.

forward log recovery. The third phase of restart processing during which DB2 processes the log in a forward direction to apply all REDO log records.

free space. The total amount of unused space in a page; that is, the space that is not used to store records or control information is free space.

full outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of both tables. See also *join*.

fullselect. A subselect, a values-clause, or a number of both that are combined by set operators. *Fullselect* specifies a result table. If UNION is not used, the result of the fullselect is the result of the specified subselect.

| **fully escaped mapping.** A mapping from an SQL
| identifier to an XML name when the SQL identifier is a
| column name.

function. A mapping, which is embodied as a
program (the function body) that is invocable by means
of zero or more input values (arguments) to a single
value (the result). See also *aggregate function* and *scalar
function*.
Functions can be user-defined, built-in, or generated by
DB2. (See also *built-in function*, *cast function*, *external
function*, *sourced function*, *SQL function*, and *user-defined
function*.)

function definer. The authorization ID of the owner of the schema of the function that is specified in the CREATE FUNCTION statement.

function implementer. The authorization ID of the owner of the function program and function package.

function package. A package that results from binding the DBRM for a function program.

function package owner. The authorization ID of the user who binds the function program's DBRM into a function package.

function resolution. The process, internal to the DBMS, by which a function invocation is bound to a particular function instance. This process uses the function name, the data types of the arguments, and a

list of the applicable schema names (called the *SQL path*) to make the selection. This process is sometimes called *function selection*.

function selection. See *function resolution*.

function signature. The logical concatenation of a fully qualified function name with the data types of all of its parameters.

G

GB. Gigabyte (1 073 741 824 bytes).

GBP. Group buffer pool.

GBP-dependent. The status of a page set or page set partition that is dependent on the group buffer pool. Either read/write interest is active among DB2 subsystems for this page set, or the page set has changed pages in the group buffer pool that have not yet been cast out to disk.

generalized trace facility (GTF). A z/OS service program that records significant system events such as I/O interrupts, SVC interrupts, program interrupts, or external interrupts.

generic resource name. A name that VTAM uses to represent several application programs that provide the same function in order to handle session distribution and balancing in a Sysplex environment.

getpage. An operation in which DB2 accesses a data page.

global lock. A lock that provides concurrency control within and among DB2 subsystems. The scope of the lock is across all DB2 subsystems of a data sharing group.

global lock contention. Conflicts on locking requests between different DB2 members of a data sharing group when those members are trying to serialize shared resources.

governor. See *resource limit facility*.

graphic string. A sequence of DBCS characters.

gross lock. The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group buffer pool (GBP). A coupling facility cache structure that is used by a data sharing group to cache data and to ensure that the data is consistent for all members.

group buffer pool duplexing. The ability to write data to two instances of a group buffer pool structure: a *primary group buffer pool* and a *secondary group buffer*

pool. z/OS publications refer to these instances as the "old" (for primary) and "new" (for secondary) structures.

group level. The release level of a data sharing group, which is established when the first member migrates to a new release.

group name. The z/OS XCF identifier for a data sharing group.

group restart. A restart of at least one member of a data sharing group after the loss of either locks or the shared communications area.

GTF. Generalized trace facility.

H

handle. In DB2 ODBC, a variable that refers to a data structure and associated resources. See also *statement handle*, *connection handle*, and *environment handle*.

help panel. A screen of information that presents tutorial text to assist a user at the workstation or terminal.

heuristic damage. The inconsistency in data between one or more participants that results when a heuristic decision to resolve an indoubt LUW at one or more participants differs from the decision that is recorded at the coordinator.

heuristic decision. A decision that forces indoubt resolution at a participant by means other than automatic resynchronization between coordinator and participant.

| **hole.** A row of the result table that cannot be accessed
| because of a delete or an update that has been
| performed on the row. See also *delete hole* and *update hole*.

home address space. The area of storage that z/OS currently recognizes as *dispatched*.

host. The set of programs and resources that are available on a given TCP/IP instance.

host expression. A Java variable or expression that is referenced by SQL clauses in an SQLJ application program.

host identifier. A name that is declared in the host program.

host language. A programming language in which you can embed SQL statements.

host program. An application program that is written in a host language and that contains embedded SQL statements.

host structure. In an application program, a structure that is referenced by embedded SQL statements.

host variable. In an application program, an application variable that is referenced by embedded SQL statements.

| **host variable array.** An array of elements, each of
| which corresponds to a value for a column. The
| dimension of the array determines the maximum
| number of rows for which the array can be used.

HSM. Hierarchical storage manager.

HTML. Hypertext Markup Language, a standard method for presenting Web data to users.

HTTP. Hypertext Transfer Protocol, a communication protocol that the Web uses.

I

ICF. Integrated catalog facility.

IDCAMS. An IBM program that is used to process access method services commands. It can be invoked as a job or jobstep, from a TSO terminal, or from within a user's application program.

IDCAMS LISTCAT. A facility for obtaining information that is contained in the access method services catalog.

identify. A request that an attachment service program in an address space that is separate from DB2 issues thorough the z/OS subsystem interface to inform DB2 of its existence and to initiate the process of becoming connected to DB2.

identity column. A column that provides a way for DB2 to automatically generate a numeric value for each row. The generated values are unique if cycling is not used. Identity columns are defined with the AS IDENTITY clause. Uniqueness of values can be ensured by defining a unique index that contains only the identity column. A table can have no more than one identity column.

IFCID. Instrumentation facility component identifier.

IFI. Instrumentation facility interface.

IFI call. An invocation of the instrumentation facility interface (IFI) by means of one of its defined functions.

IFP. IMS Fast Path.

image copy. An exact reproduction of all or part of a table space. DB2 provides utility programs to make full image copies (to copy the entire table space) or incremental image copies (to copy only those pages that have been modified since the last image copy).

implied forget. In the presumed-abort protocol, an implied response of *forget* to the second-phase *committed* request from the coordinator. The response is implied when the participant responds to any subsequent request from the coordinator.

IMS. Information Management System.

IMS attachment facility. A DB2 subcomponent that uses z/OS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

IMS DB. Information Management System Database.

IMS TM. Information Management System Transaction Manager.

in-abort. A status of a unit of recovery. If DB2 fails after a unit of recovery begins to be rolled back, but before the process is completed, DB2 continues to back out the changes during restart.

in-commit. A status of a unit of recovery. If DB2 fails after beginning its phase 2 commit processing, it "knows," when restarted, that changes made to data are consistent. Such units of recovery are termed *in-commit*.

independent. An object (row, table, or table space) that is neither a parent nor a dependent of another object.

index. A set of pointers that are logically ordered by the values of a key. Indexes can provide faster access to data and can enforce uniqueness on the rows in a table.

| **index-controlled partitioning.** A type of partitioning
| in which partition boundaries for a partitioned table are
| controlled by values that are specified on the CREATE
| INDEX statement. Partition limits are saved in the
| LIMITKEY column of the SYSIBM.SYSINDEXPART
| catalog table.

index key. The set of columns in a table that is used to determine the order of index entries.

index partition. A VSAM data set that is contained within a partitioning index space.

index space. A page set that is used to store the entries of one index.

indicator column. A 4-byte value that is stored in a base table in place of a LOB column.

indicator variable. A variable that is used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

indoubt. A status of a unit of recovery. If DB2 fails after it has finished its phase 1 commit processing and before it has started phase 2, only the commit coordinator knows if an individual unit of recovery is

to be committed or rolled back. At emergency restart, if DB2 lacks the information it needs to make this decision, the status of the unit of recovery is *indoubt* until DB2 obtains this information from the coordinator. More than one unit of recovery can be *indoubt* at restart.

indoubt resolution. The process of resolving the status of an *indoubt* logical unit of work to either the committed or the rollback state.

inflight. A status of a unit of recovery. If DB2 fails before its unit of recovery completes phase 1 of the commit process, it merely backs out the updates of its unit of recovery at restart. These units of recovery are termed *inflight*.

inheritance. The passing downstream of class resources or attributes from a parent class in the class hierarchy to a child class.

initialization file. For DB2 ODBC applications, a file containing values that can be set to adjust the performance of the database manager.

inline copy. A copy that is produced by the LOAD or REORG utility. The data set that the inline copy produces is logically equivalent to a full image copy that is produced by running the COPY utility with read-only access (SHRLEVEL REFERENCE).

inner join. The result of a join operation that includes only the matched rows of both tables that are being joined. See also *join*.

inoperative package. A package that cannot be used because one or more user-defined functions or procedures that the package depends on were dropped. Such a package must be explicitly rebound. Contrast with *invalid package*.

| **insensitive cursor.** A cursor that is not sensitive to
| inserts, updates, or deletes that are made to the
| underlying rows of a result table after the result table
| has been materialized.

insert trigger. A trigger that is defined with the triggering SQL operation INSERT.

install. The process of preparing a DB2 subsystem to operate as a z/OS subsystem.

installation verification scenario. A sequence of operations that exercises the main DB2 functions and tests whether DB2 was correctly installed.

instrumentation facility component identifier (IFCID). A value that names and identifies a trace record of an event that can be traced. As a parameter on the START TRACE and MODIFY TRACE commands, it specifies that the corresponding event is to be traced.

instrumentation facility interface (IFI). A programming interface that enables programs to obtain online trace data about DB2, to submit DB2 commands, and to pass data to DB2.

Interactive System Productivity Facility (ISPF). An IBM licensed program that provides interactive dialog services in a z/OS environment.

inter-DB2 R/W interest. A property of data in a table space, index, or partition that has been opened by more than one member of a data sharing group and that has been opened for writing by at least one of those members.

intermediate database server. The target of a request from a local application or a remote application requester that is forwarded to another database server. In the DB2 environment, the remote request is forwarded transparently to another database server if the object that is referenced by a three-part name does not reference the local location.

internationalization. The support for an encoding scheme that is able to represent the code points of characters from many different geographies and languages. To support all geographies, the Unicode standard requires more than 1 byte to represent a single character. See also *Unicode*.

internal resource lock manager (IRLM). A z/OS subsystem that DB2 uses to control communication and database locking.

| **International Organization for Standardization.** An international body charged with creating standards to facilitate the exchange of goods and services as well as cooperation in intellectual, scientific, technological, and economic activity.

invalid package. A package that depends on an object (other than a user-defined function) that is dropped. Such a package is implicitly rebound on invocation. Contrast with *inoperative package*.

invariant character set. (1) A character set, such as the syntactic character set, whose code point assignments do not change from code page to code page. (2) A minimum set of characters that is available as part of all character sets.

IP address. A 4-byte value that uniquely identifies a TCP/IP host.

IRLM. Internal resource lock manager.

ISO. International Organization for Standardization.

isolation level. The degree to which a unit of work is isolated from the updating operations of other units of work. See also *cursor stability*, *read stability*, *repeatable read*, and *uncommitted read*.

ISPF. Interactive System Productivity Facility.

ISPF/PDF. Interactive System Productivity Facility/Program Development Facility.

iterator. In SQLJ, an object that contains the result set of a query. An iterator is equivalent to a cursor in other host languages.

iterator declaration clause. In SQLJ, a statement that generates an iterator declaration class. An iterator is an object of an iterator declaration class.

J

| **Japanese Industrial Standard.** An encoding scheme that is used to process Japanese characters.

| **JAR.** Java Archive.

Java Archive (JAR). A file format that is used for aggregating many files into a single file.

JCL. Job control language.

JDBC. A Sun Microsystems database application programming interface (API) for Java that allows programs to access database management systems by using callable SQL. JDBC does not require the use of an SQL preprocessor. In addition, JDBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time.

JES. Job Entry Subsystem.

JIS. Japanese Industrial Standard.

job control language (JCL). A control language that is used to identify a job to an operating system and to describe the job's requirements.

Job Entry Subsystem (JES). An IBM licensed program that receives jobs into the system and processes all output data that is produced by the jobs.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values. See also *equijoin*, *full outer join*, *inner join*, *left outer join*, *outer join*, and *right outer join*.

K

KB. Kilobyte (1024 bytes).

Kerberos. A network authentication protocol that is designed to provide strong authentication for client/server applications by using secret-key cryptography.

Kerberos ticket. A transparent application mechanism that transmits the identity of an initiating principal to its target. A simple ticket contains the principal's

identity, a session key, a timestamp, and other information, which is sealed using the target's secret key.

key. A column or an ordered collection of columns that is identified in the description of a table, index, or referential constraint. The same column can be part of more than one key.

key-sequenced data set (KSDS). A VSAM file or data set whose records are loaded in key sequence and controlled by an index.

keyword. In SQL, a name that identifies an option that is used in an SQL statement.

KSDS. Key-sequenced data set.

L

labeled duration. A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

large object (LOB). A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single- and double-byte characters. A LOB can be up to 2 GB–1 byte in length. See also *BLOB*, *CLOB*, and *DBCLOB*.

last agent optimization. An optimized commit flow for either presumed-nothing or presumed-abort protocols in which the last agent, or final participant, becomes the commit coordinator. This flow saves at least one message.

latch. A DB2 internal mechanism for controlling concurrent events or the use of system resources.

LCID. Log control interval definition.

LDS. Linear data set.

leaf page. A page that contains pairs of keys and RIDs and that points to actual data. Contrast with *nonleaf page*.

left outer join. The result of a join operation that includes the matched rows of both tables that are being joined, and that preserves the unmatched rows of the first table. See also *join*.

limit key. The highest value of the index key for a partition.

linear data set (LDS). A VSAM data set that contains data but no control information. A linear data set can be accessed as a byte-addressable string in virtual storage.

linkage editor. A computer program for creating load modules from one or more object modules or load

modules by resolving cross references among the modules and, if necessary, adjusting addresses.

link-edit. The action of creating a loadable computer program using a linkage editor.

list. A type of object, which DB2 utilities can process, that identifies multiple table spaces, multiple index spaces, or both. A list is defined with the LISTDEF utility control statement.

list structure. A coupling facility structure that lets data be shared and manipulated as elements of a queue.

LLE. Load list element.

L-lock. Logical lock.

| **load list element.** A z/OS control block that controls
| the loading and deleting of a particular load module
| based on entry point names.

load module. A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB. Large object.

LOB locator. A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a fullword integer value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL operations to the associated LOB value using the locator.

LOB lock. A lock on a LOB value.

LOB table space. A table space in an auxiliary table that contains all the data for a particular LOB column in the related base table.

local. A way of referring to any object that the local DB2 subsystem maintains. A *local table*, for example, is a table that is maintained by the local DB2 subsystem. Contrast with *remote*.

locale. The definition of a subset of a user's environment that combines a CCSID and characters that are defined for a specific language and country.

local lock. A lock that provides intra-DB2 concurrency control, but not inter-DB2 concurrency control; that is, its scope is a single DB2.

local subsystem. The unique relational DBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

| **location.** The unique name of a database server. An
| application uses the location name to access a DB2

| database server. A database alias can be used to
| override the location name when accessing a remote
| server.

| **location alias.** Another name by which a database
| server identifies itself in the network. Applications can
| use this name to access a DB2 database server.

lock. A means of controlling concurrent events or access to data. DB2 locking is performed by the IRLM.

lock duration. The interval over which a DB2 lock is held.

lock escalation. The promotion of a lock from a row, page, or LOB lock to a table space lock because the number of page locks that are concurrently held on a given resource exceeds a preset limit.

locking. The process by which the integrity of data is ensured. Locking prevents concurrent users from accessing inconsistent data.

lock mode. A representation for the type of access that concurrently running programs can have to a resource that a DB2 lock is holding.

lock object. The resource that is controlled by a DB2 lock.

lock promotion. The process of changing the size or mode of a DB2 lock to a higher, more restrictive level.

lock size. The amount of data that is controlled by a DB2 lock on table data; the value can be a row, a page, a LOB, a partition, a table, or a table space.

lock structure. A coupling facility data structure that is composed of a series of lock entries to support shared and exclusive locking for logical resources.

log. A collection of records that describe the events that occur during DB2 execution and that indicate their sequence. The information thus recorded is used for recovery in the event of a failure during DB2 execution.

| **log control interval definition.** A suffix of the
| physical log record that tells how record segments are
| placed in the physical control interval.

logical claim. A claim on a logical partition of a nonpartitioning index.

logical data modeling. The process of documenting the comprehensive business information requirements in an accurate and consistent format. Data modeling is the first task of designing a database.

logical drain. A drain on a logical partition of a nonpartitioning index.

logical index partition. The set of all keys that reference the same data partition.

logical lock (L-lock). The lock type that transactions use to control intra- and inter-DB2 data concurrency between transactions. Contrast with *physical lock (P-lock)*.

logically complete. A state in which the concurrent copy process is finished with the initialization of the target objects that are being copied. The target objects are available for update.

logical page list (LPL). A list of pages that are in error and that cannot be referenced by applications until the pages are recovered. The page is in *logical error* because the actual media (coupling facility or disk) might not contain any errors. Usually a connection to the media has been lost.

logical partition. A set of key or RID pairs in a nonpartitioning index that are associated with a particular partition.

logical recovery pending (LRECP). The state in which the data and the index keys that reference the data are inconsistent.

logical unit (LU). An access point through which an application program accesses the SNA network in order to communicate with another application program.

logical unit of work (LUW). The processing that a program performs between synchronization points.

logical unit of work identifier (LUWID). A name that uniquely identifies a thread within a network. This name consists of a fully-qualified LU network name, an LUW instance number, and an LUW sequence number.

log initialization. The first phase of restart processing during which DB2 attempts to locate the current end of the log.

log record header (LRH). A prefix, in every logical record, that contains control information.

log record sequence number (LRSN). A unique identifier for a log record that is associated with a data sharing member. DB2 uses the LRSN for recovery in the data sharing environment.

log truncation. A process by which an explicit starting RBA is established. This RBA is the point at which the next byte of log data is to be written.

LPL. Logical page list.

LRECP. Logical recovery pending.

LRH. Log record header.

LRSN. Log record sequence number.

LU. Logical unit.

LU name. Logical unit name, which is the name by which VTAM refers to a node in a network. Contrast with *location name*.

LUW. Logical unit of work.

LUWID. Logical unit of work identifier.

M

mapping table. A table that the REORG utility uses to map the associations of the RIDs of data records in the original copy and in the shadow copy. This table is created by the user.

mass delete. The deletion of all rows of a table.

master terminal. The IMS logical terminal that has complete control of IMS resources during online operations.

master terminal operator (MTO). See *master terminal*.

materialize. (1) The process of putting rows from a view or nested table expression into a work file for additional processing by a query.

(2) The placement of a LOB value into contiguous storage. Because LOB values can be very large, DB2 avoids materializing LOB data until doing so becomes absolutely necessary.

| **materialized query table.** A table that is used to
| contain information that is derived and can be
| summarized from one or more source tables.

MB. Megabyte (1 048 576 bytes).

MBCS. Multibyte character set. UTF-8 is an example of an MBCS. Characters in UTF-8 can range from 1 to 4 bytes in DB2.

member name. The z/OS XCF identifier for a particular DB2 subsystem in a data sharing group.

menu. A displayed list of available functions for selection by the operator. A menu is sometimes called a *menu panel*.

| **metalanguage.** A language that is used to create other
| specialized languages.

migration. The process of converting a subsystem with a previous release of DB2 to an updated or current release. In this process, you can acquire the functions of the updated or current release without losing the data that you created on the previous release.

mixed data string. A character string that can contain both single-byte and double-byte characters.

MLPA. Modified link pack area.

MODEENT. A VTAM macro instruction that associates a logon mode name with a set of parameters representing session protocols. A set of MODEENT macro instructions defines a logon mode table.

modeling database. A DB2 database that you create on your workstation that you use to model a DB2 UDB for z/OS subsystem, which can then be evaluated by the Index Advisor.

mode name. A VTAM name for the collection of physical and logical characteristics and attributes of a session.

modify locks. An L-lock or P-lock with a MODIFY attribute. A list of these active locks is kept at all times in the coupling facility lock structure. If the requesting DB2 subsystem fails, that DB2 subsystem's modify locks are converted to retained locks.

MPP. Message processing program (in IMS).

MTO. Master terminal operator.

multibyte character set (MBCS). A character set that represents single characters with more than a single byte. Contrast with *single-byte character set* and *double-byte character set*. See also *Unicode*.

multidimensional analysis. The process of assessing and evaluating an enterprise on more than one level.

Multiple Virtual Storage. An element of the z/OS operating system. This element is also called the Base Control Program (BCP).

multisite update. Distributed relational database processing in which data is updated in more than one location within a single unit of work.

multithreading. Multiple TCBs that are executing one copy of DB2 ODBC code concurrently (sharing a processor) or in parallel (on separate central processors).

must-complete. A state during DB2 processing in which the entire operation must be completed to maintain data integrity.

mutex. Pthread mutual exclusion; a lock. A Pthread mutex variable is used as a locking mechanism to allow serialization of critical sections of code by temporarily blocking the execution of all but one thread.

| **MVS.** See *Multiple Virtual Storage*.

N

negotiable lock. A lock whose mode can be downgraded, by agreement among contending users, to be compatible to all. A physical lock is an example of a negotiable lock.

nested table expression. A fullselect in a FROM clause (surrounded by parentheses).

network identifier (NID). The network ID that is assigned by IMS or CICS, or if the connection type is RRSAF, the RRS unit of recovery ID (URID).

NID. Network identifier.

nonleaf page. A page that contains keys and page numbers of other pages in the index (either leaf or nonleaf pages). Nonleaf pages never point to actual data.

| **nonpartitioned index.** An index that is not physically
| partitioned. Both partitioning indexes and secondary
| indexes can be nonpartitioned.

nonscrollable cursor. A cursor that can be moved only in a forward direction. Nonscrollable cursors are sometimes called forward-only cursors or serial cursors.

normalization. A key step in the task of building a logical relational database design. Normalization helps you avoid redundancies and inconsistencies in your data. An entity is normalized if it meets a set of constraints for a particular normal form (first normal form, second normal form, and so on). Contrast with *denormalization*.

nondeterministic function. A user-defined function whose result is not solely dependent on the values of the input arguments. That is, successive invocations with the same argument values can produce a different answer. This type of function is sometimes called a *variant* function. Contrast this with a *deterministic function* (sometimes called a *not-variant function*), which always produces the same result for the same inputs.

not-variant function. See *deterministic function*.

| **NPSI.** See *nonpartitioned secondary index*.

NRE. Network recovery element.

NUL. The null character ('\0'), which is represented by the value X'00'. In C, this character denotes the end of a string.

null. A special value that indicates the absence of information.

NULLIF. A scalar function that evaluates two passed expressions, returning either NULL if the arguments are equal or the value of the first argument if they are not.

null-terminated host variable. A varying-length host variable in which the end of the data is indicated by a null terminator.

null terminator. In C, the value that indicates the end of a string. For EBCDIC, ASCII, and Unicode UTF-8 strings, the null terminator is a single-byte value (X'00').

For Unicode UCS-2 (wide) strings, the null terminator is a double-byte value (X'0000').

O

OASN (origin application schedule number). In IMS, a 4-byte number that is assigned sequentially to each IMS schedule since the last cold start of IMS. The OASN is used as an identifier for a unit of work. In an 8-byte format, the first 4 bytes contain the schedule number and the last 4 bytes contain the number of IMS sync points (*commit points*) during the current schedule. The OASN is part of the NID for an IMS connection.

ODBC. Open Database Connectivity.

ODBC driver. A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

OBID. Data object identifier.

Open Database Connectivity (ODBC). A Microsoft® database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time. This means that applications no longer need to be directly linked to the modules of all the database management systems that are supported.

ordinary identifier. An uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier must not be a reserved word.

ordinary token. A numeric constant, an ordinary identifier, a host identifier, or a keyword.

originating task. In a parallel group, the primary agent that receives data from other execution units (referred to as *parallel tasks*) that are executing portions of the query in parallel.

OS/390. Operating System/390.

outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves some or all of the unmatched rows of the tables that are being joined. See also *join*.

overloaded function. A function name for which multiple function instances exist.

P

package. An object containing a set of SQL statements that have been statically bound and that is available for processing. A package is sometimes also called an *application package*.

package list. An ordered list of package names that may be used to extend an application plan.

package name. The name of an object that is created by a BIND PACKAGE or REBIND PACKAGE command. The object is a bound version of a database request module (DBRM). The name consists of a location name, a collection ID, a package ID, and a version ID.

page. A unit of storage within a table space (4 KB, 8 KB, 16 KB, or 32 KB) or index space (4 KB). In a table space, a page contains one or more rows of a table. In a LOB table space, a LOB value can span more than one page, but no more than one LOB value is stored on a page.

page set. Another way to refer to a table space or index space. Each page set consists of a collection of VSAM data sets.

page set recovery pending (PSRCP). A restrictive state of an index space. In this case, the entire page set must be recovered. Recovery of a logical part is prohibited.

panel. A predefined display image that defines the locations and characteristics of display fields on a display surface (for example, a *menu panel*).

parallel complex. A cluster of machines that work together to handle multiple transactions and applications.

parallel group. A set of consecutive operations that execute in parallel and that have the same number of parallel tasks.

parallel I/O processing. A form of I/O processing in which DB2 initiates multiple concurrent requests for a single user query and performs I/O processing concurrently (in *parallel*) on multiple data partitions.

parallelism assistant. In Sysplex query parallelism, a DB2 subsystem that helps to process parts of a parallel query that originates on another DB2 subsystem in the data sharing group.

parallelism coordinator. In Sysplex query parallelism, the DB2 subsystem from which the parallel query originates.

Parallel Sysplex. A set of z/OS systems that communicate and cooperate with each other through certain multisystem hardware components and software services to process customer workloads.

parallel task. The execution unit that is dynamically created to process a query in parallel. A parallel task is implemented by a z/OS service request block.

parameter marker. A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable could appear if the statement string were a static SQL statement.

| **parameter-name.** An SQL identifier that designates a
| parameter in an SQL procedure or an SQL function.

parent key. A primary key or unique key in the parent table of a referential constraint. The values of a parent key determine the valid values of the foreign key in the referential constraint.

| **parent lock.** For explicit hierarchical locking, a lock
| that is held on a resource that might have child locks
| that are lower in the hierarchy. A parent lock is usually
| the table space lock or the partition intent lock. See also
| *child lock*.

parent row. A row whose primary key value is the foreign key value of a dependent row.

parent table. A table whose primary key is referenced by the foreign key of a dependent table.

parent table space. A table space that contains a parent table. A table space containing a dependent of that table is a dependent table space.

participant. An entity other than the commit coordinator that takes part in the commit process. The term participant is synonymous with *agent* in SNA.

partition. A portion of a page set. Each partition corresponds to a single, independently extendable data set. Partitions can be extended to a maximum size of 1, 2, or 4 GB, depending on the number of partitions in the partitioned page set. All partitions of a given page set have the same maximum size.

partitioned data set (PDS). A data set in disk storage that is divided into partitions, which are called members. Each partition can contain a program, part of a program, or data. The term partitioned data set is synonymous with program library.

| **partitioned index.** An index that is physically
| partitioned. Both partitioning indexes and secondary
| indexes can be partitioned.

partitioned page set. A partitioned table space or an index space. Header pages, space map pages, data pages, and index pages reference data only within the scope of the partition.

partitioned table space. A table space that is subdivided into parts (based on index key range), each of which can be processed independently by utilities.

| **partitioning index.** An index in which the leftmost
| columns are the partitioning columns of the table. The
| index can be partitioned or nonpartitioned.

| **partition pruning.** The removal from consideration of
| inapplicable partitions through setting up predicates in
| a query on a partitioned table to access only certain
| partitions to satisfy the query.

partner logical unit. An access point in the SNA network that is connected to the local DB2 subsystem by way of a VTAM conversation.

path. See *SQL path*.

PCT. Program control table (in CICS).

PDS. Partitioned data set.

piece. A data set of a nonpartitioned page set.

physical claim. A claim on an entire nonpartitioning index.

physical consistency. The state of a page that is not in a partially changed state.

physical drain. A drain on an entire nonpartitioning index.

physical lock (P-lock). A type of lock that DB2 acquires to provide consistency of data that is cached in different DB2 subsystems. Physical locks are used only in data sharing environments. Contrast with *logical lock (L-lock)*.

physical lock contention. Conflicting states of the requesters for a physical lock. See also *negotiable lock*.

physically complete. The state in which the concurrent copy process is completed and the output data set has been created.

plan. See *application plan*.

plan allocation. The process of allocating DB2 resources to a plan in preparation for execution.

plan member. The bound copy of a DBRM that is identified in the member clause.

plan name. The name of an application plan.

plan segmentation. The dividing of each plan into sections. When a section is needed, it is independently brought into the EDM pool.

P-lock. Physical lock.

PLT. Program list table (in CICS).

point of consistency. A time when all recoverable data that an application accesses is consistent with other data. The term point of consistency is synonymous with *sync point* or *commit point*.

policy. See *CFRM policy*.

Portable Operating System Interface (POSIX). The IEEE operating system interface standard, which defines the Pthread standard of threading. See also *Pthread*.

POSIX. Portable Operating System Interface.

postponed abort UR. A unit of recovery that was inflight or in-abort, was interrupted by system failure or cancellation, and did not complete backout during restart.

PPT. (1) Processing program table (in CICS). (2) Program properties table (in z/OS).

precision. In SQL, the total number of digits in a decimal number (called the *size* in the C language). In the C language, the number of digits to the right of the decimal point (called the *scale* in SQL). The DB2 library uses the SQL terms.

precompilation. A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

predicate. An element of a search condition that expresses or implies a comparison operation.

prefix. A code at the beginning of a message or record.

preformat. The process of preparing a VSAM ESDS for DB2 use, by writing specific data patterns.

prepare. The first phase of a two-phase commit process in which all participants are requested to prepare for commit.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

presumed-abort. An optimization of the presumed-nothing two-phase commit protocol that reduces the number of recovery log records, the duration of state maintenance, and the number of messages between coordinator and participant. The optimization also modifies the indoubt resolution responsibility.

presumed-nothing. The standard two-phase commit protocol that defines coordinator and participant responsibilities, relative to logical unit of work states, recovery logging, and indoubt resolution.

primary authorization ID. The authorization ID that is used to identify the application process to DB2.

primary group buffer pool. For a duplexed group buffer pool, the structure that is used to maintain the coherency of cached data. This structure is used for page registration and cross-invalidation. The z/OS equivalent is *old* structure. Compare with *secondary group buffer pool*.

primary index. An index that enforces the uniqueness of a primary key.

primary key. In a relational database, a unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

principal. An entity that can communicate securely with another entity. In Kerberos, principals are represented as entries in the Kerberos registry database and include users, servers, computers, and others.

principal name. The name by which a principal is known to the DCE security services.

private connection. A communications connection that is specific to DB2.

private protocol access. A method of accessing distributed data by which you can direct a query to another DB2 system. Contrast with *DRDA access*.

private protocol connection. A DB2 private connection of the application process. See also *private connection*.

privilege. The capability of performing a specific function, sometimes on a specific object. The types of privileges are:

explicit privileges, which have names and are held as the result of SQL GRANT and REVOKE statements. For example, the SELECT privilege.

implicit privileges, which accompany the ownership of an object, such as the privilege to drop a synonym that one owns, or the holding of an authority, such as the privilege of SYSADM authority to terminate any utility job.

privilege set. For the installation SYSADM ID, the set of all possible privileges. For any other authorization ID, the set of all privileges that are recorded for that ID in the DB2 catalog.

process. In DB2, the unit to which DB2 allocates resources and locks. Sometimes called an *application process*, a process involves the execution of one or more programs. The execution of an SQL statement is always associated with some process. The means of initiating and terminating a process are dependent on the environment.

program. A single, compilable collection of executable statements in a programming language.

program temporary fix (PTF). A solution or bypass of a problem that is diagnosed as a result of a defect in a

current unaltered release of a licensed program. An authorized program analysis report (APAR) fix is corrective service for an existing problem. A PTF is preventive service for problems that might be encountered by other users of the product. A PTF is *temporary*, because a permanent fix is usually not incorporated into the product until its next release.

protected conversation. A VTAM conversation that supports two-phase commit flows.

PSRCP. Page set recovery pending.

PTE. Program temporary fix.

Pthread. The POSIX threading standard model for splitting an application into subtasks. The Pthread standard includes functions for creating threads, terminating threads, synchronizing threads through locking, and other thread control facilities.

Q

QMF™. Query Management Facility.

QSAM. Queued sequential access method.

query. A component of certain SQL statements that specifies a result table.

query block. The part of a query that is represented by one of the FROM clauses. Each FROM clause can have multiple query blocks, depending on DB2's internal processing of the query.

query CP parallelism. Parallel execution of a single query, which is accomplished by using multiple tasks. See also *Sysplex query parallelism*.

query I/O parallelism. Parallel access of data, which is accomplished by triggering multiple I/O requests within a single query.

queued sequential access method (QSAM). An extended version of the basic sequential access method (BSAM). When this method is used, a queue of data blocks is formed. Input data blocks await processing, and output data blocks await transfer to auxiliary storage or to an output device.

quiesce point. A point at which data is consistent as a result of running the DB2 QUIESCE utility.

quiesced member state. A state of a member of a data sharing group. An active member becomes quiesced when a STOP DB2 command takes effect without a failure. If the member's task, address space, or z/OS system fails before the command takes effect, the member state is failed.

R

| **RACF.** Resource Access Control Facility, which is a
| component of the z/OS Security Server.

RAMAC®. IBM family of enterprise disk storage system products.

RBA. Relative byte address.

RCT. Resource control table (in CICS attachment facility).

RDB. Relational database.

RDBMS. Relational database management system.

RDBNAM. Relational database name.

RDF. Record definition field.

read stability (RS). An isolation level that is similar to repeatable read but does not completely isolate an application process from all other concurrently executing application processes. Under level RS, an application that issues the same query more than once might read additional rows that were inserted and committed by a concurrently executing application process.

rebind. The creation of a new application plan for an application program that has been bound previously. If, for example, you have added an index for a table that your application accesses, you must rebind the application in order to take advantage of that index.

rebuild. The process of reallocating a coupling facility structure. For the shared communications area (SCA) and lock structure, the structure is repopulated; for the group buffer pool, changed pages are usually cast out to disk, and the new structure is populated only with changed pages that were not successfully cast out.

RECFM. Record format.

record. The storage representation of a row or other data.

record identifier (RID). A unique identifier that DB2 uses internally to identify a row of data in a table. Compare with *row ID*.

| **record identifier (RID) pool.** An area of main storage
| that is used for sorting record identifiers during
| list-prefetch processing.

record length. The sum of the length of all the columns in a table, which is the length of the data as it is physically stored in the database. Records can be fixed length or varying length, depending on how the columns are defined. If all columns are fixed-length

columns, the record is a fixed-length record. If one or more columns are varying-length columns, the record is a varying-length column.

Recoverable Resource Manager Services attachment facility (RRSAF). A DB2 subcomponent that uses Resource Recovery Services to coordinate resource commitment between DB2 and all other resource managers that also use RRS in a z/OS system.

recovery. The process of rebuilding databases after a system failure.

recovery log. A collection of records that describes the events that occur during DB2 execution and indicates their sequence. The recorded information is used for recovery in the event of a failure during DB2 execution.

recovery manager. (1) A subcomponent that supplies coordination services that control the interaction of DB2 resource managers during commit, abort, checkpoint, and restart processes. The recovery manager also supports the recovery mechanisms of other subsystems (for example, IMS) by acting as a participant in the other subsystem's process for protecting data that has reached a point of consistency. (2) A coordinator or a participant (or both), in the execution of a two-phase commit, that can access a recovery log that maintains the state of the logical unit of work and names the immediate upstream coordinator and downstream participants.

recovery pending (RECP). A condition that prevents SQL access to a table space that needs to be recovered.

recovery token. An identifier for an element that is used in recovery (for example, NID or URID).

RECP. Recovery pending.

redo. A state of a unit of recovery that indicates that changes are to be reapplied to the disk media to ensure data integrity.

reentrant. Executable code that can reside in storage as one shared copy for all threads. Reentrant code is not self-modifying and provides separate storage areas for each thread. Reentrancy is a compiler and operating system concept, and reentrancy alone is not enough to guarantee logically consistent results when multithreading. See also *threadsafe*.

referential constraint. The requirement that nonnull values of a designated foreign key are valid only if they equal values of the primary key of a designated table.

referential integrity. The state of a database in which all values of all foreign keys are valid. Maintaining referential integrity requires the enforcement of referential constraints on all operations that change the data in a table on which the referential constraints are defined.

referential structure. A set of tables and relationships that includes at least one table and, for every table in the set, all the relationships in which that table participates and all the tables to which it is related.

refresh age. The time duration between the current time and the time during which a materialized query table was last refreshed.

registry. See *registry database*.

registry database. A database of security information about principals, groups, organizations, accounts, and security policies.

relational database (RDB). A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

relational database management system (RDBMS). A collection of hardware and software that organizes and provides access to a relational database.

relational database name (RDBNAM). A unique identifier for an RDBMS within a network. In DB2, this must be the value in the LOCATION column of table SYSIBM.LOCATIONS in the CDB. DB2 publications refer to the name of another RDBMS as a LOCATION value or a location name.

relationship. A defined connection between the rows of a table or the rows of two tables. A relationship is the internal representation of a referential constraint.

relative byte address (RBA). The offset of a data record or control interval from the beginning of the storage space that is allocated to the data set or file to which it belongs.

remigration. The process of returning to a current release of DB2 following a fallback to a previous release. This procedure constitutes another migration process.

remote. Any object that is maintained by a remote DB2 subsystem (that is, by a DB2 subsystem other than the local one). A *remote view*, for example, is a view that is maintained by a remote DB2 subsystem. Contrast with *local*.

remote attach request. A request by a remote location to attach to the local DB2 subsystem. Specifically, the request that is sent is an SNA Function Management Header 5.

remote subsystem. Any relational DBMS, except the *local subsystem*, with which the user or application can communicate. The subsystem need not be remote in any physical sense, and might even operate on the same processor under the same z/OS system.

reoptimization. The DB2 process of reconsidering the access path of an SQL statement at run time; during

reoptimization, DB2 uses the values of host variables, parameter markers, or special registers.

REORG pending (REORP). A condition that restricts SQL access and most utility access to an object that must be reorganized.

REORP. REORG pending.

repeatable read (RR). The isolation level that provides maximum protection from other executing application programs. When an application program executes with repeatable read protection, rows that the program references cannot be changed by other programs until the program reaches a commit point.

repeating group. A situation in which an entity includes multiple attributes that are inherently the same. The presence of a repeating group violates the requirement of first normal form. In an entity that satisfies the requirement of first normal form, each attribute is independent and unique in its meaning and its name. See also *normalization*.

replay detection mechanism. A method that allows a principal to detect whether a request is a valid request from a source that can be trusted or whether an untrustworthy entity has captured information from a previous exchange and is replaying the information exchange to gain access to the principal.

request commit. The vote that is submitted to the prepare phase if the participant has modified data and is prepared to commit or roll back.

requester. The source of a request to access data at a remote server. In the DB2 environment, the requester function is provided by the distributed data facility.

resource. The object of a lock or claim, which could be a table space, an index space, a data partition, an index partition, or a logical partition.

resource allocation. The part of plan allocation that deals specifically with the database resources.

resource control table (RCT). A construct of the CICS attachment facility, created by site-provided macro parameters, that defines authorization and access attributes for transactions or transaction groups.

resource definition online. A CICS feature that you use to define CICS resources online without assembling tables.

resource limit facility (RLF). A portion of DB2 code that prevents dynamic manipulative SQL statements from exceeding specified time limits. The resource limit facility is sometimes called the governor.

resource limit specification table (RLST). A site-defined table that specifies the limits to be enforced by the resource limit facility.

resource manager. (1) A function that is responsible for managing a particular resource and that guarantees the consistency of all updates made to recoverable resources within a logical unit of work. The resource that is being managed can be physical (for example, disk or main storage) or logical (for example, a particular type of system service). (2) A participant, in the execution of a two-phase commit, that has recoverable resources that could have been modified. The resource manager has access to a recovery log so that it can commit or roll back the effects of the logical unit of work to the recoverable resources.

restart pending (RESTOP). A restrictive state of a page set or partition that indicates that restart (backout) work needs to be performed on the object. All access to the page set or partition is denied except for access by the:

- RECOVER POSTPONED command
- Automatic online backout (which DB2 invokes after restart if the system parameter LBACKOUT=AUTO)

RESTOP. Restart pending.

result set. The set of rows that a stored procedure returns to a client application.

result set locator. A 4-byte value that DB2 uses to uniquely identify a query result set that a stored procedure returns.

result table. The set of rows that are specified by a SELECT statement.

retained lock. A MODIFY lock that a DB2 subsystem was holding at the time of a subsystem failure. The lock is retained in the coupling facility lock structure across a DB2 failure.

RID. Record identifier.

RID pool. Record identifier pool.

right outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of the second join operand. See also *join*.

RLF. Resource limit facility.

RLST. Resource limit specification table.

RMID. Resource manager identifier.

RO. Read-only access.

rollback. The process of restoring data that was changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

root page. The index page that is at the highest level (or the beginning point) in an index.

routine. A term that refers to either a user-defined function or a stored procedure.

row. The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

ROWID. Row identifier.

row identifier (ROWID). A value that uniquely identifies a row. This value is stored with the row and never changes.

row lock. A lock on a single row of data.

| **rowset.** A set of rows for which a cursor position is established.

| **rowset cursor.** A cursor that is defined so that one or more rows can be returned as a rowset for a single FETCH statement, and the cursor is positioned on the set of rows that is fetched.

| **rowset-positioned access.** The ability to retrieve multiple rows from a single FETCH statement.

| **row-positioned access.** The ability to retrieve a single row from a single FETCH statement.

row trigger. A trigger that is defined with the trigger granularity FOR EACH ROW.

RRE. Residual recovery entry (in IMS).

RRSAF. Recoverable Resource Manager Services attachment facility.

RS. Read stability.

RTT. Resource translation table.

RURE. Restart URE.

S

savepoint. A named entity that represents the state of data and schemas at a particular point in time within a unit of work. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents. The restoration of data and schemas to a savepoint is usually referred to as *rolling back to a savepoint*.

SBCS. Single-byte character set.

SCA. Shared communications area.

scalar function. An SQL operation that produces a single value from another value and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses. Contrast with *aggregate function*.

scale. In SQL, the number of digits to the right of the decimal point (called the *precision* in the C language). The DB2 library uses the SQL definition.

| **schema.** (1) The organization or structure of a
| database. (2) A logical grouping for user-defined
| functions, distinct types, triggers, and stored
| procedures. When an object of one of these types is
| created, it is assigned to one schema, which is
| determined by the name of the object. For example, the
| following statement creates a distinct type T in schema
| C:
| CREATE DISTINCT TYPE C.T ...

scrollability. The ability to use a cursor to fetch in either a forward or backward direction. The FETCH statement supports multiple fetch orientations to indicate the new position of the cursor. See also *fetch orientation*.

scrollable cursor. A cursor that can be moved in both a forward and a backward direction.

SDWA. System diagnostic work area.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

secondary authorization ID. An authorization ID that has been associated with a primary authorization ID by an authorization exit routine.

secondary group buffer pool. For a duplexed group buffer pool, the structure that is used to back up changed pages that are written to the primary group buffer pool. No page registration or cross-invalidation occurs using the secondary group buffer pool. The z/OS equivalent is *new* structure.

| **secondary index.** A nonpartitioning index on a
| partitioned table.

section. The segment of a plan or package that contains the executable structures for a single SQL statement. For most SQL statements, one section in the plan exists for each SQL statement in the source program. However, for cursor-related statements, the DECLARE, OPEN, FETCH, and CLOSE statements reference the same section because they each refer to the SELECT statement that is named in the DECLARE CURSOR statement. SQL statements such as COMMIT, ROLLBACK, and some SET statements do not use a section.

segment. A group of pages that holds rows of a single table. See also *segmented table space*.

segmented table space. A table space that is divided into equal-sized groups of pages called segments. Segments are assigned to tables so that rows of different tables are never stored in the same segment.

self-referencing constraint. A referential constraint that defines a relationship in which a table is a dependent of itself.

self-referencing table. A table with a self-referencing constraint.

| **sensitive cursor.** A cursor that is sensitive to changes
| that are made to the database after the result table has
| been materialized.

| **sequence.** A user-defined object that generates a
| sequence of numeric values according to user
| specifications.

sequential data set. A non-DB2 data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Several of the DB2 database utilities require sequential data sets.

sequential prefetch. A mechanism that triggers consecutive asynchronous I/O operations. Pages are fetched before they are required, and several pages are read with a single I/O operation.

serial cursor. A cursor that can be moved only in a forward direction.

serialized profile. A Java object that contains SQL statements and descriptions of host variables. The SQLJ translator produces a serialized profile for each connection context.

server. The target of a request from a remote requester. In the DB2 environment, the server function is provided by the distributed data facility, which is used to access DB2 data from remote applications.

server-side programming. A method for adding DB2 data into dynamic Web pages.

service class. An eight-character identifier that is used by the z/OS Workload Manager to associate user performance goals with a particular DDF thread or stored procedure. A service class is also used to classify work on parallelism assistants.

service request block. A unit of work that is scheduled to execute in another address space.

session. A link between two nodes in a VTAM network.

session protocols. The available set of SNA communication requests and responses.

shared communications area (SCA). A coupling facility list structure that a DB2 data sharing group uses for inter-DB2 communication.

share lock. A lock that prevents concurrently executing application processes from changing data, but not from reading data. Contrast with *exclusive lock*.

shift-in character. A special control character (X'0F') that is used in EBCDIC systems to denote that the subsequent bytes represent SBCS characters. See also *shift-out character*.

shift-out character. A special control character (X'0E') that is used in EBCDIC systems to denote that the subsequent bytes, up to the next shift-in control character, represent DBCS characters. See also *shift-in character*.

sign-on. A request that is made on behalf of an individual CICS or IMS application process by an attachment facility to enable DB2 to verify that it is authorized to use DB2 resources.

simple page set. A nonpartitioned page set. A simple page set initially consists of a single data set (page set piece). If and when that data set is extended to 2 GB, another data set is created, and so on, up to a total of 32 data sets. DB2 considers the data sets to be a single contiguous linear address space containing a maximum of 64 GB. Data is stored in the next available location within this address space without regard to any partitioning scheme.

simple table space. A table space that is neither partitioned nor segmented.

single-byte character set (SBCS). A set of characters in which each character is represented by a single byte. Contrast with *double-byte character set* or *multibyte character set*.

single-precision floating point number. A 32-bit approximate representation of a real number.

size. In the C language, the total number of digits in a decimal number (called the *precision* in SQL). The DB2 library uses the SQL term.

SMF. System Management Facilities.

SMP/E. System Modification Program/Extended.

SMS. Storage Management Subsystem.

SNA. Systems Network Architecture.

SNA network. The part of a network that conforms to the formats and protocols of Systems Network Architecture (SNA).

socket. A callable TCP/IP programming interface that TCP/IP network applications use to communicate with remote TCP/IP partners.

sourced function. A function that is implemented by another built-in or user-defined function that is already known to the database manager. This function can be a scalar function or a column (aggregating) function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with *built-in function*, *external function*, and *SQL function*.

source program. A set of host language statements and SQL statements that is processed by an SQL precompiler.

| **source table.** A table that can be a base table, a view, a
| table expression, or a user-defined table function.

source type. An existing type that DB2 uses to internally represent a distinct type.

space. A sequence of one or more blank characters.

special register. A storage area that DB2 defines for an application process to use for storing information that can be referenced in SQL statements. Examples of special registers are USER and CURRENT DATE.

specific function name. A particular user-defined function that is known to the database manager by its specific name. Many specific user-defined functions can have the same function name. When a user-defined function is defined to the database, every function is assigned a specific name that is unique within its schema. Either the user can provide this name, or a default name is used.

SPUFI. SQL Processor Using File Input.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQLCA. SQL communication area.

SQL communication area (SQLCA). A structure that is used to provide an application program with information about the execution of its SQL statements.

SQL connection. An association between an application process and a local or remote application server or database server.

SQLDA. SQL descriptor area.

SQL descriptor area (SQLDA). A structure that describes input variables, output variables, or the columns of a result table.

SQL escape character. The symbol that is used to enclose an SQL delimited identifier. This symbol is the double quotation mark ("). See also *escape character*.

SQL function. A user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL user-defined function can return only one parameter.

SQL ID. SQL authorization ID.

SQLJ. Structured Query Language (SQL) that is embedded in the Java programming language.

SQL path. An ordered list of schema names that are used in the resolution of unqualified references to user-defined functions, distinct types, and stored procedures. In dynamic SQL, the current path is found in the CURRENT PATH special register. In static SQL, it is defined in the PATH bind option.

SQL procedure. A user-written program that can be invoked with the SQL CALL statement. Contrast with *external procedure*.

SQL processing conversation. Any conversation that requires access of DB2 data, either through an application or by dynamic query requests.

SQL Processor Using File Input (SPUFI). A facility of the TSO attachment subcomponent that enables the DB2I user to execute SQL statements without embedding them in an application program.

SQL return code. Either SQLCODE or SQLSTATE.

SQL routine. A user-defined function or stored procedure that is based on code that is written in SQL.

SQL statement coprocessor. An alternative to the DB2 precompiler that lets the user process SQL statements at compile time. The user invokes an SQL statement coprocessor by specifying a compiler option.

SQL string delimiter. A symbol that is used to enclose an SQL string constant. The SQL string delimiter is the apostrophe ('), except in COBOL applications, where the user assigns the symbol, which is either an apostrophe or a double quotation mark (").

SRB. Service request block.

SSI. Subsystem interface (in z/OS).

SSM. Subsystem member (in IMS).

stand-alone. An attribute of a program that means that it is capable of executing separately from DB2, without using DB2 services.

star join. A method of joining a dimension column of a fact table to the key column of the corresponding dimension table. See also *join*, *dimension*, and *star schema*.

star schema. The combination of a fact table (which contains most of the data) and a number of dimension tables. See also *star join*, *dimension*, and *dimension table*.

statement handle. In DB2 ODBC, the data object that contains information about an SQL statement that is managed by DB2 ODBC. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values, and status information. Each statement handle is associated with the connection handle.

statement string. For a dynamic SQL statement, the character string form of the statement.

statement trigger. A trigger that is defined with the trigger granularity FOR EACH STATEMENT.

| **static cursor.** A named control structure that does not
| change the size of the result table or the order of its
| rows after an application opens the cursor. Contrast
| with *dynamic cursor*.

static SQL. SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables that are specified by the statement might change).

storage group. A named set of disks on which DB2 data can be stored.

stored procedure. A user-written application program that can be invoked through the use of the SQL CALL statement.

string. See *character string* or *graphic string*.

strong typing. A process that guarantees that only user-defined functions and operations that are defined on a distinct type can be applied to that type. For example, you cannot directly compare two currency types, such as Canadian dollars and U.S. dollars. But you can provide a user-defined function to convert one currency to the other and then do the comparison.

structure. (1) A name that refers collectively to different types of DB2 objects, such as tables, databases, views, indexes, and table spaces. (2) A construct that uses z/OS to map and manage storage on a coupling facility. See also *cache structure*, *list structure*, or *lock structure*.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

structure owner. In relation to group buffer pools, the DB2 member that is responsible for the following activities:

- Coordinating rebuild, checkpoint, and damage assessment processing
- Monitoring the group buffer pool threshold and notifying castout owners when the threshold has been reached

subcomponent. A group of closely related DB2 modules that work together to provide a general function.

subject table. The table for which a trigger is created. When the defined triggering event occurs on this table, the trigger is activated.

subpage. The unit into which a physical index page can be divided.

subquery. A SELECT statement within the WHERE or HAVING clause of another SQL statement; a nested SQL statement.

subselect. That form of a query that does not include an ORDER BY clause, an UPDATE clause, or UNION operators.

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

subsystem. A distinct instance of a relational database management system (RDBMS).

surrogate pair. A coded representation for a single character that consists of a sequence of two 16-bit code units, in which the first value of the pair is a high-surrogate code unit in the range U+D800 through U+DBFF, and the second value is a low-surrogate code unit in the range U+DC00 through U+DFFF. Surrogate pairs provide an extension mechanism for encoding 917 476 characters without requiring the use of 32-bit characters.

SVC dump. A dump that is issued when a z/OS or a DB2 functional recovery routine detects an error.

sync point. See *commit point*.

syncpoint tree. The tree of recovery managers and resource managers that are involved in a logical unit of work, starting with the recovery manager, that make the final commit decision.

synonym. In SQL, an alternative name for a table or view. Synonyms can be used to refer only to objects at the subsystem in which the synonym is defined.

syntactic character set. A set of 81 graphic characters that are registered in the IBM registry as character set 00640. This set was originally recommended to the programming language community to be used for syntactic purposes toward maximizing portability and interchangeability across systems and country boundaries. It is contained in most of the primary registered character sets, with a few exceptions. See also *invariant character set*.

Sysplex. See *Parallel Sysplex*.

Sysplex query parallelism. Parallel execution of a single query that is accomplished by using multiple tasks on more than one DB2 subsystem. See also *query CP parallelism*.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

system agent. A work request that DB2 creates internally such as prefetch processing, deferred writes, and service tasks.

system conversation. The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

system diagnostic work area (SDWA). The data that is recorded in a SYS1.LOGREC entry that describes a program or hardware error.

system-directed connection. A connection that a relational DBMS manages by processing SQL statements with three-part names.

System Modification Program/Extended (SMP/E). A z/OS tool for making software changes in programming systems (such as DB2) and for controlling those changes.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information through and controlling the configuration and operation of networks.

SYS1.DUMPxx data set. A data set that contains a system dump (in z/OS).

SYS1.LOGREC. A service aid that contains important information about program and hardware errors (in z/OS).

T

table. A named data object consisting of a specific number of columns and some number of unordered rows. See also *base table* or *temporary table*.

| **table-controlled partitioning.** A type of partitioning in
| which partition boundaries for a partitioned table are
| controlled by values that are defined in the CREATE
| TABLE statement. Partition limits are saved in the
| LIMITKEY_INTERNAL column of the
| SYSIBM.SYSTABLEPART catalog table.

table function. A function that receives a set of arguments and returns a table to the SQL statement that references the function. A table function can be referenced only in the FROM clause of a subselect.

table locator. A mechanism that allows access to trigger transition tables in the FROM clause of SELECT statements, in the subselect of INSERT statements, or from within user-defined functions. A table locator is a fullword integer value that represents a transition table.

table space. A page set that is used to store the records in one or more tables.

table space set. A set of table spaces and partitions that should be recovered together for one of these reasons:

- Each of them contains a table that is a parent or descendent of a table in one of the others.
- The set contains a base table and associated auxiliary tables.

A table space set can contain both types of relationships.

task control block (TCB). A z/OS control block that is used to communicate information about tasks within an address space that are connected to DB2. See also *address space connection*.

TB. Terabyte (1 099 511 627 776 bytes).

TCB. Task control block (in z/OS).

TCP/IP. A network communication protocol that computer systems use to exchange information across telecommunication links.

TCP/IP port. A 2-byte value that identifies an end user or a TCP/IP network application within a TCP/IP host.

template. A DB2 utilities output data set descriptor that is used for dynamic allocation. A template is defined by the TEMPLATE utility control statement.

temporary table. A table that holds temporary data. Temporary tables are useful for holding or sorting intermediate results from queries that contain a large number of rows. The two types of temporary table, which are created by different SQL statements, are the created temporary table and the declared temporary table. Contrast with *result table*. See also *created temporary table* and *declared temporary table*.

Terminal Monitor Program (TMP). A program that provides an interface between terminal users and command processors and has access to many system services (in z/OS).

thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services. Most DB2 functions execute under a thread structure. See also *allied thread* and *database access thread*.

threadsafe. A characteristic of code that allows multithreading both by providing private storage areas for each thread, and by properly serializing shared (global) storage areas.

three-part name. The full name of a table, view, or alias. It consists of a location name, authorization ID, and an object name, separated by a period.

time. A three-part value that designates a time of day in hours, minutes, and seconds.

time duration. A decimal integer that represents a number of hours, minutes, and seconds.

timeout. Abnormal termination of either the DB2 subsystem or of an application because of the unavailability of resources. Installation specifications are set to determine both the amount of time DB2 is to wait for IRLM services after starting, and the amount of time IRLM is to wait if a resource that an application requests is unavailable. If either of these time specifications is exceeded, a timeout is declared.

Time-Sharing Option (TSO). An option in MVS that provides interactive time sharing from remote terminals.

timestamp. A seven-part value that consists of a date and time. The timestamp is expressed in years, months, days, hours, minutes, seconds, and microseconds.

TMP. Terminal Monitor Program.

to-do. A state of a unit of recovery that indicates that the unit of recovery's changes to recoverable DB2 resources are indoubt and must either be applied to the disk media or backed out, as determined by the commit coordinator.

trace. A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

transaction lock. A lock that is used to control concurrent execution of SQL statements.

transaction program name. In SNA LU 6.2 conversations, the name of the program at the remote logical unit that is to be the other half of the conversation.

| **transient XML data type.** A data type for XML values
| that exists only during query processing.

transition table. A temporary table that contains all the affected rows of the subject table in their state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the table of changed rows in the old state or the new state.

transition variable. A variable that contains a column value of the affected row of the subject table in its state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the set of old values or the set of new values.

| **tree structure.** A data structure that represents entities
| in nodes, with a most one parent node for each node,
| and with only one root node.

trigger. A set of SQL statements that are stored in a DB2 database and executed when a certain event occurs in a DB2 table.

trigger activation. The process that occurs when the trigger event that is defined in a trigger definition is executed. Trigger activation consists of the evaluation of the triggered action condition and conditional execution of the triggered SQL statements.

trigger activation time. An indication in the trigger definition of whether the trigger should be activated before or after the triggered event.

trigger body. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. A trigger body is also called *triggered SQL statements*.

trigger cascading. The process that occurs when the triggered action of a trigger causes the activation of another trigger.

triggered action. The SQL logic that is performed when a trigger is activated. The triggered action consists of an optional triggered action condition and a set of triggered SQL statements that are executed only if the condition evaluates to true.

triggered action condition. An optional part of the triggered action. This Boolean condition appears as a WHEN clause and specifies a condition that DB2 evaluates to determine if the triggered SQL statements should be executed.

triggered SQL statements. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. Triggered SQL statements are also called the *trigger body*.

trigger granularity. A characteristic of a trigger, which determines whether the trigger is activated:

- Only once for the triggering SQL statement
- Once for each row that the SQL statement modifies

triggering event. The specified operation in a trigger definition that causes the activation of that trigger. The triggering event is comprised of a triggering operation (INSERT, UPDATE, or DELETE) and a subject table on which the operation is performed.

triggering SQL operation. The SQL operation that causes a trigger to be activated when performed on the subject table.

trigger package. A package that is created when a CREATE TRIGGER statement is executed. The package is executed when the trigger is activated.

TSO. Time-Sharing Option.

TSO attachment facility. A DB2 facility consisting of the DSN command processor and DB2I. Applications

that are not written for the CICS or IMS environments can run under the TSO attachment facility.

typed parameter marker. A parameter marker that is specified along with its target data type. It has the general form:

CAST(? AS data-type)

type 1 indexes. Indexes that were created by a release of DB2 before DB2 Version 4 or that are specified as type 1 indexes in Version 4. Contrast with *type 2 indexes*. As of Version 8, type 1 indexes are no longer supported.

type 2 indexes. Indexes that are created on a release of DB2 after Version 7 or that are specified as type 2 indexes in Version 4 or later.

U

UCS-2. Universal Character Set, coded in 2 octets, which means that characters are represented in 16-bits per character.

UDF. User-defined function.

UDT. User-defined data type. In DB2 UDB for z/OS, the term *distinct type* is used instead of user-defined data type. See *distinct type*.

uncommitted read (UR). The isolation level that allows an application to read uncommitted data.

underlying view. The view on which another view is directly or indirectly defined.

undo. A state of a unit of recovery that indicates that the changes that the unit of recovery made to recoverable DB2 resources must be backed out.

Unicode. A standard that parallels the ISO-10646 standard. Several implementations of the Unicode standard exist, all of which have the ability to represent a large percentage of the characters that are contained in the many scripts that are used throughout the world.

uniform resource locator (URL). A Web address, which offers a way of naming and locating specific items on the Web.

union. An SQL operation that combines the results of two SELECT statements. Unions are often used to merge lists of values that are obtained from several tables.

unique constraint. An SQL rule that no two values in a primary key, or in the key of a unique index, can be the same.

unique index. An index that ensures that no identical key values are stored in a column or a set of columns in a table.

unit of recovery. A recoverable sequence of operations within a single resource manager, such as an instance of DB2. Contrast with *unit of work*.

unit of recovery identifier (URID). The LOGRBA of the first log record for a unit of recovery. The URID also appears in all subsequent log records for that unit of recovery.

unit of work. A recoverable sequence of operations within an application process. At any time, an application process is a single unit of work, but the life of an application process can involve many units of work as a result of commit or rollback operations. In a *multisite update* operation, a single unit of work can include several *units of recovery*. Contrast with *unit of recovery*.

Universal Unique Identifier (UUID). An identifier that is immutable and unique across time and space (in z/OS).

unlock. The act of releasing an object or system resource that was previously locked and returning it to general availability within DB2.

untyped parameter marker. A parameter marker that is specified without its target data type. It has the form of a single question mark (?).

updatability. The ability of a cursor to perform positioned updates and deletes. The updatability of a cursor can be influenced by the SELECT statement and the cursor sensitivity option that is specified on the DECLARE CURSOR statement.

update hole. The location on which a cursor is positioned when a row in a result table is fetched again and the new values no longer satisfy the search condition. DB2 marks a row in the result table as an update hole when an update to the corresponding row in the database causes that row to no longer qualify for the result table.

update trigger. A trigger that is defined with the triggering SQL operation UPDATE.

upstream. The node in the syncpoint tree that is responsible, in addition to other recovery or resource managers, for coordinating the execution of a two-phase commit.

UR. Uncommitted read.

URE. Unit of recovery element.

URID . Unit of recovery identifier.

URL. Uniform resource locator.

user-defined data type (UDT). See *distinct type*.

user-defined function (UDF). A function that is defined to DB2 by using the CREATE FUNCTION

statement and that can be referenced thereafter in SQL statements. A user-defined function can be an *external function*, a *sourced function*, or an *SQL function*. Contrast with *built-in function*.

user view. In logical data modeling, a model or representation of critical information that the business requires.

UTF-8. Unicode Transformation Format, 8-bit encoding form, which is designed for ease of use with existing ASCII-based systems. The CCSID value for data in UTF-8 format is 1208. DB2 UDB for z/OS supports UTF-8 in mixed data fields.

UTF-16. Unicode Transformation Format, 16-bit encoding form, which is designed to provide code values for over a million characters and a superset of UCS-2. The CCSID value for data in UTF-16 format is 1200. DB2 UDB for z/OS supports UTF-16 in graphic data fields.

UUID. Universal Unique Identifier.

V

value. The smallest unit of data that is manipulated in SQL.

variable. A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with *constant*.

variant function. See *nondeterministic function*.

varying-length string. A character or graphic string whose length varies within set limits. Contrast with *fixed-length string*.

version. A member of a set of similar programs, DBRMs, packages, or LOBs.

A version of a program is the source code that is produced by precompiling the program. The program version is identified by the program name and a timestamp (consistency token).

A version of a DBRM is the DBRM that is produced by precompiling a program. The DBRM version is identified by the same program name and timestamp as a corresponding program version.

A version of a package is the result of binding a DBRM within a particular database system. The package version is identified by the same program name and consistency token as the DBRM.

A version of a LOB is a copy of a LOB value at a point in time. The version number for a LOB is stored in the auxiliary index entry for the LOB.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns that are contained in tables on which it is defined.

view check option. An option that specifies whether every row that is inserted or updated through a view must conform to the definition of that view. A view check option can be specified with the WITH CASCADED CHECK OPTION, WITH CHECK OPTION, or WITH LOCAL CHECK OPTION clauses of the CREATE VIEW statement.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed- and varying-length records on disk devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number (in z/OS).

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network (in z/OS).

| **volatile table.** A table for which SQL operations
| choose index access whenever possible.

VSAM. Virtual Storage Access Method.

VTAM. Virtual Telecommunication Access Method (in z/OS).

W

warm start. The normal DB2 restart process, which involves reading and processing log records so that data that is under the control of DB2 is consistent. Contrast with *cold start*.

WLM application environment. A z/OS Workload Manager attribute that is associated with one or more stored procedures. The WLM application environment determines the address space in which a given DB2 stored procedure runs.

write to operator (WTO). An optional user-coded service that allows a message to be written to the system console operator informing the operator of errors and unusual system conditions that might need to be corrected (in z/OS).

WTO. Write to operator.

WTOR. Write to operator (WTO) with reply.

X

XCF. See *cross-system coupling facility*.

XES. See *cross-system extended services*.

| **XML.** See *Extensible Markup Language*.

| **XML attribute.** A name-value pair within a tagged
| XML element that modifies certain features of the
| element.

XML element. A logical structure in an XML
document that is delimited by a start and an end tag.
Anything between the start tag and the end tag is the
content of the element.

| **XML node.** The smallest unit of valid, complete
| structure in a document. For example, a node can
| represent an element, an attribute, or a text string.

| **XML publishing functions.** Functions that return
| XML values from SQL values.

X/Open. An independent, worldwide open systems organization that is supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal is to increase the portability of applications by combining existing and emerging standards.

XRF. Extended recovery facility.

Z

| **z/OS.** An operating system for the eServer™ product
| line that supports 64-bit real and virtual storage.

z/OS Distributed Computing Environment (z/OS DCE). A set of technologies that are provided by the Open Software Foundation to implement distributed computing.

Bibliography

DB2 Universal Database for z/OS Version 8 product information:

- *DB2 Administration Guide*, SC18-7413
- *DB2 Application Programming and SQL Guide*, SC18-7415
- *DB2 Application Programming Guide and Reference for Java*, SC18-7414
- *DB2 Codes*, GC18-9603
- *DB2 Command Reference*, SC18-7416
- *DB2 Common Criteria Guide*, SC18-9672
- *DB2 Data Sharing: Planning and Administration*, SC18-7417
- *DB2 Diagnosis Guide and Reference*, LY37-3201
- *DB2 Diagnostic Quick Reference Card*, LY37-3202
- *DB2 Image, Audio, and Video Extenders Administration and Programming*, SC26-9947
- # • *DB2 Installation Guide*, GC18-7418
- *DB2 Licensed Program Specifications*, GC18-7420
- *DB2 Management Clients Package Program Directory*, GI10-8567
- *DB2 Messages*, GC18-9602
- *DB2 ODBC Guide and Reference*, SC18-7423
- *The Official Introduction to DB2 UDB for z/OS*
- *DB2 Program Directory*, GI10-8566
- *DB2 RACF Access Control Module Guide*, SC18-7433
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC18-7424
- *DB2 Reference Summary*, SX26-3853
- *DB2 Release Planning Guide*, SC18-7425
- *DB2 SQL Reference*, SC18-7426
- *DB2 Text Extender Administration and Programming*, SC26-9948
- # • *DB2 Utility Guide and Reference*, SC18-7427
- *DB2 What's New?*, GC18-7428
- *DB2 XML Extender for z/OS Administration and Programming*, SC18-7431

Books and resources about related products:

APL2®

- *APL2 Programming Guide*, SH21-1072
- *APL2 Programming: Language Reference*, SH21-1061

- *APL2 Programming: Using Structured Query Language (SQL)*, SH21-1057

BookManager READ/MVS

- *BookManager READ/MVS V1R3: Installation Planning & Customization*, SC38-2035

C language: IBM C/C++ for z/OS

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821

Character Data Representation Architecture

- *Character Data Representation Architecture Overview*, GC09-2207
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

CICS Transaction Server for z/OS

The publication order numbers below are for Version 2 Release 2 and Version 2 Release 3 (with the release 2 number listed first).

- *CICS Transaction Server for z/OS Information Center*, SK3T-6903 or SK3T-6957.
- *CICS Transaction Server for z/OS Application Programming Guide*, SC34-5993 or SC34-6231
- *CICS Transaction Server for z/OS Application Programming Reference*, SC34-5994 or SC34-6232
- *CICS Transaction Server for z/OS CICS-RACF Security Guide*, SC34-6011 or SC34-6249
- *CICS Transaction Server for z/OS CICS Supplied Transactions*, SC34-5992 or SC34-6230
- *CICS Transaction Server for z/OS Customization Guide*, SC34-5989 or SC34-6227
- *CICS Transaction Server for z/OS Data Areas*, LY33-6100 or LY33-6103
- *CICS Transaction Server for z/OS DB2 Guide*, SC34-6014 or SC34-6252
- *CICS Transaction Server for z/OS External Interfaces Guide*, SC34-6006 or SC34-6244
- *CICS Transaction Server for z/OS Installation Guide*, GC34-5985 or GC34-6224
- *CICS Transaction Server for z/OS Intercommunication Guide*, SC34-6005 or SC34-6243
- *CICS Transaction Server for z/OS Messages and Codes*, GC34-6003 or GC34-6241
- *CICS Transaction Server for z/OS Operations and Utilities Guide*, SC34-5991 or SC34-6229

- *CICS Transaction Server for z/OS Performance Guide*, SC34-6009 or SC34-6247
- *CICS Transaction Server for z/OS Problem Determination Guide*, SC34-6002 or SC34-6239
- *CICS Transaction Server for z/OS Release Guide*, GC34-5983 or GC34-6218
- *CICS Transaction Server for z/OS Resource Definition Guide*, SC34-5990 or SC34-6228
- *CICS Transaction Server for z/OS System Definition Guide*, SC34-5988 or SC34-6226
- *CICS Transaction Server for z/OS System Programming Reference*, SC34-5595 or SC34-6233

CICS Transaction Server for OS/390

- *CICS Transaction Server for OS/390 Application Programming Guide*, SC33-1687
- *CICS Transaction Server for OS/390 DB2 Guide*, SC33-1939
- *CICS Transaction Server for OS/390 External Interfaces Guide*, SC33-1944
- *CICS Transaction Server for OS/390 Resource Definition Guide*, SC33-1684

COBOL:

- *IBM COBOL Language Reference*, SC27-1408
- *Enterprise COBOL for z/OS Programming Guide*, SC27-1412

Database Design

- *DB2 for z/OS and OS/390 Development for Performance Volume I* by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-605-2
- *DB2 for z/OS and OS/390 Development for Performance Volume II* by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-606-0
- *Handbook of Relational Database Design* by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8

DB2 Administration Tool

- *DB2 Administration Tool for z/OS User's Guide and Reference*, available on the Web at www.ibm.com/software/data/db2imstools/library.html

DB2 Buffer Pool Analyzer for z/OS

- *DB2 Buffer Pool Tool for z/OS User's Guide and Reference*, available on the Web at www.ibm.com/software/data/db2imstools/library.html

DB2 Connect™

- *IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition*, GC09-4833

- *IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition*, GC09-4834
- *IBM DB2 Connect User's Guide*, SC09-4835

DB2 DataPropagator

- *DB2 Universal Database Replication Guide and Reference*, SC27-1121

DB2 Performance Expert for z/OS, Version 1

The following books are part of the DB2 Performance Expert library. Some of these books include information about the following tools: IBM DB2 Performance Expert for z/OS; IBM DB2 Performance Monitor for z/OS; and DB2 Buffer Pool Analyzer for z/OS.

- *OMEGAMON Buffer Pool Analyzer User's Guide*, SC18-7972
- *OMEGAMON Configuration and Customization*, SC18-7973
- *OMEGAMON Messages*, SC18-7974
- *OMEGAMON Monitoring Performance from ISPF*, SC18-7975
- *OMEGAMON Monitoring Performance from Performance Expert Client*, SC18-7976
- *OMEGAMON Program Directory*, GI10-8549
- *OMEGAMON Report Command Reference*, SC18-7977
- *OMEGAMON Report Reference*, SC18-7978
- *Using IBM Tivoli OMEGAMON XE on z/OS*, SC18-7979

DB2 Query Management Facility (QMF) Version 8.1

- *DB2 Query Management Facility: DB2 QMF High Performance Option User's Guide for TSO/CICS*, SC18-7450
- *DB2 Query Management Facility: DB2 QMF Messages and Codes*, GC18-7447
- *DB2 Query Management Facility: DB2 QMF Reference*, SC18-7446
- *DB2 Query Management Facility: Developing DB2 QMF Applications*, SC18-7651
- *DB2 Query Management Facility: Getting Started with DB2 QMF for Windows and DB2 QMF for WebSphere*, SC18-7449
- *DB2 Query Management Facility: Getting Started with DB2 QMF Query Miner*, GC18-7451
- *DB2 Query Management Facility: Installing and Managing DB2 QMF for TSO/CICS*, GC18-7444
- *DB2 Query Management Facility: Installing and Managing DB2 QMF for Windows and DB2 QMF for WebSphere*, GC18-7448

- *DB2 Query Management Facility: Introducing DB2 QMF*, GC18-7443
- *DB2 Query Management Facility: Using DB2 QMF*, SC18-7445
- *DB2 Query Management Facility: DB2 QMF Visionary Developer's Guide*, SC18-9093
- *DB2 Query Management Facility: DB2 QMF Visionary Getting Started Guide*, GC18-9092

DB2 Redbooks™

For access to all IBM Redbooks about DB2, see the IBM Redbooks Web page at www.ibm.com/redbooks

DB2 Server for VSE & VM

- *DB2 Server for VM: DBS Utility*, SC09-2983

DB2 Universal Database Cross-Platform information

- *IBM DB2 Universal Database SQL Reference for Cross-Platform Development*, available at www.ibm.com/software/data/developer/cpsqlref/

DB2 Universal Database for iSeries

The following books are available at www.ibm.com/series/infocenter

- *DB2 Universal Database for iSeries Performance and Query Optimization*
- *DB2 Universal Database for iSeries Database Programming*
- *DB2 Universal Database for iSeries SQL Programming Concepts*
- *DB2 Universal Database for iSeries SQL Programming with Host Languages*
- *DB2 Universal Database for iSeries SQL Reference*
- *DB2 Universal Database for iSeries Distributed Data Management*
- *DB2 Universal Database for iSeries Distributed Database Programming*

DB2 Universal Database for Linux, UNIX, and Windows:

- *DB2 Universal Database Administration Guide: Planning*, SC09-4822
- *DB2 Universal Database Administration Guide: Implementation*, SC09-4820
- *DB2 Universal Database Administration Guide: Performance*, SC09-4821
- *DB2 Universal Database Administrative API Reference*, SC09-4824
- *DB2 Universal Database Application Development Guide: Building and Running Applications*, SC09-4825

- *DB2 Universal Database Call Level Interface Guide and Reference, Volumes 1 and 2*, SC09-4849 and SC09-4850
- *DB2 Universal Database Command Reference*, SC09-4828
- *DB2 Universal Database SQL Reference Volume 1*, SC09-4844
- *DB2 Universal Database SQL Reference Volume 2*, SC09-4845

Device Support Facilities

- *Device Support Facilities User's Guide and Reference*, GC35-0033

DFSMS

These books provide information about a variety of components of DFSMS, including z/OS DFSMS, z/OS DFSMSdftp, z/OS DFSMSdss, z/OS DFSMSshsm, and z/OS DFP.

- *z/OS DFSMS Access Method Services for Catalogs*, SC26-7394
- *z/OS DFSMSdss Storage Administration Guide*, SC35-0423
- *z/OS DFSMSdss Storage Administration Reference*, SC35-0424
- *z/OS DFSMSshsm Managing Your Own Data*, SC35-0420
- *z/OS DFSMSdftp: Using DFSMSdftp in the z/OS Environment*, SC26-7473
- *z/OS DFSMSdftp Diagnosis Reference*, GY27-7618
- *z/OS DFSMS: Implementing System-Managed Storage*, SC27-7407
- *z/OS DFSMS: Macro Instructions for Data Sets*, SC26-7408
- *z/OS DFSMS: Managing Catalogs*, SC26-7409
- *z/OS MVS: Program Management User's Guide and Reference*, SA22-7643
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
- *z/OS DFSMSdftp Storage Administration Reference*, SC26-7402
- *z/OS DFSMS: Using Data Sets*, SC26-7410
- *DFSMSdftp Advanced Services*, SC26-7400
- *DFSMS/MVS: Utilities*, SC26-7414

DFSORT™

- *DFSORT Application Programming: Guide*, SC33-4035
- *DFSORT Installation and Customization*, SC33-4034

Distributed Relational Database Architecture

- *Open Group Technical Standard*; the Open Group presently makes the following DRDA books available through its Web site at www.opengroup.org
 - *Open Group Technical Standard, DRDA Version 3 Vol. 1: Distributed Relational Database Architecture*
 - *Open Group Technical Standard, DRDA Version 3 Vol. 2: Formatted Data Object Content Architecture*
 - *Open Group Technical Standard, DRDA Version 3 Vol. 3: Distributed Data Management Architecture*

Domain Name System

- *DNS and BIND, Third Edition, Paul Albitz and Cricket Liu, O'Reilly, ISBN 0-59600-158-4*

Education

- Information about IBM educational offerings is available on the Web at <http://www.ibm.com/software/sw-training/>
- A collection of glossaries of IBM terms is available on the IBM Terminology Web site at www.ibm.com/ibm/terminology/index.html

eServer zSeries®

- *IBM eServer zSeries Processor Resource/System Manager Planning Guide, SB10-7033*

Fortran: VS Fortran

- *VS Fortran Version 2: Language and Library Reference, SC26-4221*
- *VS Fortran Version 2: Programming Guide for CMS and MVS, SC26-4222*

High Level Assembler

- *High Level Assembler for MVS and VM and VSE Language Reference, SC26-4940*
- *High Level Assembler for MVS and VM and VSE Programmer's Guide, SC26-4941*

ICSF

- *z/OS ICSF Overview, SA22-7519*
- *Integrated Cryptographic Service Facility Administrator's Guide, SA22-7521*

IMS Version 8

IMS product information is available on the IMS Library Web page, which you can find at www.ibm.com/ims

- *IMS Administration Guide: System, SC27-1284*
- *IMS Administration Guide: Transaction Manager, SC27-1285*

- *IMS Application Programming: Database Manager, SC27-1286*
- *IMS Application Programming: Design Guide, SC27-1287*
- *IMS Application Programming: Transaction Manager, SC27-1289*
- *IMS Command Reference, SC27-1291*
- *IMS Customization Guide, SC27-1294*
- *IMS Install Volume 1: Installation Verification, GC27-1297*
- *IMS Install Volume 2: System Definition and Tailoring, GC27-1298*
- *IMS Messages and Codes Volumes 1 and 2, GC27-1301 and GC27-1302*
- *IMS Open Transaction Manager Access Guide and Reference, SC18-7829*
- *IMS Utilities Reference: System, SC27-1309*

General information about IMS Batch Terminal Simulator for z/OS is available on the Web at www.ibm.com/software/data/db2imstools/library.html

IMS DataPropagator

- *IMS DataPropagator for z/OS Administrator's Guide for Log, SC27-1216*
- *IMS DataPropagator: An Introduction, GC27-1211*
- *IMS DataPropagator for z/OS Reference, SC27-1210*

ISPF

- *z/OS ISPF Dialog Developer's Guide, SC23-4821*
- *z/OS ISPF Messages and Codes, SC34-4815*
- *z/OS ISPF Planning and Customizing, GC34-4814*
- *z/OS ISPF User's Guide Volumes 1 and 2, SC34-4822 and SC34-4823*

Language Environment

- *Debug Tool User's Guide and Reference, SC18-7171*
- *Debug Tool for z/OS and OS/390 Reference and Messages, SC18-7172*
- *z/OS Language Environment Concepts Guide, SA22-7567*
- *z/OS Language Environment Customization, SA22-7564*
- *z/OS Language Environment Debugging Guide, GA22-7560*
- *z/OS Language Environment Programming Guide, SA22-7561*
- *z/OS Language Environment Programming Reference, SA22-7562*

MQSeries

- *MQSeries Application Messaging Interface, SC34-5604*

- *MQSeries for OS/390 Concepts and Planning Guide*, GC34-5650
- *MQSeries for OS/390 System Setup Guide*, SC34-5651

National Language Support

- *National Language Design Guide Volume 1*, SE09-8001
- *IBM National Language Support Reference Manual Volume 2*, SE09-8002

NetView®

- *Tivoli NetView for z/OS Installation: Getting Started*, SC31-8872
- *Tivoli NetView for z/OS User's Guide*, GC31-8849

Microsoft ODBC

Information about Microsoft ODBC is available at <http://msdn.microsoft.com/library/>

Parallel Sysplex Library

- *System/390 9672 Parallel Transaction Server, 9672 Parallel Enterprise Server, 9674 Coupling Facility System Overview For R1/R2/R3 Based Models*, SB10-7033
- *z/OS Parallel Sysplex Application Migration*, SA22-7662
- *z/OS Parallel Sysplex Overview: An Introduction to Data Sharing and Parallelism*, SA22-7661
- *z/OS Parallel Sysplex Test Report*, SA22-7663

The *Parallel Sysplex Configuration Assistant* is available at www.ibm.com/s390/psa/psotool

PL/I: Enterprise PL/I for z/OS

- *IBM Enterprise PL/I for z/OS Language Reference*, SC27-1460
- *IBM Enterprise PL/I for z/OS Programming Guide*, SC27-1457

PL/I: PL/I for MVS & VM

- *PL/I for MVS & VM Programming Guide*, SC26-3113

SMP/E

- *SMP/E for z/OS and OS/390 Reference*, SA22-7772
- *SMP/E for z/OS and OS/390 User's Guide*, SA22-7773

Storage Management

- *z/OS DFSMS: Implementing System-Managed Storage*, SC26-7407
- *MVS/ESA Storage Management Library: Managing Data*, SC26-7397

- *MVS/ESA Storage Management Library: Managing Storage Groups*, SC35-0421
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide*, GC26-7398

System Network Architecture (SNA)

- *SNA Formats*, GA27-3136
- *SNA LU 6.2 Peer Protocols Reference*, SC31-6808
- *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084
- *SNA/Management Services Alert Implementation Guide*, GC31-6809

TCP/IP

- *IBM TCP/IP for MVS: Customization & Administration Guide*, SC31-7134
- *IBM TCP/IP for MVS: Diagnosis Guide*, LY43-0105
- *IBM TCP/IP for MVS: Messages and Codes*, SC31-7132
- *IBM TCP/IP for MVS: Planning and Migration Guide*, SC31-7189

TotalStorage™ Enterprise Storage Server

- *RAMAC Virtual Array: Implementing Peer-to-Peer Remote Copy*, SG24-5680
- *Enterprise Storage Server Introduction and Planning*, GC26-7444
- *IBM RAMAC Virtual Array*, SG24-6424

Unicode

- *z/OS Support for Unicode: Using Conversion Services*, SA22-7649

Information about Unicode, the Unicode consortium, the Unicode standard, and standards conformance requirements is available at www.unicode.org

VTAM

- *Planning for NetView, NCP, and VTAM*, SC31-8063
- *VTAM for MVS/ESA Diagnosis*, LY43-0078
- *VTAM for MVS/ESA Messages and Codes*, GC31-8369
- *VTAM for MVS/ESA Network Implementation Guide*, SC31-8370
- *VTAM for MVS/ESA Operation*, SC31-8372
- *z/OS Communications Server SNA Programming*, SC31-8829
- *z/OS Communications Server SNA Programmer's LU 6.2 Reference*, SC31-8810
- *VTAM for MVS/ESA Resource Definition Reference*, SC31-8377

WebSphere family

- *WebSphere MQ Integrator Broker: Administration Guide, SC34-6171*
- *WebSphere MQ Integrator Broker for z/OS: Customization and Administration Guide, SC34-6175*
- *WebSphere MQ Integrator Broker: Introduction and Planning, GC34-5599*
- *WebSphere MQ Integrator Broker: Using the Control Center, SC34-6168*

z/Architecture™

- *z/Architecture Principles of Operation, SA22-7832*

z/OS

- *z/OS C/C++ Programming Guide, SC09-4765*
- *z/OS C/C++ Run-Time Library Reference, SA22-7821*
- *z/OS C/C++ User's Guide, SC09-4767*
- *z/OS Communications Server: IP Configuration Guide, SC31-8875*
- *z/OS DCE Administration Guide, SC24-5904*
- *z/OS DCE Introduction, GC24-5911*
- *z/OS DCE Messages and Codes, SC24-5912*
- *z/OS Information Roadmap, SA22-7500*
- *z/OS Introduction and Release Guide, GA22-7502*
- *z/OS JES2 Initialization and Tuning Guide, SA22-7532*
- *z/OS JES3 Initialization and Tuning Guide, SA22-7549*
- *z/OS Language Environment Concepts Guide, SA22-7567*
- *z/OS Language Environment Customization, SA22-7564*
- *z/OS Language Environment Debugging Guide, GA22-7560*
- *z/OS Language Environment Programming Guide, SA22-7561*
- *z/OS Language Environment Programming Reference, SA22-7562*
- *z/OS Managed System Infrastructure for Setup User's Guide, SC33-7985*
- *z/OS MVS Diagnosis: Procedures, GA22-7587*
- *z/OS MVS Diagnosis: Reference, GA22-7588*
- *z/OS MVS Diagnosis: Tools and Service Aids, GA22-7589*
- *z/OS MVS Initialization and Tuning Guide, SA22-7591*
- *z/OS MVS Initialization and Tuning Reference, SA22-7592*
- *z/OS MVS Installation Exits, SA22-7593*
- *z/OS MVS JCL Reference, SA22-7597*
- *z/OS MVS JCL User's Guide, SA22-7598*
- *z/OS MVS Planning: Global Resource Serialization, SA22-7600*
- *z/OS MVS Planning: Operations, SA22-7601*

- *z/OS MVS Planning: Workload Management, SA22-7602*
- *z/OS MVS Programming: Assembler Services Guide, SA22-7605*
- *z/OS MVS Programming: Assembler Services Reference, Volumes 1 and 2, SA22-7606 and SA22-7607*
- *z/OS MVS Programming: Authorized Assembler Services Guide, SA22-7608*
- *z/OS MVS Programming: Authorized Assembler Services Reference Volumes 1-4, SA22-7609, SA22-7610, SA22-7611, and SA22-7612*
- *z/OS MVS Programming: Callable Services for High-Level Languages, SA22-7613*
- *z/OS MVS Programming: Extended Addressability Guide, SA22-7614*
- *z/OS MVS Programming: Sysplex Services Guide, SA22-7617*
- *z/OS MVS Programming: Sysplex Services Reference, SA22-7618*
- *z/OS MVS Programming: Workload Management Services, SA22-7619*
- *z/OS MVS Recovery and Reconfiguration Guide, SA22-7623*
- *z/OS MVS Routing and Descriptor Codes, SA22-7624*
- *z/OS MVS Setting Up a Sysplex, SA22-7625*
- *z/OS MVS System Codes SA22-7626*
- *z/OS MVS System Commands, SA22-7627*
- *z/OS MVS System Messages Volumes 1-10, SA22-7631, SA22-7632, SA22-7633, SA22-7634, SA22-7635, SA22-7636, SA22-7637, SA22-7638, SA22-7639, and SA22-7640*
- *z/OS MVS Using the Subsystem Interface, SA22-7642*
- *z/OS Planning for Multilevel Security and the Common Criteria, SA22-7509*
- *z/OS RMF User's Guide, SC33-7990*
- *z/OS Security Server Network Authentication Server Administration, SC24-5926*
- *z/OS Security Server RACF Auditor's Guide, SA22-7684*
- *z/OS Security Server RACF Command Language Reference, SA22-7687*
- *z/OS Security Server RACF Macros and Interfaces, SA22-7682*
- *z/OS Security Server RACF Security Administrator's Guide, SA22-7683*
- *z/OS Security Server RACF System Programmer's Guide, SA22-7681*
- *z/OS Security Server RACROUTE Macro Reference, SA22-7692*
- *z/OS Support for Unicode: Using Conversion Services, SA22-7649*
- *z/OS TSO/E CLISTs, SA22-7781*
- *z/OS TSO/E Command Reference, SA22-7782*

- *z/OS TSO/E Customization*, SA22-7783
- *z/OS TSO/E Messages*, SA22-7786
- *z/OS TSO/E Programming Guide*, SA22-7788
- *z/OS TSO/E Programming Services*, SA22-7789
- *z/OS TSO/E REXX Reference*, SA22-7790
- *z/OS TSO/E User's Guide*, SA22-7794
- *z/OS UNIX System Services Command Reference*, SA22-7802
- *z/OS UNIX System Services Messages and Codes*, SA22-7807
- *z/OS UNIX System Services Planning*, GA22-7800
- *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803
- *z/OS UNIX System Services User's Guide*, SA22-7801

Index

Special characters

- _ (underscore)
 - assembler host variable 145
- :
- : (colon)
 - assembler host variable 148
 - C host variable 162
 - COBOL 192
 - Fortran host variable 223
 - PL/I host variable 234
- ' (apostrophe)
 - string delimiter precompiler option 484

A

- abend
 - before commit point 435
 - DB2 869, 901
 - effect on cursor position 123
 - exit routines 884
 - for synchronization calls 576
 - IMS
 - U0102 581
 - U0775 437
 - U0778 438, 439
 - multiple-mode program 434
 - program 432
 - reason codes 885
 - return code posted to CAF CONNECT 871
 - return code posted to RRSF CONNECT 904
 - single-mode program 434
 - system
 - X"04E" 573
- ABRT parameter of CAF (call attachment facility) 877, 886
- access path
 - affects lock attributes 424
 - direct row access 270, 803
 - index-only access 802
 - low cluster ratio
 - suggests table space scan 809
 - with list prefetch 832
 - multiple index access
 - description 812
 - PLAN_TABLE 801
 - selection
 - influencing with SQL 774
 - problems 731
 - queries containing host variables 759
 - Visual Explain 775, 789
 - table space scan 809
 - unique index with matching value 814
- ACQUIRE
 - option of BIND PLAN subcommand
 - locking tables and table spaces 408
- activity sample table 993
- address space
 - examples 932
 - initialization
 - CAF CONNECT command 873
 - CAF OPEN command 875
 - sample scenarios 882
- address space (*continued*)
 - separate tasks 862, 894
 - termination
 - CAF CLOSE command 878
 - CAF DISCONNECT command 879
- ALL quantified predicate 52
- ALLOCATE CURSOR statement 710
- ambiguous cursor 419
- AMODE link-edit option 495, 554
- ANY quantified predicate 52
- APOST precompiler option 484
- application plan
 - binding 498
 - creating 495
 - dynamic plan selection for CICS applications 507
 - invalidated 389
 - listing packages 498
 - rebinding 388
 - using packages 384
- application program
 - bill of materials 1097
 - coding SQL statements
 - assembler 143
 - coding conventions 78
 - data declarations 131
 - data entry 30
 - description 77
 - dynamic SQL 593, 627
 - host variables 80
 - selecting rows using a cursor 103
- design considerations
 - bind 381
 - CAF 862
 - checkpoint 575
 - IMS calls 575
 - planning for changes 386
 - precompile 381
 - programming for DL/I batch 574
 - RRSAF 894
 - SQL statements 575
 - stored procedures 629
 - structure 857
 - synchronization call abends 576
 - using ISPF (interactive system productivity facility) 518
 - XRST call 575
- external stored procedures 641
- object extensions 295
- preparation
 - assembling 494
 - binding 384, 495
 - compiling 494
 - DB2 precompiler option defaults 492
 - defining to CICS 494
 - DRDA access 445
 - example 522
 - link-editing 494
 - precompiler option defaults 473
 - precompiler options 383
 - preparing for running 471
 - program preparation panel 518

- application program (*continued*)
 - preparation (*continued*)
 - using DB2 coprocessor 382
 - using DB2 precompiler 381
 - using DB2I (DB2 Interactive) 518
 - running
 - CAF (call attachment facility) 863
 - CICS 512
 - IMS 511
 - program synchronization in DL/I batch 575
 - TSO 508
 - TSO CLIST 511
 - suspension
 - description 394
 - test environment 557
 - testing 557
- arithmetic expressions in UPDATE statement 37
- AS clause
 - naming columns for view 7
 - naming columns in union 8
 - naming derived columns 8
 - naming result columns 7
 - ORDER BY name 7
 - with ORDER BY clause 10
- ASCII data, retrieving 619
- assembler application program
 - assembling 494
 - character host variable 149
 - coding SQL statements 143
 - data type compatibility 155
 - data types 151
 - declaring tables 145
 - declaring views 145
 - fixed-length character string 149
 - graphic host variable 150
 - host variable
 - declaring 148
 - naming convention 145
 - INCLUDE statement 145
 - including SQLCA 144
 - including SQLDA 144
 - indicator variable 156
 - LOB variable 151
 - numeric host variable 149
 - reentrant 146
 - result set locator 150
 - ROWID variable 151
 - SQLCODE 143
 - SQLSTATE 143
 - table locator 150
 - variable declaration 154
 - varying-length character string 149
- assignment, compatibility rules 4
- ASSOCIATE LOCATORS statement 709
- ATTACH option
 - CAF 867
 - precompiler 867, 900
 - RRSAF 900
- ATTACH precompiler option 484
- attention processing 862, 883, 894
- AUTH SIGNON, RRSAF
 - syntax 914
 - usage 914
- authority
 - authorization ID 510
 - creating test tables 559
 - SYSIBM.SYSTABAUTH table 18

- AUTOCOMMIT field of SPUFI panel 61
- automatic
 - rebind
 - EXPLAIN processing 799
- automatic query rewrite 267
- automatic rebind
 - conditions for 390
 - invalid plan or package 389
 - SQLCA not available 391
- auxiliary table
 - LOCK TABLE statement 428

B

- batch processing
 - access to DB2 and DL/I together
 - binding a plan 578
 - checkpoint calls 575
 - commits 575
 - precompiling 578
 - batch DB2 application
 - running 510
 - starting with a CLIST 511
- bill of materials applications 1097
- binary large object (BLOB) 297
- BIND PACKAGE subcommand of DSN
 - options
 - CURRENTDATA 447
 - DBPROTOCOL 447
 - ENCODING 447
 - ISOLATION 412
 - KEEPDYNAMIC 600
 - location-name 446
 - OPTIONS 447
 - RELEASE 408
 - REOPT(ALWAYS) 760
 - REOPT(NONE) 760
 - REOPT(ONCE) 760
 - SQLERROR 446
 - options associated with DRDA access 446, 447
 - remote 496
- BIND PLAN subcommand of DSN
 - options
 - ACQUIRE 408
 - CACHESIZE 504
 - CURRENTDATA 446
 - DBPROTOCOL 446
 - DISCONNECT 445
 - ENCODING 446
 - ISOLATION 412
 - KEEPDYNAMIC 600
 - RELEASE 408
 - REOPT(ALWAYS) 760
 - REOPT(NONE) 760
 - REOPT(ONCE) 760
 - SQLRULES 446, 505
 - options associated with DRDA access 445
 - remote 496
- binding
 - advantages of packages 385
 - application plans 495
 - changes that require 386
 - checking BIND PACKAGE options 447
 - DBRMs precompiled elsewhere 478
 - method
 - DBRMs and package list 385
 - DBRMs into single plan 385

- binding (*continued*)
 - method (*continued*)
 - package list only 384
 - options associated with DRDA access 446
 - options for 384
 - packages
 - deciding how to use 384
 - in use 384
 - remote 496
 - planning for 384
 - plans
 - in use 384
 - including DBRMs 498
 - including packages 498
 - options 498
 - remote package requirements 496
 - specify SQL rules 505
- block fetch
 - conditions for use by non-scrollable cursor 459
 - conditions for use by scrollable cursor 459
 - preventing 467
 - using 458
 - with cursor stability 467
- BMP (batch message processing) program
 - checkpoints 436, 437
 - transaction-oriented 436
- BTS (batch terminal simulator) 562

C

- C application program
 - declaring tables 160
 - LOB variable 167
 - LOB variable array 172
 - numeric host variable 162
 - numeric host variable array 168
 - result set locator 166
 - sample application 1013
- C/C++ application program
 - character host variable 162
 - character host variable array 169
 - coding considerations 186
 - coding SQL statements 158
 - constants 180
 - data type compatibility 181
 - data types 175
 - DB2 coprocessor 478
 - DCLGEN support 136
 - declaring views 160
 - examples 1043
 - fixed-length string 182
 - graphic host variable 164
 - graphic host variable array 171
 - host variable array, declaring 161
 - host variable, declaring 161
 - INCLUDE statement 161
 - including SQLCA 159
 - including SQLDA 160
 - indicator variable 182
 - indicator variable array 182
 - naming convention 161
 - precompiler option defaults 492
 - ROWID variable 168
 - ROWID variable array 173
 - SQLCODE 159
 - SQLSTATE 159
 - table locator 167
- C/C++ application program (*continued*)
 - variable declaration 179
 - varying-length string 182
 - with classes, preparing 517
- C++ application program
 - DB2 coprocessor 479
- cache
 - dynamic SQL
 - effect of RELEASE(DEALLOCATE) 409
- cache (dynamic SQL)
 - statements 596
- CACHESIZE
 - option of BIND PLAN subcommand 504
 - REBIND subcommand 504
- CAF (call attachment facility)
 - application program
 - examples 886
 - preparation 862
 - connecting to DB2 886
 - description 861
 - function descriptions 869
 - load module structure 864
 - parameters 869
 - programming language 862
 - register conventions 869
 - restrictions 861
 - return codes
 - checking 888
 - CLOSE 877
 - CONNECT 871
 - DISCONNECT 878
 - OPEN 875
 - TRANSLATE 880
 - run environment 863
 - running an application program 863
- calculated values
 - groups with conditions 11
 - summarizing group values 11
- call attachment facility (CAF) 861
- CALL DSNALI statement 869, 881
- CALL DSNRLI statement 903
- CALL statement
 - example 682
 - SQL procedure 660
- cardinality of user-defined table function
 - improving query performance 779
- Cartesian join 819
- CASE statement (SQL procedure) 660
- catalog statistics
 - influencing access paths 785
- catalog table
 - accessing 17
 - SYSIBM.LOCATIONS 451
 - SYSIBM.SYSCOLUMNS 18
 - SYSIBM.SYSTABAUTH 18
- CCSID (coded character set identifier)
 - controlling in COBOL programs 213
 - effect of DECLARE VARIABLE 85
 - host variable 85
 - precompiler option 485
 - SQLDA 618
- character host variable
 - assembler 149
 - C 162
 - COBOL 194
 - Fortran 224
 - PL/I 235

- character host variable array
 - C 169
 - COBOL 200
 - PL/I 238
- character large object (CLOB) 297
- character string
 - literals 78
 - mixed data 4
 - width of column in results 64, 70
- check constraint
 - check integrity 260
 - considerations 259
 - CURRENT RULES special register effect 260
 - defining 259
 - description 259
 - determining violations 989
 - enforcement 260
 - programming considerations 989
- CHECK-pending status 260
- checkpoint
 - calls 434, 436
 - frequency 438
- CHKP call, IMS 433, 435
- CICS
 - attachment facility
 - controlling from applications 939
 - programming considerations 939
 - DSNTIAC subroutine
 - assembler 158
 - C 186
 - COBOL 219
 - PL/I 249
 - environment planning 512
 - facilities
 - command language translator 493
 - control areas 557
 - EDF (execution diagnostic facility) 563
 - language interface module (DSNCLI)
 - use in link-editing an application 495
 - logical unit of work 432
 - operating
 - indoubt data 433
 - running a program 557
 - system failure 433
 - preparing with JCL procedures 516
 - programming
 - DFHEIENT macro 148
 - sample applications 1015, 1018
 - SYNCPOINT command 433
 - storage handling
 - assembler 158
 - C 186
 - COBOL 219
 - PL/I 249
 - sync point 433
 - thread reuse 939
 - unit of work 432
- claim
 - effect of cursor WITH HOLD 421
- CLOSE
 - statement
 - description 108
 - recommendation 113
 - WHENEVER NOT FOUND clause 612, 623
- CLOSE (connection function of CAF)
 - description 866
 - language examples 878
- CLOSE (connection function of CAF) (*continued*)
 - program example 886
 - syntax 877
 - syntax usage 877
- cluster ratio
 - effects
 - table space scan 809
 - with list prefetch 832
- COALESCE function 42
- COBOL application program
 - assignment rules 213
 - character host variable 194
 - fixed-length string 194
 - varying-length string 194
 - character host variable array 200
 - fixed-length string array 200
 - varying-length string array 201
- CODEPAGE compiler option 213
- coding SQL statements 186
- compiling 494
- controlling CCSID 213
- data type compatibility 214
- data types 210
- DB2 coprocessor 480
- DB2 precompiler option defaults 492
- DCLGEN support 136
- declaring tables 188
- declaring views 188
- defining the SQLDA 187
- dynamic SQL 627
- FILLER entry name 214
- graphic host variable 195
- graphic host variable array 203
- host variable
 - declaring 191
 - use of hyphens 191
- host variable array
 - declaring 191
- INCLUDE statement 189
- including SQLCA 186
- indicator variable 216
- indicator variable array 216
- LOB variable 198
- LOB variable array 204
- naming convention 189
- numeric host variable 192
- numeric host variable array 199
- object-oriented extensions 219
- options 189, 190
- preparation 494
- record description from DCLGEN 136
- resetting SQL-INIT-FLAG 191
- result set locator 197
- ROWID variable 198
- ROWID variable array 205
- sample program 1031
- SQLCODE 186
- SQLSTATE 186
- table locator 198
- variable declaration 212
- WHENEVER statement 189
 - with classes, preparing 517

- CODEPAGE compiler option 213
- coding SQL statements
- assembler 143
- C 158
- C++ 158

- coding SQL statements (*continued*)
 - COBOL 186
 - dynamic 593
 - Fortran 220
 - PL/I 230
 - REXX 249
- collection, package
 - identifying 499
 - SET CURRENT PACKAGESET statement 499
- colon
 - assembler host variable 148
 - C host variable 162
 - C host variable array 162
 - COBOL host variable 192
 - Fortran host variable 223
 - PL/I host variable 234
 - PL/I host variable array 234
 - preceding a host variable 80
 - preceding a host variable array 86
- column
 - data types 4
 - default value
 - system-defined 20
 - user-defined 20
 - displaying, list of 18
 - heading created by SPUFI 71
 - labels
 - DCLGEN 133
 - labels, usage 621
 - name
 - as a host variable 134
 - name, with UPDATE statement 36
 - retrieving, with SELECT 5
 - specified in CREATE TABLE 19
 - width of results 64, 70
- COMMA precompiler option 485
- commit point
 - description 432
 - IMS unit of work 434
 - lock release 434
- COMMIT statement
 - description 61
 - ending unit of work 432
 - in a stored procedure 646
 - when to issue 432
- commit, using RRSAP 895
- common table expressions
 - description 13
 - examples 1097
 - in a CREATE VIEW statement 14
 - in a SELECT statement 14
 - in an INSERT statement 15
 - infinite loops 16
 - recursion 1097
- comparison
 - compatibility rules 4
 - operator, subquery 51
- compatibility
 - data types 4
 - locks 406
 - rules 4
- composite key 262
- compound statement
 - example
 - dynamic SQL 668
 - nested IF and WHILE statements 667
 - EXIT handler 663
- compound statement (*continued*)
 - labels 662
 - SQL procedure 660
 - valid SQL statements 660
- concurrency
 - control by locks 394
 - description 393
 - effect of
 - ISOLATION options 415, 416
 - lock size 404
 - uncommitted read 414
 - recommendations 397
- CONNECT
 - statement
 - SPUFI 61
- CONNECT (connection function of CAF)
 - description 866
 - language examples 875
 - program example 886
 - syntax 871
- CONNECT (Type 1) statement 453
- CONNECT LOCATION field of SPUFI panel 61
- CONNECT precompiler option 485
- CONNECT statement, with DRDA access 450
- connection
 - DB2
 - connecting from tasks 857
 - function of CAF
 - CLOSE 877, 886
 - CONNECT 871, 875, 886
 - description 864
 - DISCONNECT 878, 886
 - OPEN 875, 886
 - sample scenarios 882, 883
 - summary of behavior 881
 - TRANSLATE 880, 888
 - function of RRSAP
 - AUTH SIGNON 914
 - CREATE THREAD 936
 - description 896
 - examples 932
 - IDENTIFY 904, 936
 - SIGNON 909, 936
 - summary of behavior 902
 - TERMINATE IDENTIFY 930, 936
 - TERMINATE THREAD 928, 936
 - TRANSLATE 931
- constants, syntax
 - C 180
 - Fortran 227
- CONTINUE clause of WHENEVER statement 93
- CONTINUE handler (SQL procedure)
 - description 663
 - example 663
- correlated reference
 - correlation name 55
 - SQL rules 47
 - usage 47
 - using in subquery 54
- correlated subqueries 766
- correlation name 55
- CREATE GLOBAL TEMPORARY TABLE statement 22
- CREATE TABLE statement
 - DEFAULT clause 20
 - NOT NULL clause 20
 - PRIMARY KEY clause 262
 - relationship names 264

CREATE TABLE statement (*continued*)

- UNIQUE clause 20, 262
- usage 19

CREATE THREAD, RRSAF

- description 897
- effect of call order 902
- implicit connection 897
- language examples 928
- program example 936
- syntax usage 926

CREATE TRIGGER

- activation order 287
- description 277
- example 277
- timestamp 287
- trigger naming 279

CREATE VIEW statement 26

- created temporary table
- instances 22
- table space scan 809
- use of NOT NULL 22
- working with 21

CS (cursor stability)

- optimistic concurrency control 412
- page and row locking 412

CURRENTDATA option of BIND

- plan and package options differ 420

CURRENT PACKAGESET special register

- dynamic plan switching 507
- identify package collection 499

CURRENT RULES special register

- effect on check constraints 260
- usage 505

CURRENT SERVER special register

- description 499
- saving value in application program 468

CURRENT SQLID special register

- use in test 557
- value in INSERT statement 20

cursor

- ambiguous 419
- attributes
 - using GET DIAGNOSTICS 116
 - using SQLCA 116
- closing 108
 - CLOSE statement 113
- deleting a current row 112
- description 103
- dynamic scrollable 115
- effect of abend on position 123
- example
 - retrieving backward with scrollable cursor 125
 - updating specific row with rowset-positioned cursor 127
 - updating with non-scrollable cursor 124
 - updating with rowset-positioned cursor 126
- insensitive scrollable 114
- maintaining position 122
- non-scrollable 113
- open state 122
- OPEN statement 105
- result table 103
- row-positioned
 - declaring 103
 - deleting a current row 107
 - description 103
 - end-of-data condition 105

cursor (*continued*)

row-positioned (*continued*)

- retrieving a row of data 106
- steps in using 103
- updating a current row 107

rowset-positioned

- declaring 108
- description 103
- end-of-data condition 109
- number of rows 109
- number of rows in rowset 113
- opening 108
- retrieving a rowset of data 109
- steps in using 108
- updating a current rowset 111

scrollable

- description 114
- dynamic 115
- fetch orientation 117
- INSENSITIVE 114
- retrieving rows 116
- SENSITIVE DYNAMIC 115
- SENSITIVE STATIC 114
- sensitivity 114
- static 115
- updatable 114

static scrollable 115

types 113

WITH HOLD

- claims 421
- description 122
- locks 420

D

data

- adding to the end of a table 988
- associated with WHERE clause 8
- currency 467
- effect of locks on integrity 394
- improving access 789
- indoubt state 435
- not in a table 16
- retrieval using SELECT * 987
- retrieving a rowset 109
- retrieving a set of rows 106
- retrieving large volumes 987
- scrolling backward through 983
- security and integrity 431
- understanding access 789
- updating during retrieval 986
- updating previously retrieved data 986

data encryption 266

data type

- built-in 4
- comparisons 85
- compatibility
 - assembler and SQL 151
 - assembler application program 155
 - C and SQL 175
 - C application program 181
 - COBOL and SQL 210, 214
 - Fortran and SQL 225, 227
 - PL/I and SQL 241
 - PL/I application program 245
 - REXX and SQL 254
- result set locator 710

- DATE precompiler option 486
- datetime data type 4
- DB2abend
 - CAF 869
 - DL/I batch 576
 - RRSAF 901
- DB2 coprocessor
 - for C 478
 - for C++ 479
 - for COBOL 480
 - for PL/I 481
 - processing SQL statements 473
- DB2 private protocol access
 - coding an application 448
 - compared to DRDA access 442
 - mixed environment 1113
 - planning 442
 - sample program 1069
- DB2I (DB2 Interactive)
 - background processing
 - run-time libraries 527
 - EDITJCL processing
 - run-time libraries 527
 - help system 518
 - interrupting 68
 - menu 59
 - panels
 - BIND PACKAGE 533
 - BIND PLAN 536
 - Compile, Link, and Run 553
 - Current SPUFI Defaults 62
 - DB2I Primary Option Menu 59, 519
 - DCLGEN 132, 139
 - Defaults for BIND PLAN 546
 - Precompile 528
 - Program Preparation 522
 - System Connection Types 550
 - preparing programs 518
 - program preparation example 522
 - selecting
 - DCLGEN (declarations generator) 136
 - SPUFI 59
 - SPUFI 59
- DBCS (double-byte character set)
 - table names 131
 - translation in CICS 493
- DBINFO
 - stored procedure 636
 - user-defined function 330
- DBPROTOCOL(DRDA) 456
- DBRM (database request module)
 - binding to a package 496
 - binding to a plan 498
 - deciding how to bind 384
 - description 477
- DCLGEN subcommand of DSN
 - building data declarations 131
 - COBOL example 138
 - column labels 133
 - DBCS table names 131
 - forming host variable names 134
 - identifying tables 131
 - INCLUDE statement 136
 - including declarations in a program 136
 - indicator variable array declaration 134
 - right margin 135
 - starting 131
- DCLGEN subcommand of DSN (*continued*)
 - using 131
 - using PDS (partitioned data set) 131
- DDITV02 input data set 576
- DDOTV02 output data set 578
- deadlock
 - description 395
 - example 395
 - indications
 - in CICS 397
 - in IMS 397
 - in TSO 396
 - recommendation for avoiding 399
 - with RELEASE(DEALLOCATE) 400
 - X'00C90088' reason code in SQLCA 396
- debugging application programs 560
- DEC15
 - precompiler option 486
 - rules 16
- DEC31
 - avoiding overflow 17
 - precompiler option 486
 - rules 16
- decimal
 - 15 digit precision 16
 - 31 digit precision 17
 - arithmetic 16
- DECIMAL
 - constants 180
 - data type, in C 179
 - function, in C 179
- declaration
 - generator (DCLGEN) 131
 - in an application program 136
 - variables in CAF program examples 891
- DECLARE (SQL procedure) 661
- DECLARE CURSOR statement
 - description, row-positioned 103
 - description, rowset-positioned 108
 - FOR UPDATE clause 104
 - multilevel security 104
 - prepared statement 611, 615
 - scrollable cursor 114
 - WITH HOLD clause 122
 - WITH RETURN option 650
 - WITH ROWSET POSITIONING clause 108
- DECLARE GLOBAL TEMPORARY TABLE statement 23
- DECLARE TABLE statement
 - advantages of using 79
 - assembler 145
 - C 160
 - COBOL 188
 - description 79
 - Fortran 222
 - PL/I 231
 - table description 131
- DECLARE VARIABLE statement
 - changing CCSID 86
 - coding 86
 - description 85
- declared temporary table
 - including column defaults 24
 - including identity columns 23
 - instances 23
 - ON COMMIT clause 25
 - qualifier for 23
 - remote access using a three-part name 449

- declared temporary table (*continued*)
 - requirements 23
 - working with 21
- dedicated virtual memory pool 828
- DEFER(PREPARE) 456
- DELETE statement
 - correlated subquery 56
 - description 37
 - positioned
 - FOR ROW n OF ROWSET clause 112
 - restrictions 107
 - WHERE CURRENT clause 107, 112
 - subquery 51
- deleting
 - current rows 107
 - data 37
 - every row from a table 38
 - rows from a table 37
- delimiter, SQL 78
- department sample table
 - creating 20
 - description 994
- DESCRIBE CURSOR statement 710
- DESCRIBE INPUT statement 610
- DESCRIBE PROCEDURE statement 709
- DESCRIBE statement
 - column labels 621
 - INTO clauses 615, 617
- DFHEIENT macro 148
- DFSLI000 (IMS language interface module) 495
- direct row access 270, 803
- DISCONNECT (connection function of CAF)
 - description 866
 - language examples 879
 - program example 886
 - syntax 878
 - syntax usage 878
- displaying
 - table columns 18
 - table privileges 18
- DISTINCT
 - clause of SELECT statement 7
 - unique values 7
- distinct type
 - assigning values 369
 - comparing types 368
 - description 367
 - example
 - argument of user-defined function (UDF) 372
 - arguments of infix operator 372
 - casting constants 372
 - casting function arguments 372
 - casting host variables 372
 - LOB data type 372
 - function arguments 371
 - strong typing 368
 - UNION of 371
- distributed data
 - choosing an access method 442
 - coordinating updates 452
 - copying a remote table 467
 - DBPROTOCOL bind option 444, 449
 - encoding scheme of retrieved data 466
 - example
 - accessing remote temporary table 449
 - calling stored procedure at remote location 444
 - connecting to remote server 444, 450
- distributed data (*continued*)
 - example (*continued*)
 - limiting number of retrieved rows 464
 - specifying location in table name 444
 - using alias for multiple sites 451
 - using RELEASE statement 451
 - using three-part table names 448
 - executing long SQL statements 466
 - identifying server at run time 468
 - LOB performance
 - setting CURRENT RULES special register 455
 - using LOB locators 455
 - using stored procedure result sets 455
 - maintaining data currency 467
 - moving from DB2 private protocol access to DRDA access 442
 - performance
 - choosing bind options 456
 - coding efficient queries 454
 - forcing block fetch 458
 - limiting number of retrieved rows 461, 464
 - optimizing access path 457
 - specifying package list 456
 - using block fetch 458
 - using DRDA 458
 - performance considerations 456
 - planning
 - access by a program 441
 - DB2 private protocol access 444
 - DRDA access 444
 - program preparation 447
 - programming
 - coding with DB2 private protocol access 448
 - coding with DRDA access 448
 - resource limit facility 441
 - restricted systems 453
 - retrieving from ASCII or Unicode tables 466
 - savepoints 442
 - scrollable cursors 442
 - terminology 441
 - three-part table names 448
 - transmitting mixed data 467
 - two-phase commit 452
 - using alias for location 451
- DL/I batch
 - application programming 574
 - checkpoint call 433
 - checkpoint ID 582
 - commit and rollback coordination 438
 - DB2 requirements 574
 - DDITV02 input data set 576
 - DSNMTV01 module 579
 - features 573
 - SSM= parameter 579
 - submitting an application 579
 - TERM call 433
- double-byte character large object (DBCLOB) 297
- DPSI
 - performance considerations 772
- DRDA access
 - accessing remote temporary table 449
 - bind options 445, 446
 - coding an application 448
 - compared to DB2 private protocol access 442
 - connecting to remote server 450
 - mixed environment 1113
 - planning 442, 444

- DRDA access (*continued*)
 - precompiler options 445
 - preparing programs 445
 - programming hints 465
 - releasing connections 451
 - sample program 1061
 - SQL limitations at different servers 465
- DROP TABLE statement 25
- DSN applications, running with CAF 863
- DSN command of TSO
 - return code processing 509
 - RUN subcommands 508
 - services lost under CAF 863
- DSN_FUNCTION_TABLE table 361
- DSN_STATEMENT_CACHE_TABLE 599
- DSN_STATEMENT_TABLE table
 - column descriptions 843
- DSN8BC3 sample program 218
- DSN8BD3 sample program 185
- DSN8BE3 sample program 185
- DSN8BF3 sample program 230
- DSN8BP3 sample program 248
- DSNACICS stored procedure
 - debugging 1139
 - description 1131
 - invocation example 1137
 - invocation syntax 1132
 - output 1138
 - parameter descriptions 1133
 - restrictions 1139
- DSNACICX user exit routine
 - description 1135
 - parameter list 1136
 - rules for writing 1135
- DSNAEXP stored procedure
 - description 1143
 - example call 1145
 - option descriptions 1144
 - output 1146
 - syntax diagram 1144
- DSNAIMS
 - option descriptions 1140
 - syntax diagram 1139
- DSNAIMS stored procedure
 - description 1139
 - examples 1142
- DSNALI (CAF language interface module)
 - deleting 886
 - loading 886
- DSNCLI (CICS language interface module) 495
- DSNELI (TSO language interface module) 863
- DSNH command of TSO 567
- DSNHASM procedure 513
- DSNHC procedure 513
- DSNHCOB procedure 513
- DSNHCOB2 procedure 513
- DSNHCPP procedure 513
- DSNHCPP2 procedure 513
- DSNHDECP
 - implicit CAF connection 866
 - implicit RRSAP connection 897
- DSNHFOR procedure 513
- DSNHICB2 procedure 513
- DSNHICOB procedure 513
- DSNHILI entry point to DSNALI
 - implicit calls 866
 - program example 890
- DSNHILI entry point to DSNRLI
 - implicit calls 897
 - program example 935
- DSNHILI2 entry point to DSNALI 888
- DSNHPLI procedure 513
- DSNMTV01 module 579
- DSNRLI (RRSAF language interface module)
 - deleting 935
 - loading 935
- DSNTEDIT CLIST 1103
- DSNTEP2 and DSNTEP4 sample program
 - specifying SQL terminator 1026
- DSNTEP2 sample program
 - how to run 1019
 - parameters 1020
 - program preparation 1019
- DSNTEP4 sample program
 - how to run 1019
 - parameters 1020
 - program preparation 1019
- DSNTIAC subroutine
 - assembler 158
 - C 186
 - COBOL 219
 - PL/I 249
- DSNTIAD sample program
 - how to run 1019
 - parameters 1020
 - program preparation 1019
 - specifying SQL terminator 1024
- DSNTIAR subroutine
 - assembler 157
 - C 184
 - COBOL 217
 - description 98
 - Fortran 229
 - PL/I 247
 - return codes 100
 - using 100
- DSNTIAUL sample program
 - how to run 1019
 - parameters 1020
 - program preparation 1019
- DSNTIR subroutine 229
- DSNTPSMP stored procedure 672
- DSNTRACE data set 884
- DSNXDBRM 477
- DSNXNBRM 477
- duration of locks
 - controlling 408
 - description 404
- DXMQGEN stored procedure
 - description 1165
 - invocation example 1168
 - invocation syntax 1166
 - output 1169
 - parameter descriptions 1166
- DXMQGENCLOB stored procedure
 - description 1173
 - invocation example 1175
 - invocation syntax 1173
 - output 1176
 - parameter descriptions 1174
- DXMQINSERT stored procedure
 - description 1146
 - invocation example 1147
 - invocation syntax 1147

DXXMQINSERT stored procedure *(continued)*
 output 1148
 parameter descriptions 1147
 DXXMQINSERTALL stored procedure
 description 1156
 invocation example 1157
 invocation syntax 1156
 output 1158
 parameter descriptions 1157
 DXXMQINSERTALLCLOB stored procedure
 description 1163
 invocation example 1165
 invocation syntax 1164
 output 1165
 parameter descriptions 1164
 DXXMQINSERTCLOB stored procedure
 description 1151
 invocation example 1152
 invocation syntax 1151
 output 1153
 parameter descriptions 1152
 DXXMQRETRIEVE stored procedure
 description 1169
 invocation example 1171
 invocation syntax 1169
 output 1173
 parameter descriptions 1170
 DXXMQRETRIEVECLOB stored procedure
 description 1176
 invocation example 1179
 invocation syntax 1177
 output 1180
 parameter descriptions 1177
 DXXMQSHRED stored procedure
 description 1148
 invocation example 1150
 invocation syntax 1149
 output 1151
 parameter descriptions 1149
 DXXMQSHREDALL stored procedure
 description 1158
 invocation example 1160
 invocation syntax 1159
 output 1160
 parameter descriptions 1159
 DXXMQSHREDALLCLOB stored procedure
 description 1161
 invocation example 1162
 invocation syntax 1161
 output 1163
 parameter descriptions 1161
 DXXMQSHREDCLOB stored procedure
 description 1153
 invocation example 1155
 invocation syntax 1154
 output 1156
 parameter descriptions 1154
 DYNAM option of COBOL 189
 dynamic plan selection
 restrictions with CURRENT PACKAGESET special
 register 507
 using packages with 507
 dynamic prefetch
 description 831
 dynamic SQL
 advantages and disadvantages 593
 assembler program 614

dynamic SQL *(continued)*
 C program 614
 caching
 effect of RELEASE bind option 409
 caching prepared statements 596
 COBOL application program 189
 COBOL program 627
 description 593
 effect of bind option REOPT(ALWAYS) 625
 effect of WITH HOLD cursor 607
 EXECUTE IMMEDIATE statement 604
 fixed-list SELECT statements 610, 613
 Fortran program 222
 host languages 603
 non-SELECT statements 603, 607
 PL/I 614
 PREPARE and EXECUTE 605, 607
 programming 593
 requirements 594
 restrictions 594
 sample C program 1043
 statement caching 596
 statements allowed 1113
 using DESCRIBE INPUT 610
 varying-list SELECT statements 613, 625
 DYNAMICRULES bind option 502

E

ECB (event control block)
 address in CALL DSNALI parameter list 869
 CONNECT connection function of CAF 871, 875
 CONNECT, RRSAF 904
 program example 886, 888
 programming with CAF (call attachment facility) 886
 EDIT panel, SPUFI
 empty 66
 SQL statements 67
 embedded semicolon
 embedded 1025
 employee photo and resume sample table 999
 employee sample table 996
 employee-to-project-activity sample table 1002
 ENCRYPT_TDES function 266
 END-EXEC delimiter 78
 end-of-data condition 105, 109
 error
 arithmetic expression 93
 division by zero 93
 handling 93
 messages generated by precompiler 567
 overflow 93
 return codes 91
 run 566
 ESTAE routine in CAF (call attachment facility) 884
 exception condition handling 93
 EXCLUSIVE
 lock mode
 effect on resources 405
 LOB 427
 page 405
 row 405
 table, partition, and table space 405
 EXEC SQL delimiter 78
 EXECUTE IMMEDIATE statement 604
 EXECUTE statement
 dynamic execution 607

- EXECUTE statement (*continued*)
 - parameter types 624
 - USING DESCRIPTOR clause 625
- EXISTS predicate, subquery 53
- EXIT handler (SQL procedure) 663
- exit routine
 - abend recovery with CAF 884
 - attention processing with CAF 883
 - DSNACICX 1135
- EXPLAIN
 - automatic rebind 391
 - report of outer join 817
 - statement
 - description 789
 - index scans 802
 - interpreting output 800
 - investigating SQL processing 789
- EXPLAIN PROCESSING field of panel DSNTIPO
 - overhead 799
- EXPLAIN STATEMENT CACHE ALL 599

F

- FETCH FIRST *n* ROWS ONLY clause
 - effect on OPTIMIZE clause 775
- FETCH FIRST *n* ROWS ONLY clause
 - effect on distributed performance 464
- FETCH statement
 - description, multiple rows 109
 - description, single row 106
 - fetch orientation 117
 - host variables 612
 - multiple-row
 - assembler 145
 - description 109
 - FOR *n* ROWS clause 113
 - number of rows in rowset 113
 - using with descriptor 109, 111
 - using with host variable arrays 109
 - row and rowset positioning 117
 - scrolling through data 983
 - USING DESCRIPTOR clause 623
 - using row-positioned cursor 106
- filter factor
 - predicate 746
- fixed-length character string
 - assembler 149
 - COBOL 200
- FLAG precompiler option 486
- FLOAT precompiler option 486
- FOLD
 - value for C and CPP 487
 - value of precompiler option HOST 487
- FOR FETCH ONLY clause 458
- FOR READ ONLY clause 458
- FOR UPDATE clause 104
- FOREIGN KEY clause
 - description 264
 - usage 265
- format
 - SELECT statement results 70
 - SQL in input data set 66
- Fortran application program
 - @PROCESS statement 222
 - assignment rules 226
 - byte data type 223
 - character host variable 223, 224
- Fortran application program (*continued*)
 - coding SQL statements 220
 - constant syntax 227
 - data type compatibility 227
 - data types 225
 - declaring tables 222
 - declaring views 222
 - defining the SQLDA 221
 - host variable, declaring 223
 - INCLUDE statement 222
 - including SQLCA 220
 - indicator variable 228
 - LOB variable 224
 - naming convention 222
 - numeric host variable 223
 - parallel option 223
 - precompiler option defaults 492
 - result set locator 224
 - ROWID variable 224
 - SQLCODE 220
 - SQLSTATE 220
 - statement labels 222
 - variable declaration 226
 - WHENEVER statement 222
- FROM clause
 - joining tables 39
 - SELECT statement 5
- FRR (functional recovery routine) 884
- FULL OUTER JOIN clause 41
- function
 - column
 - when evaluated 808
- function resolution 356
- functional recovery routine (FRR) 884

G

- GET DIAGNOSTICS
 - output host variable processing 89
- GET DIAGNOSTICS statement
 - condition items 94
 - connection items 94
 - data types for items 94, 95
 - description 94
 - multiple-row INSERT 94
 - RETURN_STATUS item 665
 - ROW_COUNT item 109
 - SQL procedure 660
 - statement items 94
 - using in handler 664
- global transaction
 - RRSAF support 911, 916, 920
- glossary 1187
- GO TO clause of WHENEVER statement 93
- GOTO statement (SQL procedure) 660
- governor (resource limit facility) 601
- GRANT statement 559
- graphic host variable
 - assembler 150
 - C 164
 - COBOL 195
 - PL/I 235
- graphic host variable array
 - C 171
 - COBOL 203
 - PL/I 238
- GRAPHIC precompiler option 486

- GROUP BY clause
 - effect on OPTIMIZE clause 777
 - use with aggregate functions 11

H

- handler, using in SQL procedure 663
- HAVING clause
 - selecting groups subject to conditions 11
 - subquery 51
- HOST
 - FOLD value for C and CPP 487
 - precompiler option 487
- host language
 - declarations in DB2I (DB2 Interactive) 132
 - dynamic SQL 603
- host structure
 - C 173
 - COBOL 205
 - description 80, 90
 - PL/I 240
 - retrieving row of data 90
 - using SELECT INTO 90
- host variable
 - assembler 148
 - C 161, 162
 - changing CCSID 85
 - character
 - assembler 149
 - C 162
 - COBOL 194
 - Fortran 224
 - PL/I 235
 - COBOL 191, 192
 - description 79
 - example query 759
 - FETCH statement 612
 - floating-point
 - assembler 149
 - C/C++ 179
 - COBOL 192
 - PL/I 244
 - Fortran 223
 - graphic
 - assembler 150
 - C 164
 - COBOL 195
 - PL/I 235
 - impact on access path selection 759
 - in equal predicate 763
 - inserting into tables 83
- LOB
 - assembler 301
 - C 301
 - COBOL 302
 - Fortran 303
 - PL/I 304
- naming a structure
 - C 173
 - COBOL 205
 - PL/I program 240
- numeric
 - assembler 149
 - C 162
 - COBOL 192
 - Fortran 223
 - PL/I 234

- host variable (*continued*)
 - PL/I 233, 234
 - PREPARE statement 611
 - REXX 254
 - selecting single row 81
 - static SQL flexibility 594
 - tuning queries 759
 - updating values in tables 82
 - using 80
 - using INSERT with VALUES clause 83
 - using SELECT INTO 81
 - using SELECT INTO with aggregate function 82
 - using SELECT INTO with expressions 82
- host variable array
 - C 161, 168
 - character
 - C 169
 - COBOL 200
 - PL/I 238
 - COBOL 191, 199
 - description 80, 86
 - graphic
 - C 171
 - COBOL 203
 - PL/I 238
 - indicator variable array 87
 - inserting multiple rows 87
 - numeric
 - C 168
 - COBOL 199
 - PL/I 237
 - PL/I 233, 237
 - retrieving multiple rows 87
- hybrid join
 - description 821

I

- I/O processing
 - parallel
 - queries 849
- IDENTIFY, RRSAP
 - program example 936
 - syntax 904
 - usage 904
- identity column
 - defining 31, 271
 - IDENTITY_VAL_LOCAL function 272
 - inserting in table 983
 - inserting values into 30
 - trigger 281
 - using as parent key 272
- IF statement (SQL procedure) 660
- IKJEFT01 terminal monitor program in TSO 510
- IMS
 - checkpoint calls 434
 - CHKP call 433
 - commit point 434
 - environment planning 511
 - language interface module (DFSLI000) 495
 - link-editing 495
 - recovery 433, 435
 - restrictions on commit 435
 - ROLB call 433, 438
 - ROLL call 433, 438
 - SYNC call 433
 - unit of work 433, 434

- IMS (*continued*)
 - XRST call 435
- IMS transactions stored procedure
 - multiple connections 1143
 - option descriptions 1140
 - syntax diagram 1139
- IN predicate, subquery 52
- INCLUDE statement, DCLGEN output 136
- index
 - access methods
 - access path selection 810
 - by nonmatching index 812
 - IN-list index scan 812
 - matching index columns 802
 - matching index description 811
 - multiple 812
 - one-fetch index scan 814
 - locking 407
 - types
 - foreign key 264
 - primary 263
 - unique 263
 - unique on primary key 261
- indicator structure 90
- indicator variable
 - array declaration in DCLGEN 134
 - assembler application program 156
 - C 182
 - COBOL 216
 - description 83
 - Fortran 228
 - incorrect test for null column value 84
 - inserting null value 84
 - null value 84
 - PL/I 246
 - REXX 257
 - specifying 84
 - testing 83
- indicator variable array
 - C 182
 - COBOL 216
 - description 87
 - inserting null values 88
 - PL/I 246
 - specifying 88
 - testing for null value 87
- infinite loop 16
- informational referential constraint
 - automatic query rewrite 267
 - description 267
- INLISTP 787
- INNER JOIN clause 40
- input data set DDITV02 576
- INSERT processing, effect of MEMBER CLUSTER option of
 - CREATE TABLESPACE 398
- INSERT statement
 - description 27
 - multiple rows 29
 - single row 28
 - subquery 51
 - VALUES clause 27
 - with identity column 30
 - with ROWID column 30
- inserting
 - values from host variable arrays 87
 - values from host variables 83
- INTENT EXCLUSIVE lock mode 405, 427

- INTENT SHARE lock mode 405, 427
- Interactive System Productivity Facility (ISPF) 59
- internal resource lock manager (IRLM) 579
- invalid SQL terminator characters 1024
- IS DISTINCT FROM predicate 85
- ISOLATION
 - option of BIND PLAN subcommand
 - effects on locks 412
- isolation level
 - control by SQL statement
 - example 421
 - recommendations 400
 - REXX 258
- ISPF (Interactive System Productivity Facility)
 - browse 61, 69
 - DB2 uses dialog management 59
 - DB2I Primary Option Menu 519
 - precompiling under 518
 - Program Preparation panel 522
 - programming 857, 860
 - scroll command 71
- ISPLINK SELECT services 859
- ITERATE statement (SQL procedure) 660

J

- JCL (job control language)
 - batch backout example 581
 - DDNAME list format 514
 - page number format 515
 - precompilation procedures 512
 - precompiler option list format 514
 - preparing a CICS program 516
 - preparing a object-oriented program 517
 - starting a TSO batch application 510
- join operation
 - Cartesian 819
 - description 815
 - FULL OUTER JOIN 41
 - hybrid
 - description 821
 - INNER JOIN 40
 - join sequence 823
 - joining a table to itself 41
 - joining tables 39
 - LEFT OUTER JOIN 42
 - merge scan 820
 - more than one join 45
 - more than one join type 45
 - nested loop 818
 - operand
 - nested table expression 46
 - user-defined table function 46
 - RIGHT OUTER JOIN 43
 - SQL rules 44
 - star join 823
 - star schema 823
- join sequence
 - definition 738

K

- KEEPDYNAMIC option
 - BIND PACKAGE subcommand 600
 - BIND PLAN subcommand 600

- key
 - composite 262
 - foreign 264
 - parent 261
 - primary
 - choosing 261
 - defining 263
 - recommendations for defining 263
 - using timestamp 261
 - unique 983
- keywords, reserved 1109

L

- label, column 621
- language interface modules
 - DSNALI 652
 - DSNCLI 495
 - DSNRLI 652
 - program preparation 381
- large object (LOB)
 - character conversion 309
 - declaring host variables 300
 - for precompiler 300
 - declaring LOB locators 300
 - defining and moving data into DB2 297
 - description 297
 - expression 306
 - indicator variable 308
 - locator 305
 - materialization 305
 - sample applications 300
- LEAVE statement (SQL procedure) 660
- LEFT OUTER JOIN clause 42
- level of a lock 402
- LEVEL precompiler option 487
- limited partition scan 806
- LINECOUNT precompiler option 487
- link-editing 494
 - AMODE option 554
 - RMODE option 554
- list prefetch
 - description 831
 - thresholds 832
- load module structure of CAF (call attachment facility) 864
- load module structure of RRSAP 898
- LOAD MVS macro used by CAF 863
- LOAD MVS macro used by RRSAP 895
- LOB
 - lock
 - concurrency with UR readers 417
 - description 425
- LOB (large object)
 - lock duration 427
 - LOCK TABLE statement 428
 - locking 425
 - modes of LOB locks 427
 - modes of table space locks 427
- LOB column, definition 297
- LOB variable
 - assembler 151
 - C 167
 - COBOL 198
 - Fortran 224
 - PL/I 236
- LOB variable array
 - C 172

- LOB variable array (*continued*)
 - COBOL 204
 - PL/I 239
- lock
 - avoidance 418
 - benefits 394
 - class
 - transaction 393
 - compatibility 406
 - description 393
 - duration
 - controlling 408
 - description 404
 - LOBs 427
 - effect of cursor WITH HOLD 420
 - effects
 - deadlock 395
 - suspension 394
 - timeout 394
 - escalation
 - when retrieving large numbers of rows 987
 - hierarchy
 - description 402
 - LOB locks 425
 - mode 404
 - object
 - description 407
 - indexes 407
 - options affecting
 - access path 424
 - bind 408
 - cursor stability 412
 - program 408
 - read stability 415
 - repeatable read 416
 - uncommitted read 414
 - page locks
 - CS, RS, and RR compared 416
 - description 402
 - recommendations for concurrency 397
 - size
 - page 402
 - partition 402
 - table 402
 - table space 402
 - unit of work 431, 432
- LOCK TABLE statement
 - effect on auxiliary tables 428
 - effect on locks 422
- LOCKPART clause of CREATE and ALTER TABLESPACE
 - effect on locking 403
- LOCKSIZE clause
 - recommendations 398
- LOOP statement (SQL procedure) 660

M

- mapping macro
 - assembler applications 158
 - DSNXDBRM 477
 - DSNXNBRM 477
- MARGINS precompiler option 487
- mass delete
 - contends with UR process 417
- materialization
 - LOBs 305
 - outer join 817

- materialization (*continued*)
 - views and nested table expressions 837
- MEMBER CLUSTER option of CREATE TABLESPACE 398
- merge processing
 - views or nested table expressions 836
- message
 - analyzing 567
 - CAF errors 881
 - obtaining text
 - assembler 157
 - C 184
 - COBOL 217
 - Fortran 229
 - PL/I 247
 - RRSAF errors 902
- mixed data
 - converting 467
 - description 4
 - transmitting to remote location 467
- MLS (multilevel security)
 - referential constraints 266
 - triggers 289
- mode of a lock 404
- modified source statements 477
- MQSeries
 - DB2 functions
 - commit environment 948
 - connecting applications 962
 - MQPUBLISH 945
 - MQPUBLISHXML 947
 - MQREAD 945
 - MQREADALL 946
 - MQREADALLCLOB 946
 - MQREADALLXML 947
 - MQREADCLOB 945
 - MQREADXML 947
 - MQRECEIVE 945
 - MQRECEIVEALL 946
 - MQRECEIVEALLCLOB 946
 - MQRECEIVEALLXML 947
 - MQRECEIVECLOB 945
 - MQRECEIVEXML 947
 - MQSEND 945
 - MQSENDXML 947
 - MQSENDXMLFILE 947
 - MQSENDXMLFILECLOB 947
 - MQSUBSCRIBE 945
 - MQUNSUBSCRIBE 945
 - programming considerations 944
 - retrieving messages 961
 - sending messages 960
 - DB2 scalar functions 944
 - DB2 stored procedures
 - DXXMQINSERT 947, 948
 - DXXMQINSERTALL 947, 948
 - DXXMQINSERTALLCLOB 947
 - DXXMQINSERTCLOB 947
 - DXXMQRETRIEVE 948
 - DXXMQRETRIEVECLOB 948
 - DXXMQSHRED 947
 - DXXMQSHREDALL 947
 - DXXMQSHREDALLCLOB 947
 - DXXMQSHREDCLOB 947
 - DB2 table functions 946
 - DB2 XML-specific functions 947
 - description 941

- multilevel security (MLS) check
 - referential constraints 266
 - triggers 289
- multiple-mode IMS programs 436
- multiple-row FETCH statement
 - checking DB2_LAST_ROW 96
 - specifying indicator arrays 88
 - SQLCODE +100 92
 - testing for null 88
- multiple-row INSERT statement
 - dynamic execution 608
 - NOT ATOMIC CONTINUE ON SQLEXCEPTION 94
 - using GET DIAGNOSTICS 94
- MVS
 - 31-bit addressing 554

N

- naming convention
 - assembler 145
 - C 161
 - COBOL 189
 - Fortran 222
 - PL/I 232
 - REXX 253
 - tables you create 20
- NATIONAL data type 213
- nested table expression
 - correlated reference 46
 - correlation name 46
 - join operation 46
 - processing 836
- NEWFUN
 - enabling V8 new object 472
 - precompiler option 488
- NODYNAM option of COBOL 190
- NOFOR precompiler option 488
- NOGRAPHIC precompiler option 488
- noncorrelated subqueries 767
- nonsegmented table space
 - scan 810
- nontabular data storage 989
- NOOPTIONS precompiler option 488
- NOPADNTSTR precompiler option 488
- NOSOURCE precompiler option 488
- NOT FOUND clause of WHENEVER statement 93
- notices, legal 1183
- NOXREF precompiler option 488
- NUL character in C 161
- NUL-terminated string in C 180
- NULL
 - pointer in C 161
- null value
 - column value of UPDATE statement 37
 - host structure 90
 - indicator variable 84
 - indicator variable array 87
 - inserting into columns 84
 - IS DISTINCT FROM predicate 84
 - IS NULL predicate 84
- Null, in REXX 253
- numeric
 - data
 - width of column in results 70
- numeric data
 - description 4
 - width of column in results 64

- numeric host variable
 - assembler 149
 - C 162
 - COBOL 192
 - Fortran 223
 - PL/I 234
- numeric host variable array
 - C 168
 - COBOL 199
 - PL/I 237

O

- object of a lock 407
- object-oriented program, preparation 517
- ON clause, joining tables 39
- ONEPASS precompiler option 488
- OPEN
 - statement
 - opening a cursor 105
 - opening a rowset cursor 108
 - performance 835
 - prepared SELECT 612
 - USING DESCRIPTOR clause 625
 - without parameter markers 623
- OPEN (connection function of CAF)
 - description 866
 - language examples 876
 - program example 886
 - syntax 875
 - syntax usage 875
- optimistic concurrency control 412
- OPTIMIZE FOR *n* ROWS clause 776
 - interaction with FETCH FIRST clause 775
- OPTIMIZE FOR *n* ROWS clause
 - effect on distributed performance 461
- OPTIONS precompiler option 489
- ORDER BY clause
 - derived columns 10
 - effect on OPTIMIZE clause 777
 - SELECT statement 10
 - with AS clause 10
- organization application
 - examples 1013
- originating task 850
- outer join
 - EXPLAIN report 817
 - FULL OUTER JOIN 41
 - LEFT OUTER JOIN 42
 - materialization 817
 - RIGHT OUTER JOIN 43
- output host variable processing 89
 - errors 89

P

- package
 - advantages 385
 - binding
 - DBRM to a package 495
 - EXPLAIN option for remote 799
 - PLAN_TABLE 791
 - remote 496
 - to plans 498
 - deciding how to use 384
 - identifying at run time 498

- package (*continued*)
 - invalidated 389
 - dropping objects 387
 - listing 498
 - location 499
 - rebinding examples 388
 - rebinding with pattern-matching characters 387
 - selecting 498, 499
 - trigger 389
 - version, identifying 502
- PADNTSTR precompiler option 489
- page
 - locks
 - description 402
- PAGE_RANGE column of PLAN_TABLE 806
- panel
 - Current SPUFI Defaults 62, 65
 - DB2I Primary Option Menu 59
 - DCLGEN 132, 139
 - DSNEDP01 132, 139
 - DSNEPRI 59
 - DSNESP01 59
 - DSNESP02 62
 - DSNESP07 65
 - EDIT (for SPUFI input data set) 66
 - SPUFI 59
- parallel processing
 - description 847
 - enabling 850
 - related PLAN_TABLE columns 807
 - tuning 854
- parameter marker
 - casting in function invocation 363
 - dynamic SQL 606
 - more than one 607
 - values provided by OPEN 612
 - with arbitrary statements 624, 625
- parent key 261
- PARMS option 510
- partition scan, limited 806
- partitioned table space
 - locking 403
- performance
 - affected by
 - application structure 859
 - DEFER(PREPARE) 456
 - lock size 404
 - NODEFER(PREPARE) 456
 - remote queries 454, 456, 464
 - REOPT(ALWAYS) 457
 - REOPT(NONE) 457
 - REOPT(ONCE) 457
 - monitoring
 - with EXPLAIN 789
- performance considerations
 - DPSI 772
 - scrollable cursor 771
- PERIOD precompiler option 489
- phone application, description 1013
- PL/I application program
 - character host variable 235
 - character host variable array 238
 - coding considerations 233
 - coding SQL statements 230
 - data type compatibility 245
 - data types 241
 - DB2 coprocessor 481

- PL/I application program (*continued*)
 - DBCS constants 232
 - DCLGEN support 136
 - declaring tables 231
 - declaring views 231
 - graphic host variable 235
 - graphic host variable array 238
 - host variable 233
 - host variable array 233
 - INCLUDE statement 232
 - including SQLCA 230
 - including SQLDA 231
 - indicator variable array 246
 - indicator variables 246
 - LOB variable 236
 - LOB variable array 239
 - naming convention 232
 - numeric host variable 234
 - numeric host variable array 237
 - result set locator 235
 - ROWID variable 236
 - ROWID variable array 239
 - SQLCODE 230
 - SQLSTATE 230
 - statement labels 232
 - table locator 236
 - variable declaration 244
 - WHENEVER statement 232
- PLAN_TABLE table
 - column descriptions 791
 - report of outer join 817
- planning
 - accessing distributed data 441
 - binding 384
 - precompiling 383
 - recovery 431
- precompiler
 - binding on another system 478
 - description 473
 - diagnostics 477
 - functions 474
 - input 476
 - maximum size of input 476
 - modified source statements 477
 - option descriptions 483
 - options
 - CONNECT 445
 - defaults 491
 - DRDA access 445
 - SQL 445
 - output 477
 - planning for 383
 - precompiling programs 473
 - starting
 - dynamically 514
 - JCL for procedures 512
 - submitting jobs
 - DB2I panels 528
 - ISPF panels 522
 - submitting jobs with ISPF panels 520
 - using 474
- predicate
 - description 735
 - evaluation rules 739
 - filter factor 746
 - general rules 8
 - generation 755

- predicate (*continued*)
 - impact on access paths 735
 - indexable 737
 - join 736
 - local 736
 - modification 755
 - properties 735
 - stage 1 (sargable) 737
 - stage 2
 - evaluated 737
 - influencing creation 782
 - subquery 736
- predictive governing
 - in a distributed environment 602
 - with DEFER(PREPARE) 602
 - writing an application for 602
- PRELINK utility 525
- PREPARE statement
 - dynamic execution 606
 - host variable 611
 - INTO clause 615
- prepared SQL statement
 - caching 600
 - statements allowed 1113
- PRIMARY KEY clause
 - ALTER TABLE statement 263
 - CREATE TABLE statement 262
- PRIMARY_ACCESSTYPE column of PLAN_TABLE 803
- problem determination, guidelines 566
- program preparation 471
- program problems checklist
 - documenting error situations 560
 - error messages 561
- project activity sample table 1001
- project application, description 1013
- project sample table 1000

Q

- query parallelism 847
- QUOTE precompiler option 489
- QUOTESQL precompiler option 489

R

- reason code
 - CAF
 - translation 885, 888
 - X"00C10824" 878, 879
 - X"00F30050" 884
 - X"00F30083" 884
 - X'00C90088' 396
 - X'00C9008E' 395
 - X"00D44057" 573
- REBIND PACKAGE subcommand of DSN
 - generating list of 1103
 - options
 - ISOLATION 412
 - RELEASE 408
 - rebinding with wildcard characters 387
 - remote 496
- REBIND PLAN subcommand of DSN
 - generating list of 1103
 - options
 - ACQUIRE 408
 - ISOLATION 412

- REBIND PLAN subcommand of DSN *(continued)*
 - options *(continued)*
 - NOPKLIST 388
 - PKLIST 388
 - RELEASE 408
 - remote 496
- REBIND TRIGGER PACKAGE subcommand of DSN 389
- rebinding
 - automatically
 - conditions for 389
 - EXPLAIN processing 799
 - changes that require 386
 - list of plans and packages 389
 - lists of plans or packages 1103
 - options for 384
 - packages with pattern-matching characters 387
 - planning for 391
 - plans 388
 - plans or packages in use 384
- Recoverable Resource Manager Services attachment facility (RRSAF)
 - See* RRSAF
- recovery
 - identifying application requirements 437
 - IMS application program 433
 - IMS batch 439
 - planning for 431
- recursive SQL
 - controlling depth 1100
 - description 15
 - examples 1097
 - infinite loops 16
 - rules 15
 - single level explosion 1097
 - summarized explosion 1099
- referential constraint
 - defining 261
 - description 261
 - determining violations 989
 - informational 267
 - name 264
 - on tables with data encryption 266
 - on tables with multilevel security 266
- referential integrity
 - effect on subqueries 56
 - programming considerations 989
- register conventions
 - CAF (call attachment facility) 869
 - RRSAF 903
- RELEASE
 - option of BIND PLAN subcommand
 - combining with other options 408
- release information block (RIB) 869
- RELEASE LOCKS field of panel DSN TIP4
 - effect on page and row locks 420
- RELEASE SAVEPOINT statement 440
- RELEASE statement, with DRDA access 451
- reoptimizing access path 760
- REPEAT statement (SQL procedure) 660
- REPLACE statement (COBOL) 190
- reserved keywords 1109
- resetting control blocks
 - CAF 878
 - RRSAF 930
- RESIGNAL statement
 - raising a condition 665
 - setting SQLSTATE value 667
- RESIGNAL statement (SQL procedure) 661
- resource limit facility (governor)
 - description 601
 - writing an application for predictive governing 602
- resource unavailable condition
 - CAF 880
 - RRSAF 931
- restart, DL/I batch programs using JCL 581
- result column
 - join operation 39
 - naming with AS clause 7
- result set locator
 - assembler 150
 - C 166
 - COBOL 197
 - example 710
 - Fortran 224
 - how to use 710
 - PL/I 235
- result table
 - description 3
 - example 3
 - of SELECT statement 3
 - read-only 105
- retrieving
 - data in ASCII from DB2 UDB for z/OS 619
 - data in Unicode from DB2 UDB for z/OS 619
 - data using SELECT * 987
 - data, changing the CCSID 619
 - large volumes of data 987
 - multiple rows into host variable arrays 87
- return code
 - DSN command 509
 - SQL 878
- RETURN statement
 - returning SQL procedure status 665
- RETURN statement (SQL procedure) 661
- REXX procedure
 - application programming interface
 - CONNECT 250
 - DISCONNECT 251
 - EXEC SQL 250
 - coding SQL statements 249
 - data type conversion 254
 - DSNREXX 251
 - error handling 253
 - indicator variable 257
 - input data type 254, 255
 - isolation level 258
 - naming convention 253
 - naming cursors 253
 - naming prepared statements 253
 - running 512
 - SQLCA 249
 - SQLDA 250
 - statement label 253
- RIB (release information block)
 - address in CALL DSNALI parameter list 869
 - CONNECT connection function of CAF 871
 - CONNECT, RRSAF 904
 - program example 886
- RID (record identifier) pool
 - use in list prefetch 831
- RIGHT OUTER JOIN clause 43
- RMODE link-edit option 554
- ROLB call, IMS
 - advantages over ROLL 439

- ROLB call, IMS (*continued*)
 - DL/I batch programs 438
 - ends unit of work 433
- ROLL call, IMS
 - DL/I batch programs 438
 - ends unit of work 433
- ROLLBACK option
 - CICS SYNCPOINT command 433
- ROLLBACK statement
 - description 61
 - error in IMS 573
 - in a stored procedure 646
 - TO SAVEPOINT clause 440
 - unit of work in TSO 432
 - with RRSAP 895
- row
 - selecting with WHERE clause 8
 - updating 36
 - updating current 107
 - updating large volumes 986
- row-level security 266
- ROWID
 - coding example 805
 - data type 4
 - index-only access 802
 - inserting in table 983
- ROWID column
 - defining 30, 269
 - defining LOBs 297
 - inserting values into 30
 - using for direct row access 270
- ROWID variable
 - assembler 151
 - C 168
 - COBOL 198
 - Fortran 224
 - PL/I 236
- ROWID variable array
 - C 173
 - COBOL 205
 - PL/I 239
- rowset
 - deleting current 112
 - updating current 111
- rowset cursor
 - closing 113
 - DB2 for z/OS down-level requester 467
 - declaring 108
 - end-of-data condition 109
 - example 126
 - multiple-row FETCH 109
 - opening 108
 - using 108
- rowset parameter, DB2 for z/OS support for 465
- RR (repeatable read)
 - how locks are held (figure) 416
 - page and row locking 416
- RRS global transaction
 - RRSAP support 911, 916, 920
- RRSAP
 - application program
 - examples 935
 - preparation 894
 - connecting to DB2 936
 - description 893
 - function descriptions 903
 - load module structure 898

- RRSAP (*continued*)
 - programming language 894
 - register conventions 903
 - restrictions 893
 - return codes
 - AUTH SIGNON 914
 - CONNECT 904
 - SIGNON 909
 - TERMINATE IDENTIFY 930
 - TERMINATE THREAD 928
 - TRANSLATE 931
 - run environment 895
- RRSAP (Recoverable Resource Manager Services attachment facility)
 - transactions
 - using global transactions 401
- RS (read stability)
 - page and row locking (figure) 415
- RUN subcommand of DSN
 - return code processing 509
 - running a program in TSO foreground 508
- run-time libraries, DB2I
 - background processing 527
 - EDITJCL processing 527
- running application program
 - CICS 512
 - errors 566
 - IMS 511

S

- sample application
 - call attachment facility 862
 - databases, for 1010
 - DB2 private protocol access 1069
 - DRDA access 1061
 - dynamic SQL 1043
 - environments 1015
 - languages 1015
 - LOB 1014
 - organization 1013
 - phone 1013
 - programs 1015
 - project 1013
 - RRSAP 894
 - static SQL 1043
 - stored procedure 1013
 - structure of 1009
 - use 1015
 - user-defined function 1014
- sample program
 - DSN8BC3 218
 - DSN8BD3 185
 - DSN8BE3 185
 - DSN8BF3 230
 - DSN8BP3 248
- sample table
 - DSN8810.ACT (activity) 993
 - DSN8810.DEMO_UNICODE (Unicode sample) 1003
 - DSN8810.DEPT (department) 994
 - DSN8810.EMP (employee) 996
 - DSN8810.EMP_PHOTO_RESUME (employee photo and resume) 999
 - DSN8810.EMPPROJECT (employee-to-project activity) 1002
 - DSN8810.PROJ (project) 1000
 - PROJECT (project activity) 1001

- sample table (*continued*)
 - views on 1004
- savepoint
 - description 439
 - distributed environment 442
 - RELEASE SAVEPOINT statement 440
 - restrictions on use 440
 - ROLLBACK TO SAVEPOINT 440
 - SAVEPOINT statement 440
 - setting multiple times 440
 - use with DRDA access 440
- SAVEPOINT statement 440
- scope of a lock 402
- scrollable cursor
 - comparison of types 118
 - DB2 UDB for z/OS down-level requester 467
 - distributed environment 442
 - dynamic
 - dynamic model 115
 - fetching current row 119
 - fetch orientation 117
 - optimistic concurrency control 412
 - performance considerations 771
 - retrieving rows 116
 - sensitive dynamic 115
 - sensitive static 114
 - sensitivity 119
 - static
 - creating delete hole 119
 - creating update hole 120
 - holes in result table 119
 - number of rows 117
 - removing holes 121
 - static model 115
 - updatable 114
- scrolling
 - backward through data 983
 - backward using identity columns 984
 - backward using ROWIDs 984
 - in any direction 985
 - ISPF (Interactive System Productivity Facility) 71
- search condition
 - comparison operators 9
 - NOT keyword 9
 - SELECT statement 49
 - WHERE clause 9
- segmented table space
 - locking 403
 - scan 810
- SEGSIZE clause of CREATE TABLESPACE
 - recommendations 810
- SELECT FROM INSERT statement
 - BEFORE trigger values 32
 - default values 31
 - description 31
 - inserting into view 33
 - multiple rows
 - cursor sensitivity 34
 - effect of changes 34
 - effect of SAVEPOINT and ROLLBACK 35
 - effect of WITH HOLD 35
 - processing errors 35
 - result table of cursor 34
 - using cursor 33
 - using FETCH FIRST 33
 - using INPUT SEQUENCE 33
 - result table 32
- SELECT FROM INSERT statement (*continued*)
 - retrieving
 - BEFORE trigger values 31
 - default values 31
 - generated values 31
 - multiple rows 31
 - special registers 31
 - using SELECT INTO 33
- SELECT statement
 - changing result format 70
 - clauses
 - DISTINCT 7
 - FROM 5
 - GROUP BY 11
 - HAVING 11
 - ORDER BY 10
 - UNION 12
 - WHERE 8
 - derived column with AS clause 7
 - fixed-list 610, 613
 - named columns 6
 - parameter markers 624
 - search condition 49
 - selecting a set of rows 103
 - subqueries 49
 - unnamed columns 7
 - using with
 - * (to select all columns) 5
 - column-name list 6
 - DECLARE CURSOR statement 103, 108
 - varying-list 613, 625
- selecting
 - all columns 5
 - more than one row 81
 - named columns 6
 - rows 8
 - some columns 6
 - unnamed columns 7
- semicolon
 - default SPUFI statement terminator 62
 - embedded 1025
- sequence numbers
 - COBOL application program 189
 - Fortran 222
 - PL/I 232
- sequence object
 - creating 273
 - referencing 274
 - using across multiple tables 274
- sequences
 - improving concurrency 401
- sequential detection 832, 834
- sequential prefetch
 - bind time 831
 - description 830
- SET clause of UPDATE statement 36
- SET CURRENT DEGREE statement 850
- SET CURRENT PACKAGESET statement 499
- SET ENCRYPTION PASSWORD statement 266
- setting SQL terminator
 - DSNTIAD 1024
 - SPUFI 67
- SHARE
 - INTENT EXCLUSIVE lock mode 405, 427
 - lock mode
 - LOB 427
 - page 405

- SHARE (*continued*)
 - lock mode (*continued*)
 - row 405
 - table, partition, and table space 405
- SIGNAL statement
 - raising a condition 665
 - setting condition message text 666
- SIGNAL statement (SQL procedure) 661
- SIGNON, RRSF
 - program example 936
 - syntax 909
 - usage 909
- simple table space
 - locking 403
- single-mode IMS programs 436
- SOME quantified predicate 52
- sort
 - program
 - RIDs (record identifiers) 835
 - when performed 835
 - removing duplicates 835
 - shown in PLAN_TABLE 834
- sort key
 - ORDER BY clause 10
 - ordering 10
- SOURCE precompiler option 489
- special register
 - behavior in stored procedures 647
 - CURRENT PACKAGE PATH 500
 - CURRENT PACKAGESET 500
 - CURRENT RULES 505
 - user-defined functions 342
- SPUFI
 - browsing output 69
 - changed column widths 70
 - CONNECT LOCATION field 61
 - created column heading 71
 - DB2 governor 68
 - default values 62
 - entering comments 67
 - panels
 - allocates RESULT data set 60
 - filling in 60
 - format and display output 69
 - previous values displayed on panel 59
 - selecting on DB2I menu 59
 - processing SQL statements 59, 68
 - retrieving Unicode data 67
 - setting SQL terminator 67
 - specifying SQL statement terminator 62
 - SQLCODE returned 70
- SQL (Structured Query Language)
 - checking execution 91
 - coding
 - assembler 143
 - basics 77
 - C 158
 - C++ 158
 - COBOL 186
 - dynamic 627
 - Fortran 220
 - Fortran program 221
 - object extensions 295
 - PL/I 230
 - REXX 249
 - cursors 103

- SQL (Structured Query Language) (*continued*)
 - dynamic
 - coding 593
 - sample C program 1043
 - statements allowed 1113
 - host variable arrays 79
 - host variables 79
 - keywords, reserved 1109
 - return codes
 - checking 91
 - handling 98
 - statement terminator 1024
 - string delimiter 527
 - structures 79
 - syntax checking 465
 - varying-list 613, 625
- SQL communication area (SQLCA)
 - description 91
 - using DSNTIAR to format 98
- SQL precompiler option 490
- SQL procedure
 - conditions, handling 663
 - forcing SQL error 667
 - preparation using DSNTPSMP procedure 670
 - program preparation 669
 - referencing SQLCODE and SQLSTATE 664
 - SQL variable 661
 - statements allowed 1118
- SQL procedure statement
 - CALL statement 660
 - CASE statement 660
 - compound statement 660
 - CONTINUE handler 663
 - EXIT handler 663
 - GET DIAGNOSTICS statement 660
 - GOTO statement 660
 - handler 663
 - handling errors 663
 - IF statement 660
 - ITERATE statement 660
 - LEAVE statement 660
 - LOOP statement 660
 - REPEAT statement 660
 - RESIGNAL statement 661
 - RETURN statement 661
 - SIGNAL statement 661
 - SQL statement 660
 - WHILE statement 660
- SQL statement (SQL procedure) 660
- SQL statement nesting
 - restrictions 363
 - stored procedures 363
 - user-defined functions 363
- SQL statement terminator
 - modifying in DSNTPE2 and DSNTPE4 1026
 - modifying in DSNTIAD 1024
 - modifying in SPUFI 62
 - specifying in SPUFI 62
- SQL statements
 - ALLOCATE CURSOR 710
 - ALTER FUNCTION 314
 - ASSOCIATE LOCATORS 709
 - CLOSE 108, 113, 612
 - COBOL program sections 187
 - coding REXX 252
 - comments
 - assembler 145

SQL statements (*continued*)

comments (*continued*)

C 160
 COBOL 188
 Fortran 221
 PL/I 231
 REXX 252

CONNECT (Type 1) 453

CONNECT (Type 2) 453

CONNECT, with DRDA access 450

continuation

assembler 145
 C 160
 COBOL 188
 Fortran 221
 PL/I 231
 REXX 252

CREATE FUNCTION 314

DECLARE CURSOR

description 103, 108

example 611, 615

DECLARE TABLE 79, 131

DELETE

description 107

example 37

DESCRIBE 615

DESCRIBE CURSOR 710

DESCRIBE PROCEDURE 709

embedded 476

error return codes 98

EXECUTE 607

EXECUTE IMMEDIATE 604

EXPLAIN

monitor access paths 789

FETCH

description 106, 109

example 612

INSERT 27

labels

assembler 146
 C 161
 COBOL 189
 Fortran 222
 PL/I 232
 REXX 253

margins

assembler 145
 C 161
 COBOL 189
 Fortran 222
 PL/I 232
 REXX 253

OPEN

description 105, 108

example 612

PREPARE 606

RELEASE, with DRDA access 451

SELECT

description 8

joining a table to itself 41

joining tables 39

SELECT FROM INSERT 31

SET CURRENT DEGREE 850

set symbols 147

UPDATE

description 107, 111, 112

example 36

SQL statements (*continued*)

WHENEVER 93

SQL terminator, specifying in DSNTEP2 and DSNTEP4 1026

SQL terminator, specifying in DSNTIAD 1024

SQL variable 661

SQL-INIT-FLAG, resetting 191

SQLCA (SQL communication area)

assembler 143

C 158

checking SQLCODE 92

checking SQLERRD(3) 91

checking SQLSTATE 92

checking SQLWARN0 92

COBOL 186

description 91

DSNTIAC subroutine

assembler 158

C 186

COBOL 219

PL/I 249

DSNTIAR subroutine

assembler 157

C 184

COBOL 217

Fortran 229

PL/I 247

Fortran 220

PL/I 230

reason code for deadlock 396

reason code for timeout 395

REXX 249

sample C program 1043

SQLCODE

-510 419

-923 577

-925 438, 573

-926 438, 573

+004 878, 879

+100 93

+256 884

+802 94

referencing in SQL procedure 664

values 92

SQLDA (SQL descriptor area)

allocating storage 110, 616

assembler 144

assembler program 614

C 159, 614

COBOL 187

declaring 110

dynamic SELECT example 618

for LOBs and distinct types 621

Fortran 220

multiple-row FETCH statement 110

no occurrences of SQLVAR 615

OPEN statement 612

parameter in CAF TRANSLATE 880

parameter in RRSF TRANSLATE 931

parameter markers 624

PL/I 230, 614

requires storage addresses 619

REXX 250

setting output fields 110

varying-list SELECT statement 614

SQLERROR clause of WHENEVER statement 93

SQLFLAG precompiler option 490

SQLN field of SQLDA 615

- SQLRULES, option of BIND PLAN subcommand 505
- SQLSTATE
 - "01519" 94
 - "2D521" 438, 573
 - "57015" 577
 - referencing in SQL procedure 664
 - values 92
- SQLVAR field of SQLDA 617
- SQLWARNING clause of WHENEVER statement 93
- SSID (subsystem identifier), specifying 526
- SSN (subsystem name)
 - CALL DSNALI parameter list 869
 - parameter in CAF CONNECT function 871
 - parameter in CAF OPEN function 875
 - parameter in RRSAF CONNECT function 904
 - SQL calls to CAF (call attachment facility) 866
 - SQL calls to RRSAF (recoverable resources services attachment facility) 897
- star join 823
 - dedicated virtual memory pool 828
- star schema
 - defining indexes for 782
- state
 - of a lock 404
- statement cache table 599
- statement table
 - column descriptions 843
- static SQL
 - description 593
 - host variables 594
 - sample C program 1043
- STDDEV function
 - when evaluation occurs 808
- STDSQL precompiler option 490
- STOP DATABASE command
 - timeout 395
- storage
 - acquiring
 - retrieved row 617
 - SQLDA 616
 - addresses in SQLDA 619
- storage group, for sample application data 1010
- stored procedure
 - accessing transition tables 345, 714
 - binding 653
 - CALL statement 681
 - calling from a REXX procedure 714
 - defining parameter lists 687, 688, 689
 - defining to DB2 635
 - DSNACICS 1131
 - DSNAEXP 1143
 - DXXMQGEN 1165
 - DXXMQGENCLOB 1173
 - DXXMQINSERT 1146
 - DXXMQINSERTALL 1156
 - DXXMQINSERTALLCLOB 1163
 - DXXMQINSERTCLOB 1151
 - DXXMQRETRIEVE 1169
 - DXXMQRETRIEVECLOB 1176
 - DXXMQSHRED 1148
 - DXXMQSHREDALL 1158
 - DXXMQSHREDALLCLOB 1161
 - DXXMQSHREDCLOB 1153
 - example 630
 - IMS transactions 1139
 - invoking from a trigger 285
 - languages supported 641

- stored procedure (*continued*)
 - linkage conventions 684
 - returning non-relational data 651
 - returning result set 650
 - running as authorized program 652
 - statements allowed 1116
 - testing 725
 - usage 629
 - use of special registers 647
 - using COMMIT in 646
 - using host variables with 633
 - using ROLLBACK in 646
 - using temporary tables in 651
 - WLM_REFRESH 1129
 - writing 641
 - writing in REXX 654
- stormdrain effect 940
- string
 - data type 4
 - fixed-length
 - assembler 149
 - COBOL 194
 - PL/I 246
 - host variables in C 180
 - varying-length
 - assembler 149
 - COBOL 194
 - PL/I 246
- subquery
 - basic predicate 51
 - conceptual overview 49
 - correlated
 - DELETE statement 56
 - description 53
 - example 53
 - tuning 766
 - UPDATE statement 55
 - DELETE statement 56
 - description 49
 - EXISTS predicate 53
 - IN predicate 52
 - join transformation 768
 - noncorrelated 767
 - quantified predicate 51
 - referential constraints 56
 - restrictions with DELETE 56
 - tuning 766
 - tuning examples 770
 - UPDATE statement 55
 - use with UPDATE, DELETE, and INSERT 51
- subsystem
 - identifier (SSID), specifying 526
- subsystem name (SSN) 866, 897
- summarizing group values 11
- SYNC call, IMS 433, 434
- SYNC parameter of CAF (call attachment facility) 877, 886
- synchronization call abends 576
- SYNCPPOINT command of CICS 433
- syntax diagram
 - how to read xxii
- SYSLIB data sets 513
- Sysplex query parallelism
 - splitting large queries across DB2 members 847
- SYSPRINT precompiler output
 - options section 568
 - source statements section, example 569
 - summary section, example 570

SYSPRINT precompiler output (*continued*)
 symbol cross-reference section 570
 used to analyze errors 568
SYSTEM output to analyze errors 567

T

table

- altering
 - changing definitions 21
 - using CREATE and ALTER 988
- copying from remote locations 467
- declaring 79, 131
- deleting rows 37
- dependent, cycle restrictions 265
- displaying, list of 18
- DROP statement 25
- expression, nested
 - processing 836
- filling with test data 559
- incomplete definition of 263
- inserting multiple rows 29
- inserting single row 28
- loading, in referential structure 261
- locks 402
- populating 559
- referential structure 261
- retrieving 103
- selecting values as you insert rows 31
- temporary 21
- updating rows 36
- using three-part table names 448

table expressions, nested

- materialization 837

table locator

- assembler 150
- C 167
- COBOL 198
- PL/I 236

table space

- for sample application 1010
- locks
 - description 402
- scans
 - access path 809
 - determined by EXPLAIN 790

task control block (TCB)
 See TCB (task control block)

TCB (task control block)

- capabilities with CAF 862
- capabilities with RRSAP 894
- issuing CAF CLOSE 878
- issuing CAF OPEN 876

temporary table

- advantages of 22
- working with 21

TERM call in DL/I 433

terminal monitor program (TMP)
 See TMP (terminal monitor program)

TERMINATE IDENTIFY, RRSAP

- program example 936
- syntax 930
- usage 930

TERMINATE THREAD, RRSAP

- program example 936
- syntax 928
- usage 928

terminating, CAF CLOSE function 877

TEST command of TSO 561

test environment, designing 557

test tables 557

test views of existing tables 557

thread

- CLOSE function 866
- OPEN function 866

TIME precompiler option 490

timeout

- description 394
- indications in IMS 395
- X'00C9008E' reason code in SQLCA 395

TMP (terminal monitor program)

- DSN command processor 509
- running under TSO 510

transaction

- IMS
 - using global transactions 401

transaction lock

- description 393

transaction-oriented BMP, checkpoints in 436

transition table, trigger 282

transition variable, trigger 281

TRANSLATE (connection function of CAF)

- description 866
- language example 881
- program example 888
- syntax usage 880

TRANSLATE function of RRSAP

- syntax 931
- usage 931

translating requests into SQL 988

trigger

- activation order 287
- activation time 279
- cascading 286
- coding 279
- data integrity 290
- delete 280
- description 277
- FOR EACH ROW 280
- FOR EACH STATEMENT 280
- granularity 280
- insert 280
- interaction with constraints 288
- interaction with security label columns 289
- naming 279
- parts example 277
- parts of 279
- subject table 279
- transition table 282
- transition variable 281
- triggering event 279
- update 280
- using identity columns 281
- with row-level security 289

TSO

- CLISTs
 - calling application programs 511
 - running in foreground 511
- DSNALI language interface module 863
- TEST command 561

tuning

- DB2
 - queries containing host variables 759

two-phase commit, definition 452

TWOPASS precompiler option 490

U

Unicode

- data, retrieving from DB2 UDB for z/OS 619
- sample table 1003

UNION clause

- columns of result table 13
- combining SELECT statements 12
- effect on OPTIMIZE clause 777
- eliminating duplicates 13
- keeping duplicates with ALL 13
- removing duplicates with sort 835

UNIQUE clause 262

unit of recovery

- indoubt
 - recovering CICS 433
 - restarting IMS 435

unit of work

- CICS description 432
- completion
 - commit 432
 - open cursors 122
 - releasing locks 431
 - roll back 432
 - TSO 432

- description 431

- DL/I batch 438

- duration 431

- IMS

- batch 438
 - commit point 434
 - ending 433
 - starting point 433

- prevention of data access by other users 431

- TSO

- COMMIT statement 432
 - completion 432
 - ROLLBACK statement 432

updatable cursor 104

UPDATE

- lock mode

- page 405
 - row 405

- table, partition, and table space 405

UPDATE statement

- correlated subqueries 55

- description 36

- positioned

- FOR ROW n OF ROWSET 112

- restrictions 107

- WHERE CURRENT clause 107, 111

- SET clause 36

- subquery 51

updating

- during retrieval 986

- large volumes 986

- values from host variables 82

UR (uncommitted read)

- concurrent access restrictions 417

- effect on reading LOBs 426

- page and row locking 414

- recommendation 401

USE AND KEEP EXCLUSIVE LOCKS option of WITH

- clause 421

USE AND KEEP SHARE LOCKS option of WITH clause 421

USE AND KEEP UPDATE LOCKS option of WITH

- clause 421

USER special register

- value in INSERT statement 20

- value in UPDATE statement 37

user-defined function

- statements allowed 1116

user-defined function (UDF)

- abnormal termination 363

- accessing transition tables 345

- ALTER FUNCTION statement 314

- authorization ID 351

- call type 329

- casting arguments 362

- characteristics 314

- coding guidelines 318

- concurrent 352

- CREATE FUNCTION statement 314

- data type promotion 359

- DBINFO structure 331

- definer 312

- defining 314

- description 311

- diagnostic message 328

- DSN_FUNCTION_TABLE 361

- example

- external scalar 312, 316

- external table 318

- function resolution 359

- overloading operator 317

- sourced 317

- SQL 317

- function resolution 356

- host data types

- assembler 323

- C 323

- COBOL 323

- PL/I 323

- implementer 312

- implementing 318

- indicators

- input 327

- result 328

- invoker 312

- invoking 355

- invoking from a trigger 285

- invoking from predicate 365

- main program 319

- multiple programs 351

- naming 328

- nesting SQL statements 363

- parallelism considerations 320

- parameter conventions 321

- assembler 334

- C 334

- COBOL 338

- PL/I 341

- preparing 350

- reentrant 351

- restrictions 319

- samples 313

- scratchpad 328, 344

- scrollable cursor 366

- setting result values 327

- simplifying function resolution 360

- special registers 342

- specific name 328

- user-defined function (UDF) *(continued)*
 - steps in creating and using 312
 - subprogram 319
 - syntax for invocation 355
 - table locators
 - assembler 346
 - C 348
 - COBOL 348
 - PL/I 349
 - testing 352
 - types 311
- user-defined table function
 - improving query performance 779
- USING DESCRIPTOR clause
 - EXECUTE statement 625
 - FETCH statement 623
 - OPEN statement 625

V

- VALUES clause, INSERT statement 27
- variable
 - declaration
 - assembler 154
 - C 179
 - COBOL 212
 - Fortran 226
 - PL/I 244
 - declaring in SQL procedure 661
 - host
 - assembler 148
 - COBOL 192
 - Fortran 223
 - PL/I 234
- variable array
 - host
 - C 168
 - COBOL 199
 - PL/I 237
- VARIANCE function
 - when evaluation occurs 808
- varying-length character string
 - assembler 149
 - COBOL 201
- version of a package 502
- VERSION precompiler option 491, 502
- view
 - contents 26
 - declaring 79
 - description 25
 - dropping 27
 - EXPLAIN 839, 840
 - identity columns 26
 - join of two or more tables 26
 - processing
 - view materialization description 837
 - view materialization in PLAN_TABLE 806
 - view merge 836
 - referencing special registers 26
 - retrieving 103
 - summary data 26
 - union of two or more tables 26
 - using
 - deleting rows 37
 - inserting rows 27
 - updating rows 36
- Visual Explain 775, 789

- volatile table 778

W

- WHENEVER statement
 - assembler 146
 - C 161
 - COBOL 189
 - CONTINUE clause 93
 - Fortran 222
 - GO TO clause 93
 - NOT FOUND clause 93, 106
 - PL/I 232
 - specifying 93
 - SQL error codes 93
 - SQLERROR clause 93
 - SQLWARNING clause 93
- WHERE clause
 - SELECT statement
 - description 8
 - joining a table to itself 41
 - joining tables 39
 - subquery 51
- WHILE statement (SQL procedure) 660
- WITH clause
 - common table expressions 13
 - specifies isolation level 421
- WITH HOLD clause
 - and CICS 123
 - and IMS 123
 - DECLARE CURSOR statement 122
 - restrictions 123
- WITH HOLD cursor
 - effect on dynamic SQL 607
 - effect on locks and claims 420
- WLM_REFRESH stored procedure
 - description 1129
 - option descriptions 1130
 - sample JCL 1131
 - syntax diagram 1130
- write-down privilege 289

X

- XREF precompiler option 491
- XRST call, IMS 435



Program Number: 5625-DB2

Printed in USA

SC18-7415-05

