

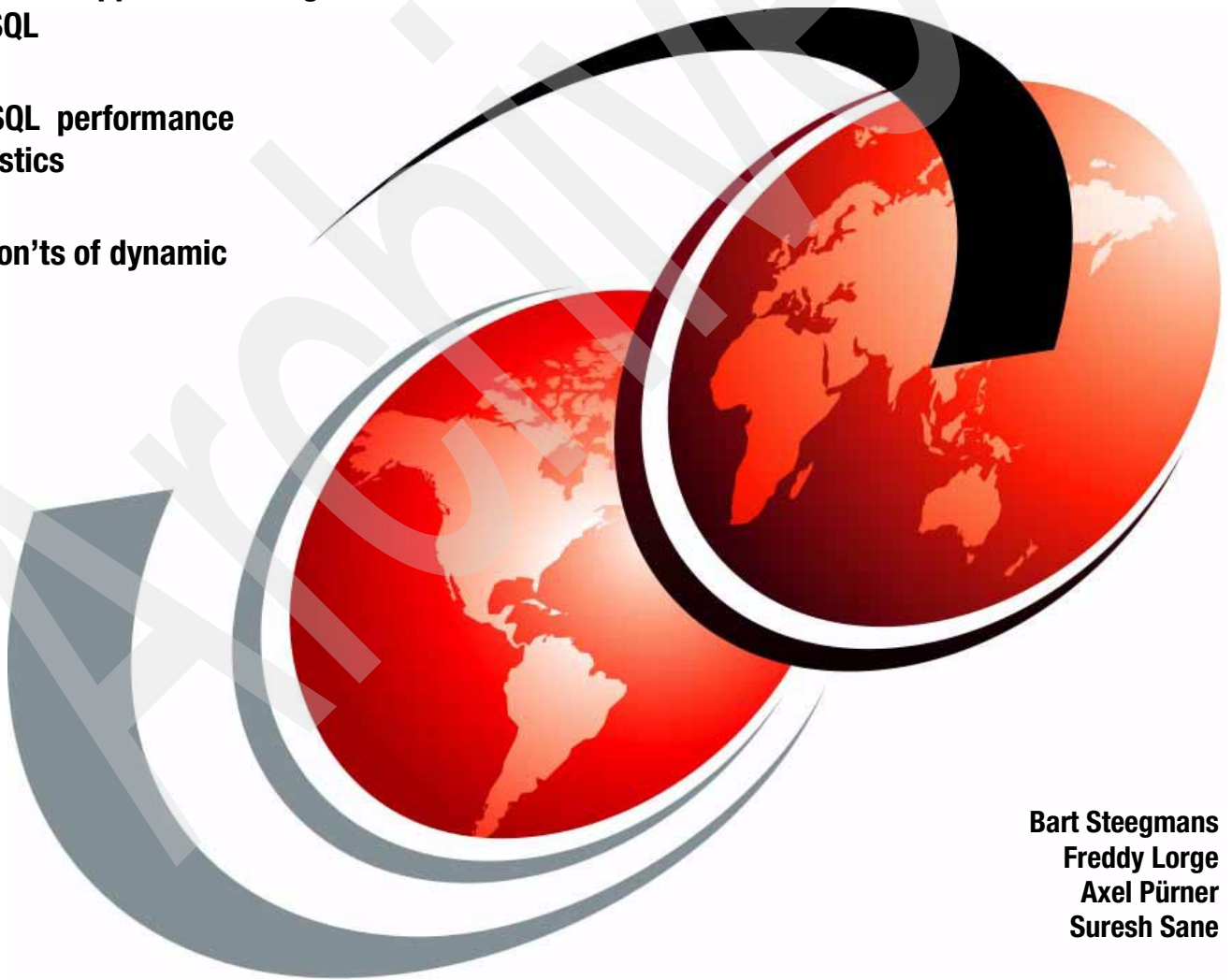
# Squeezing the Most Out of Dynamic SQL

## with DB2 for z/OS and OS/390

How to code an application using dynamic SQL

Dynamic SQL performance characteristics

Dos and don'ts of dynamic SQL



Bart Steegmans  
Freddy Lorge  
Axel Pürner  
Suresh Sane





International Technical Support Organization

**Squeezing the Most Out of Dynamic SQL  
with DB2 for z/OS and OS/390**

May 2002

Archived

**Take Note!** Before using this information and the product it supports, be sure to read the general information in “Notices” on page xv.

### **First Edition (May 2002)**

This edition applies to Version 7 of IBM DATABASE 2 Universal Database Server for z/OS and OS/390 (DB2 for z/OS and OS/390 Version 7), Program Number 5675-DB2.

Comments may be addressed to:  
IBM Corporation, International Technical Support Organization  
Dept. QXXE Building 80-E2  
650 Harry Road  
San Jose, California 95120-6099

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

© Copyright International Business Machines Corporation 2002. All rights reserved.

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

# Contents

<b>Figures</b> .....	ix
<b>Tables</b> .....	xi
<b>Examples</b> .....	xiii
<b>Notices</b> .....	xv
Trademarks .....	xvi
<b>Preface</b> .....	xvii
The team that wrote this redbook .....	xvii
Notice .....	xix
Comments welcome .....	xix
<b>Part 1. Overview of dynamic SQL</b> .....	1
<b>Chapter 1. Introduction</b> .....	3
<b>Part 2. Developing applications using dynamic SQL</b> .....	5
<b>Chapter 2. Dynamic SQL: making the right choice</b> .....	7
2.1 Introduction .....	8
2.2 Programming interfaces .....	8
2.2.1 Embedded SQL .....	9
2.2.2 DB2 callable interfaces to execute SQL statements .....	10
2.3 Classifying SQL based on how the statement is executed .....	11
2.3.1 Static execution .....	12
2.3.2 Dynamic execution .....	12
2.3.3 Combined execution mode .....	12
2.4 Persistence of dynamic SQL .....	12
2.5 Choosing between static and dynamic SQL .....	13
2.6 Choosing between REOPT(VARS) and dynamic SQL .....	16
2.7 Choosing between dynamic SQL and stored procedures .....	17
2.8 Importance of catalog statistics for dynamic SQL .....	17
2.9 Examples of interactive dynamic SQL programs .....	18
2.10 General recommendations on the use of dynamic SQL .....	18
<b>Chapter 3. Developing embedded dynamic SQL applications</b> .....	19
3.1 Introduction .....	20
3.2 Restrictions of dynamic SQL statements .....	20
3.3 The basics of using dynamic SQL .....	21
3.4 What are parameter markers? .....	22
3.5 What is the SQLDA? .....	23
3.6 Variations of embedded dynamic SQL .....	24
3.7 Embedded dynamic SQL programming samples .....	25
3.7.1 Non-SELECT statements without parameter markers .....	25
3.7.2 Non-SELECT statements with parameter markers .....	27
3.7.3 Fixed-list SELECT statements without parameter markers .....	29
3.7.4 Fixed-list SELECT statements with parameter markers .....	32
3.7.5 Varying-list SELECT statements without parameter markers .....	33

3.7.6 Varying-list SELECT statements with parameter markers . . . . .	39
3.8 Using column labels . . . . .	42
3.9 Single, double, and triple SQLDA . . . . .	43
3.10 Special considerations for LOBs and distinct types . . . . .	43
3.11 Conserving storage — right-sizing the SQLDA . . . . .	44
3.12 Handling unknown number and type of parameter markers . . . . .	45
3.13 Handling special attributes . . . . .	47
3.14 Handling SELECT and non-SELECTs together . . . . .	48
<b>Chapter 4. Developing ODBC applications . . . . .</b>	<b>49</b>
4.1 Introduction . . . . .	50
4.2 ODBC, an alternative to embedded SQL . . . . .	50
4.3 ODBC concepts and facilities . . . . .	51
4.3.1 The driver manager, the drivers, and the data sources . . . . .	52
4.3.2 ODBC handles . . . . .	53
4.3.3 Miscellaneous ODBC features . . . . .	54
4.4 The ODBC API . . . . .	55
4.4.1 Functions overview . . . . .	55
4.5 ODBC programming samples . . . . .	63
4.5.1 General format of ODBC function calls . . . . .	63
4.5.2 Programming step 1: Initializing the application . . . . .	64
4.5.3 Programming step 2: Manipulating statement handles . . . . .	66
4.5.4 Programming step 3: Preparing the execution . . . . .	66
4.5.5 Programming step 4: Executing statements . . . . .	70
4.5.6 Programming step 5: Managing execution results . . . . .	71
4.5.7 Programming step 6: Terminating the ODBC application . . . . .	76
4.5.8 Programming step 7: Retrieving diagnostics . . . . .	78
4.5.9 Mapping between embedded statements and ODBC functions . . . . .	78
4.6 Specific features of DB2 ODBC . . . . .	80
4.6.1 DB2 ODBC installation . . . . .	80
4.6.2 Overview of non-standard features . . . . .	81
4.6.3 Differences between the DB2 ODBC driver on z/OS and NT . . . . .	81
4.6.4 Connections supported by DB2 ODBC on z/OS or OS/390 . . . . .	82
4.6.5 Connections supported by DB2 ODBC on UNIX or NT . . . . .	83
4.7 Performance considerations for ODBC applications . . . . .	84
4.7.1 ODBC usage and programming hints . . . . .	84
4.7.2 Improving access to the catalog . . . . .	85
4.7.3 Using features of DB2 on z/OS . . . . .	85
<b>Chapter 5. Developing JDBC applications . . . . .</b>	<b>87</b>
5.1 Introduction . . . . .	88
5.2 JDBC, an alternative to ODBC . . . . .	88
5.3 JDBC concepts and facilities . . . . .	89
5.3.1 The driver manager, the drivers and the data sources . . . . .	89
5.3.2 The JDBC connections . . . . .	90
5.3.3 JDBC classes and interfaces . . . . .	92
5.3.4 JDBC driver types . . . . .	92
5.4 The JDBC API . . . . .	93
5.4.1 Managing connection and statement objects . . . . .	94
5.4.2 Preparing the execution . . . . .	95
5.4.3 Executing statements . . . . .	96
5.4.4 Managing execution results . . . . .	97
5.4.5 Retrieving diagnostics . . . . .	99

5.4.6 Retrieving information about a data source .....	100
5.5 JDBC programming samples .....	100
5.5.1 Initializing driver and database connection .....	100
5.5.2 Preparing and executing SQL statements .....	101
5.5.3 Transactions and termination .....	106
5.5.4 Multi-threading sample .....	107
5.6 Specific features of DB2 JDBC on z/OS .....	110
5.6.1 Properties .....	110
5.6.2 Connections supported by DB2 JDBC .....	110
5.7 SQLJ, an alternative to JDBC .....	111
5.7.1 Preparing SQLJ programs .....	111
5.7.2 SQLJ and dynamic SQL .....	112
<b>Chapter 6. Developing REXX applications using dynamic SQL</b> .....	115
6.1 REXX interface, an alternative to embedded SQL .....	116
6.2 Connecting to DB2 .....	117
6.3 Pre-defined cursors and statements .....	118
6.4 SQL statements not supported .....	118
6.5 Handling the SQLCA .....	118
6.6 Handling the SQLDA .....	119
6.7 Steps in developing REXX dynamic SQL applications .....	120
6.7.1 Non-SELECT without parameter markers .....	120
6.7.2 Non-SELECT with parameter markers .....	121
6.7.3 Fixed-list SELECT without parameter markers .....	122
6.7.4 Fixed-list SELECT with parameter markers .....	124
6.7.5 Varying-list SELECT without parameter markers .....	125
6.7.6 Varying-list SELECT with parameter markers .....	127
6.8 Error handling .....	128
6.9 Choosing the correct isolation level .....	129
6.10 Summary .....	129
<b>Chapter 7. Preparing dynamic SQL applications</b> .....	131
7.1 REOPT(VARS) option .....	132
7.2 KEEP DYNAMIC option .....	133
7.3 RELEASE option .....	133
7.4 DYNAMIC RULES option .....	134
7.5 DEFER(PREPARE) option .....	135
7.6 Specifying access path hints for dynamic SQL .....	135
7.7 Specifying the degree of parallelism for dynamic SQL .....	136
7.8 Obtaining access path information for dynamic SQL .....	137
<b>Part 3. Managing dynamic SQL</b> .....	139
<b>Chapter 8. Dynamic statement caching</b> .....	141
8.1 Overview .....	142
8.2 The different levels of statement caching .....	142
8.2.1 No caching .....	142
8.2.2 Local dynamic SQL cache only .....	143
8.2.3 Global dynamic statement cache only .....	145
8.2.4 Full caching .....	147
8.2.5 Dynamic statement cache summary .....	151
<b>Chapter 9. System controls for dynamic SQL</b> .....	153
9.1 Introduction .....	154

9.2 Controlling storage usage of statement caching . . . . .	154
9.2.1 Controlling local cache storage . . . . .	154
9.2.2 Controlling global cache storage (EDM pool) . . . . .	156
9.3 Other DSNZPARMs . . . . .	159
9.3.1 Resource Limit Facility (RLF) . . . . .	159
9.3.2 DYNAMICRULES . . . . .	159
<b>Chapter 10. Locking and concurrency . . . . .</b>	<b>161</b>
10.1 DBD locks . . . . .	162
10.2 Ambiguous cursors . . . . .	162
10.2.1 What is an ambiguous cursor? . . . . .	162
10.2.2 Using the bind option CURRENTDATA . . . . .	163
10.3 Bind option RELEASE(DEALLOCATE) . . . . .	164
<b>Chapter 11. Governing dynamic SQL . . . . .</b>	<b>167</b>
11.1 Introduction . . . . .	168
11.2 Predictive governing . . . . .	170
11.3 Reactive governing . . . . .	174
11.4 Combining predictive and reactive governing . . . . .	175
11.5 Qualifying rows in the RLST . . . . .	178
<b>Chapter 12. Monitoring, tracing, and explaining dynamic SQL . . . . .</b>	<b>179</b>
12.1 Introduction . . . . .	180
12.2 Using the DB2 Performance Monitor on a workstation . . . . .	181
12.3 Monitoring threads . . . . .	182
12.4 Monitor statistics . . . . .	187
12.4.1 Monitoring the EDM pool . . . . .	190
12.4.2 Monitoring SQL activity . . . . .	192
12.4.3 Monitoring dynamic SQL statements . . . . .	193
12.4.4 Monitoring dynamic statement cache . . . . .	195
12.4.5 Explaining a statement from the cache . . . . .	198
12.5 Using the performance database of DB2 PM . . . . .	199
12.5.1 Overview . . . . .	200
12.5.2 Configuring traces in the Workstation Online Monitor . . . . .	200
12.5.3 Activating traces in the workstation monitor . . . . .	203
12.5.4 Loading trace data into the performance database . . . . .	206
12.5.5 Analyzing performance data . . . . .	210
12.6 Analyzing dynamic SQL using batch reports . . . . .	213
12.7 Analyzing deadlocks in a dynamic SQL environment . . . . .	216
12.8 Identifying users to track resource usage . . . . .	218
<b>Chapter 13. Security aspects of dynamic SQL . . . . .</b>	<b>221</b>
13.1 Is there a security exposure with dynamic SQL? . . . . .	222
13.2 Using DYNAMICRULES(BIND) . . . . .	222
13.3 Using stored procedures to overcome security issues . . . . .	223
13.4 Impact on ODBC/JDBC applications . . . . .	223
13.5 Controlling security without an application server . . . . .	223
13.6 Controlling security with an application server . . . . .	226
13.7 Using network facilities to control database access . . . . .	226
13.8 Security for dynamic SQL in stored procedures . . . . .	227
13.9 Summary . . . . .	228
<b>Chapter 14. Remote dynamic SQL . . . . .</b>	<b>229</b>
14.1 Protocols supported by remote SQL . . . . .	230



14.2 Reusing database access threads .....	230
14.3 Minimizing message flows across the network .....	231
14.4 Miscellaneous considerations .....	233
<b>Part 4. Dynamic SQL in vendor packages .....</b>	<b>235</b>
<b>Chapter 15. Dynamic SQL in ESP packages .....</b>	<b>237</b>
15.1 Overview of product architectures .....	238
15.1.1 SAP architecture .....	238
15.1.2 PeopleSoft and Siebel architectures .....	239
15.2 Statement caching .....	239
15.2.1 Global cache .....	240
15.2.2 Local cache .....	240
15.2.3 Application server cache .....	240
15.3 Use of parameter markers .....	240
15.4 Use of REOPT(VARS) .....	241
15.5 Authorization ID and security .....	241
<b>Part 5. Appendixes .....</b>	<b>243</b>
<b>Appendix A. Comparing plan accounting, package accounting, and dynamic statement         cache statistics .....</b>	<b>245</b>
Comparison of plan, package, and cache information .....	246
<b>Appendix B. Additional material .....</b>	<b>251</b>
Locating the Web material .....	251
Using the Web material .....	251
<b>Abbreviations and acronyms .....</b>	<b>253</b>
<b>Related publications .....</b>	<b>255</b>
IBM Redbooks and Redpapers .....	255
Other resources .....	255
Referenced Web sites .....	256
How to get IBM Redbooks .....	257
IBM Redbooks collections .....	257
<b>Index .....</b>	<b>259</b>

Archived

# Figures

2-1	A typical generic search screen . . . . .	13
3-1	The structure of SQLDA. . . . .	23
3-2	Steps for non-SELECT statements without parameter markers . . . . .	26
3-3	Steps for non-SELECT statements with parameter markers. . . . .	28
3-4	Steps for fixed-list SELECT statements without parameter markers. . . . .	30
3-5	Steps for fixed-list SELECT statements with parameter markers . . . . .	32
3-6	Steps for varying-list SELECT statements without parameter markers. . . . .	34
3-7	SQLDA as populated by PREPARE INTO or DESCRIBE. . . . .	36
3-8	Steps for varying-list SELECT statements with parameter markers . . . . .	40
3-9	SQLDA as populated by DESCRIBE INPUT . . . . .	46
4-1	The ODBC environment. . . . .	53
4-2	ODBC functions to manipulate handles (Part 1) . . . . .	56
4-3	ODBC functions to manipulate handles (Part 2) . . . . .	57
4-4	ODBC functions to prepare the execution and execute SQL statements . . . . .	58
4-5	ODBC functions to manage execution results (Part 1) . . . . .	60
4-6	ODBC functions to manage execution results (Part 2) . . . . .	61
4-7	ODBC functions to manage execution results (Part 3) . . . . .	61
4-8	ODBC functions to manage execution results (Part 4) . . . . .	62
4-9	ODBC functions to retrieve diagnostics . . . . .	62
5-1	The JDBC environment . . . . .	90
5-2	The JDBC driver types. . . . .	93
5-3	JDBC classes and methods to manage Connection and Statement objects. . . . .	95
5-4	JDBC classes and methods to prepare the execution. . . . .	96
5-5	JDBC classes and methods to execute statements. . . . .	97
5-6	JDBC classes and methods to manage execution results. . . . .	99
5-7	JDBC classes and methods to retrieve diagnostics. . . . .	99
5-8	The SQLJ program preparation process . . . . .	112
6-1	Program preparation for an embedded SQL program . . . . .	116
6-2	Program preparation for a REXX program. . . . .	117
6-3	Steps for non-SELECT without parameter markers. . . . .	120
6-4	Steps for non-SELECT with parameter markers . . . . .	121
6-5	Steps for fixed-list SELECT without parameter markers . . . . .	123
6-6	Steps for fixed-list SELECT with parameter markers. . . . .	124
6-7	Steps for varying-list SELECT without parameter markers . . . . .	126
6-8	Steps for varying-list SELECT with parameter markers. . . . .	127
8-1	Dynamic SQL without statement caching . . . . .	143
8-2	Local dynamic statement caching . . . . .	144
8-3	Global dynamic statement caching . . . . .	146
8-4	Full statement caching . . . . .	147
8-5	Comparison of local, global, and full caching. . . . .	151
9-1	Using data space for global dynamic statement caching. . . . .	157
11-1	Logical flow of predictive governing. . . . .	172
11-2	Logical flow of combined predictive and reactive governing . . . . .	176
12-1	DB2 PM for OS/390 main window . . . . .	182
12-2	Monitoring DB2 threads . . . . .	183
12-3	Thread Detail SQL activity . . . . .	185
12-4	Thread detail dynamic SQL statement counters . . . . .	186
12-5	System health window of monitor statistics . . . . .	187

12-6	Statistics detail overview . . . . .	188
12-7	EDM pool statistics . . . . .	191
12-8	SQL DML activity statistics . . . . .	192
12-9	Dynamic SQL cache statistics . . . . .	193
12-10	Monitoring the statement cache: execution statistics. . . . .	195
12-11	Monitoring the statement cache: statement strings . . . . .	197
12-12	Details of cached statement. . . . .	198
12-13	Using Visual Explain for a cached statement. . . . .	199
12-14	Configure a DB2 trace . . . . .	200
12-15	Specification of trace data . . . . .	201
12-16	Stop conditions for a trace . . . . .	202
12-17	Generated START TRACE commands . . . . .	203
12-18	Trace activation window. . . . .	204
12-19	Started trace collecting data . . . . .	205
12-20	Confirmation to save collected data. . . . .	206
12-21	Trace status after termination . . . . .	206
12-22	The accounting tables of the performance database. . . . .	207
12-23	The record trace tables of the performance database. . . . .	207
12-24	Logical flow of loading trace data into the performance database. . . . .	208
12-25	Relations between relevant tables of the performance database . . . . .	212
13-1	Using application servers to improve security . . . . .	226
13-2	Security implications of dynamic SQL in a stored procedure. . . . .	227
15-1	SAP architecture overview. . . . .	238
15-2	PeopleSoft and Siebel architecture overview . . . . .	239

# Tables

2-1	Examples of dynamic SQL programs . . . . .	18
3-1	What dynamic SQL cannot do . . . . .	21
3-2	How many SQLVARs do I need? . . . . .	43
3-3	Contents of SQLDA when USING BOTH is not specified . . . . .	43
3-4	Contents of SQLDA when USING BOTH is specified . . . . .	44
3-5	Contents of minimum SQLDA . . . . .	45
4-1	Mapping between embedded non-dynamic statements and ODBC functions. . . . .	79
4-2	Impact of MULTICONTEXT and CONNECTTYPE in the z/OS environment . . . . .	83
5-1	Conversion table for setXXX methods . . . . .	102
5-2	Common getXXX methods . . . . .	103
7-1	How runtime behavior is determined . . . . .	134
7-2	What the runtime behavior means . . . . .	135
10-1	Lock avoidance factors . . . . .	164
12-1	Members for table creates and descriptions of the accounting data . . . . .	207
12-2	Members for table creates and descriptions of the record trace data . . . . .	208
12-3	Load control members for the performance database . . . . .	209
13-1	Security implications without application server . . . . .	224
13-2	How is run-time behavior determined? . . . . .	227
A-1	Summary of trace record information . . . . .	250

Archived

# Examples

4-1	Creating the environment handle. . . . .	64
4-2	Creating a connection handle and connecting to databases. . . . .	65
4-3	Connecting to database through SQLConnectDrive(). . . . .	66
4-4	Manipulating statement handles. . . . .	66
4-5	Preparing statements. . . . .	66
4-6	Describing parameters. . . . .	67
4-7	Binding parameters to program variables. . . . .	68
4-8	Binding a parameter to an array of variables. . . . .	69
4-9	Binding parameters to enter value at execution time. . . . .	70
4-10	Executing statements. . . . .	71
4-11	Counting the number of updated, deleted, or inserted rows. . . . .	71
4-12	Specifying a parameter value at execution time. . . . .	71
4-13	Retrieving data with bindings. . . . .	72
4-14	Retrieving result sets in an array (column-wise binding). . . . .	73
4-15	Retrieving result sets in an array (row-wise binding). . . . .	74
4-16	Retrieving data without bindings. . . . .	75
4-17	Use of positioned updates and deletes. . . . .	76
4-18	Disconnecting from databases and freeing handles. . . . .	77
4-19	Retrieving diagnostics. . . . .	78
5-1	Load of the DriverManager. . . . .	100
5-2	Connecting to a DB2 subsystem. . . . .	101
5-3	Creating a Statement object. . . . .	101
5-4	Preparing a statement. . . . .	102
5-5	Using ParameterMetaData to describe input parameters. . . . .	103
5-6	Executing a query. . . . .	103
5-7	Retrieving the result of a query. . . . .	104
5-8	Using ResultSetMetaData to describe a query result. . . . .	104
5-9	Using a cursor for positioned updates. . . . .	105
5-10	Calling a stored procedure. . . . .	105
5-11	Receiving SQLCODE, SQLSTATE and SQL message. . . . .	106
5-12	Closing a database connection. . . . .	106
5-13	Commit. . . . .	107
5-14	Rollback. . . . .	107
5-15	Multi-threading sample. . . . .	107
6-1	Setting up the REXX DB2 environment. . . . .	117
6-2	Connecting to DB2 via call attach. . . . .	117
6-3	Error handling in a REXX program. . . . .	128
11-1	DDL to create an RLST. . . . .	168
11-2	Creating the RLST index. . . . .	169
11-3	A definition for predictive governing. . . . .	173
11-4	A definition for reactive governing. . . . .	175
11-5	SQLERRMC after SQLCODE +495. . . . .	176
11-6	Full SQLCODE +495 error message. . . . .	176
11-7	Combining predictive and reactive governing. . . . .	177
12-1	FILE subcommand for accounting data. . . . .	208
12-2	FILE subcommand for record trace data. . . . .	209
12-3	Sample load job for accounting data from FILE data set. . . . .	209
12-4	Sample query. . . . .	211

12-5	DB2 PM command to list IFCID 22 and 63 .....	213
12-6	Listing SQL statements and miniplans. ....	214
12-7	PLAN_TABLE and DSN_STATEMNT_TABLE info .....	215
12-8	Deadlock messages in the system log. ....	216
12-9	Deadlock record (IFCID 172) .....	216
12-10	DB2 PM command to list SQL statements for a list of threads .....	217
12-11	Listing the SQL statements issued by a list of threads .....	217
A-1	Comparing plan, package and cache info .....	246



# 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.*

**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.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.


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 illustrates 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. 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 IBM's application programming interfaces.

## Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	MVS/ESA™	RMF™
BookMaster®	MVST™	S/390®
CICS®	OS/390®	SAA®
DB2 Connect™	Parallel Sysplex®	SP™
DB2 Universal Database™	Perform™	SQL/DS™
DB2®	QMF™	System/390®
DRDA®	RACF®	TME®
IBM®	Redbooks(logo)™ 	WebSphere®
IMS™	Redbooks™	z/OS™

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

Lotus®	Word Pro®
--------	-----------

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation 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.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

# Preface

Dynamic SQL has been available since release 1 of DB2. In fact it was initially intended as the primary interface to execute SQL statements against DB2. The fact that statements have to be prepared at execution time, and the CPU overhead associated with the prepare process, meant that dynamic SQL was hardly usable in a high volume on-line transaction environment, especially with the processor speed of the hardware in those days.

Although hardware as well as DB2 have come a long way since those early days, the general opinions about the usage of dynamic SQL have not evolved with it. There are still many DB2 installations where you are not allowed to mention the words “dynamic SQL” in an OLTP environment.

This IBM Redbook explodes some of the myths and misconceptions surrounding dynamic SQL. It presents a balanced discussion of issues such as complexity, performance, and control, and provides a “jump-start” to those venturing into this somewhat under-utilized area.

After reading this IBM Redbook, you should have a better idea of the current possibilities and usage of dynamic SQL with DB2 for z/OS and OS/390, enabling you to make more informed decisions about when and when not to use dynamic SQL.

## The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

**Bart Steegmans** is a DB2 Product Support Specialist from IBM Belgium currently on assignment with the ITSO in San Jose. He has over 13 years of experience in DB2. Before joining IBM in 1997, Bart worked as a DB2 system administrator in a banking and insurance group. His areas of expertise include DB2 performance, database administration, and backup and recovery.

**Freddy Lorge** is an IBM certified IT Specialist in Belgium/luxembourg. He has 16 years of experience in data management on several database systems. Before joining IBM in 1998, he worked for companies in the petrochemical and chemical industry. He holds two university degrees, in Physics Sciences and in Management, and a Master's degree in Computer Sciences. His areas of expertise include data modelling and performance tuning. He is currently leading a cross-platform team supporting database systems for Belgian customers in IBM Global Services/Strategic Outsourcing.

**Axel Pürner** is an independent consultant in Germany. He has more than 25 years of experience working with databases, and he specializes in DB2 database administration and database design. Before he became an independent consultant, he worked as a database administrator in a manufacturing company and as a consultant of a major German software company. He holds two university degrees in electrical engineering and economics. His areas of expertise include data modelling, application development, DB2 administration and performance tuning. He is also familiar with DB2 UDB for distributed platforms.

**Suresh Sane** is a Database Architect with DST Systems in Kansas City, Missouri, USA. He has been working with DB2 as an application developer, a DBA, and a performance specialist for the last 17 years. He holds Electrical Engineering and Management degrees. He is a frequent speaker at the North American conferences of IDUG with 7 presentations and an article in the IDUG Solutions Journal to his credit.

Thanks to the following people for their contributions to this project:

Paolo Bruni  
Emma Jacobs  
Yvonne Lyon  
Deanna Polm  
International Technical Support Organization, San Jose Center

Rich Conway  
IBM International Technical Support Organization, Poughkeepsie Center

Bill Bireley  
Jerry Goldsmith  
James Guo  
Bruce Hayman  
Roger Miller  
Manfred Olschanowsky  
Mary Petras  
Akira Shibamiya  
Hugh Smith  
Horacio Terrizzano  
Annie Tsang  
Casey Young  
IBM Silicon Valley Lab, San Jose

Pamela Hoffman  
IBM/Siebel Competency Center

Mike Curtis  
IBM/PeopleSoft Competency Center

Namik Hrle  
Norbert Jenninger  
IBM Germany Entwicklung GmbH

Martin Packer  
IBM United Kingdom

Svetlana Sicular  
Siebel Systems, Inc.

## Notice

This publication is intended to help managers and professionals understand and evaluate the usage of dynamic SQL in the IBM DATABASE 2 Universal Database Server for z/OS and OS/390 products. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM's DATABASE 2 Universal Database Server for z/OS and OS/390. See the PUBLICATIONS section of the IBM Programming Announcement for IBM's DATABASE 2 Universal Database Server for z/OS and OS/390 for more information about what publications are considered to be product documentation.

## Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- ▶ Use the online **Contact us** review redbook form found at:  
[ibm.com/redbooks](http://ibm.com/redbooks)
- ▶ Send your comments in an Internet note to:  
[redbook@us.ibm.com](mailto:redbook@us.ibm.com)
- ▶ Mail your comments to the address on page ii.

Archived



# Part 1

## Overview of dynamic SQL

In this part of the book, we introduce the basic concepts of dynamic SQL and explain some of the reasons for using it.

This part of the book is a general introduction to this IBM Redbook and summarizes the subjects treated in the rest of the book.

Archived



# Introduction

The time has come to set the record straight. While the use of dynamic SQL in vendor packages continues to rise at a steep rate, the perception of most DB2 professionals remains that dynamic SQL is complex, resource-intensive, and difficult to manage. How true are these perceptions?

In this IBM Redbook we explode some of the myths and misconceptions surrounding dynamic SQL. We present a balanced discussion of issues such as complexity, performance, and control, and provides a “jump-start” to those venturing into this somewhat under-utilized area.

What is dynamic SQL? When is it appropriate? How do I develop an application in COBOL, REXX, or C using ODBC containing dynamic SQL? How do I obtain the best performance from it? How do Enterprise Solution Packages (ESP) exploit dynamic SQL? How do I manage and control it? These and similar questions are the focus of this book.

It is not easy to understand what the differences between static and dynamic SQL are, especially if you consider alternative database interfaces such as Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC).

**Part 2, “Developing applications using dynamic SQL” on page 5** explains how to develop an application with dynamic SQL using any of the available programming interfaces.

First we look at the different Application Programming Interfaces (APIs) that can be used to develop SQL applications trying to classify SQL statements as static or dynamic, based on the way they are coded and the way they are executed against DB2.

We also provide an overview of the important choices you have to make when deploying an application using dynamic SQL on a DB2 for z/OS or OS/390 subsystem.

Then we dive into the details of how to code application programs using the different interfaces that are available to you:

- ▶ Using embedded dynamic SQL
- ▶ Using ODBC
- ▶ Using JDBC
- ▶ Using DSNREXX

We also highlight the different bind options that are either specific to dynamic SQL or have an impact on the way the application or the system behaves when you use dynamic SQL.

The areas where dynamic SQL plays a role when accessing remote databases or when coming into DB2 through the network are also discussed. Using remote dynamic SQL introduces a few particularities that you have to be aware of in order to get the most out of your applications. How to reuse threads and how to minimize the number of messages across the network are important issues that you have to deal with in a network computing environment that uses dynamic SQL.

**Part 3, “Managing dynamic SQL” on page 139** focuses on how to manage dynamic SQL, mostly from a systems point of view. Following are some of the topics we consider.

When dynamic statement caching became available in DB2 Version 5, all of a sudden dynamic SQL looked more attractive than ever. However, because dynamic statement caching comes with many options like local caching, global caching, full caching, and a set of DSNZPARMs that can help to tune the caching behavior on your environment, numerous misconceptions exist about the usage of the dynamic statement cache. Hopefully, things will be more clear after reading this book.

We also consider how locking behavior is different when using the various types of dynamic SQL, with and without statement caching.

When you give users the opportunity to launch their own SQL statements, it might be a good idea to control the resource usage of these queries to prevent them from using extensive amounts of resources. Even though there are many ways to control resource usage (for example, by using WLM controls), in this book we focus on the ways DB2 for z/OS offers to control dynamic SQL statement resource usage via the DB2 governor facility.

Another important aspect of controlling resource usage is monitoring. Monitoring dynamic SQL provides us with several challenges; for example, how do you know which statements used most of the CPU? Since there is no package or plan that contains all possible SQL statements, the only way to obtain the SQL statement text is by means of tracing. We show which traces you will need and how to exploit this information, as well as describing the additional tracing possibilities that are available to you when using the dynamic statement cache (in combination with a DB2 monitor that can exploit this information, like DB2 PM).

One of the main reasons why many people are reluctant to use dynamic SQL is because of the authorization schema. By default, individual user IDs or secondary authorization IDs need the necessary authorization on every object that the users have to access. These authorizations have to be managed properly in order not to create any security exposures. When the user has access to a DB2 object, he can also access that object using other programs that allow him to execute dynamic SQL, outside of the control of an application that does the necessary checking to insure validity and integrity of the data. We discuss the potential authorization pitfalls of using dynamic SQL and offer alternatives on how to reduce or overcome these, for example, by using different bind options, or embedding the dynamic SQL application in a stored procedure.

**Part 4, “Dynamic SQL in vendor packages” on page 235** provides a brief description of how some of the major enterprise solution packages like PeopleSoft, SAP, and Siebel use dynamic SQL in their applications, as well as how they manage to get good performance out of dynamic SQL.

**Part 5, “Appendixes” on page 243** includes some supplementary information that you may find useful. For example, we compare plan accounting, package accounting, and dynamic cache statistics, and we refer you to our coding examples, which are available on the Web.

# Developing applications using dynamic SQL

In this part of the book, we discuss the following topics:

- ▶ The choices you face when deciding whether or not to use dynamic SQL in your environment
- ▶ How to develop applications that use dynamic SQL, using any of the available programming interfaces:
  - Developing embedded dynamic SQL applications
  - Developing ODBC applications
  - Developing JDBC applications
  - Developing REXX applications using dynamic SQL
- ▶ How to prepare applications using dynamic SQL

Archived



## Dynamic SQL: making the right choice

In this chapter we introduce the different types of dynamic SQL as a possible interface to access DB2 data along with other alternatives, and we discuss some of the issues surrounding its use.

## 2.1 Introduction

What exactly constitutes dynamic SQL? Why should you consider using it? Why and why not? What are some of the issues surrounding its use? This chapter introduces these issues and discusses alternatives to using dynamic SQL. It is not simply a question of dynamic versus static SQL, but dynamic SQL versus a variety of alternatives.

A clear distinction between static SQL and dynamic SQL does *not* exist. As will become evident from the discussion below, there are a few variations of each, and depending on the host language and hardware platform, some overlap between them exists.

In several cases, a specific variation of dynamic SQL may be only one of several alternatives available. These alternatives include the use of static SQL with various BIND parameters as well as the deployment of stored procedures. This chapter explores these possibilities, briefly outlines the variations of dynamic SQL and introduces some of the issues that must be considered before the appropriate choice is made.

## 2.2 Programming interfaces

DB2 for z/OS allows you to code SQL statements in different ways, using different programming interfaces.

In this section we look at the different interfaces you can use to *code* SQL and how this determines the way the statement is prepared and executed by DB2.

From the *programming or coding point of view*:

- ▶ An SQL statement or SQL statement string is considered *static* if it is “hard-coded” in the program. It does not matter if the statement text is stored in program variable or coded on a DECLARE CURSOR statement; if you can see the statement in the program, it is considered static from the programming point of view. The result is that you have to go through the program deployment cycle (compile and link-edit) every time any of those “hard-coded” SQL statements changes.
- ▶ An SQL statement is considered *dynamic* when the statement text is provided (for example on the terminal by the user) or assembled (for example based on fields marked by the user) at execution time. In this case, the statements are built at runtime and must be explicitly prepared at runtime, which is done by a dedicated statement or function. Dynamic SQL allows the execution of a large set of different statements in one program, without having to go through the deployment cycle for each.

It is important to understand that the definitions of static and dynamic SQL given above are related to the way the statements are built in the program. Another possibility to classify SQL is *by the way the execution is done*. Such a classification is explained in 2.3, “Classifying SQL based on how the statement is executed” on page 11.

There are two different families of programming interfaces, which allow both the use of “hard-coded” and “dynamically built” SQL statements:

- ▶ Embedded SQL
- ▶ Callable interfaces

These two families are described from the DB2 perspective in the next two sections.

## 2.2.1 Embedded SQL

The classical interface for common programming languages is *embedded SQL* providing capabilities for both static and dynamic SQL. Embedded means the SQL statements are mixed with a host language inside the program. To indicate the fact that this is a non-programming language statement, embedded SQL statements are preceded by a special keyword; **EXEC SQL** is used by most programming languages, **#sql** is used by SQLJ.

The result is that the program deployment cycle must contain a pre-compilation step (or use a language co-processor) and a bind step, as explained next.

### Classic static SQL

Here, SQL statements are embedded and “hard-coded” in the application program. The SQL statement text is fixed and visible in the application program. The only dynamic variation of those statements lies in the usage of host variables. The use of host variables enables the program to execute the same statement with different values, but the statement itself is fixed.

Before such a program can be executed, it must be precompiled (by the DB2 precompiler or the coprocessor). This turns the SQL statements (that start with the **EXEC SQL** clause) into host language comments and generates host language **call** statements to invoke DB2. This process also produces a DBRM containing a list of all the SQL statements and the host variables used.

On the DB2 side, the DBRM for the application must be bound to the DB2 subsystem that will execute the program using the **BIND** command. This creates a package and/or plan that contains constructs that represent the SQL statements inside the DB2 system.

Because every program contains different (“hard-coded”) SQL statements, each program containing embedded static SQL has to have its own plan or package in DB2.

This interface is supported by various attach facilities; batch, TSO, CICS, IMS, CAF, RRS using languages like COBOL, C, Assembler, FORTRAN, and PL/I.

### Java static SQL interface (SQLJ)

SQLJ is an application programming interface (API) for embedded static SQL in Java programs. SQLJ statements are also “hard-coded” in the program. The use of host expressions enables the program to execute the same statement with different values, but the statements themselves are fixed.

Before such a program can be executed, it must be processed by the SQLJ translator (by executing the **sqlj** command). The SQLJ clauses are transformed by the translator into calls to the runtime environment. After translation, the Java program can be compiled normally. The translator produces *profile* files which contain information about the database schema that must be accessed. The profiles are used at execution time by the SQL runtime environment.

The profiles must be customized if the use of vendor-specific features or optimization is needed. With DB2, the profiles can be used to create DBRMs (by using the **db2prof** command) that can later be bound into DB2 packages or plans. This way, SQLJ can run as real static SQL at execution time on a DB2 subsystem, including the performance and authorization benefits that go hand in hand with static SQL on DB2 for z/OS.

SQLJ itself has no syntax defined to handle dynamic SQL statements. A Java application that wants to execute dynamic SQL statements must use JDBC. An SQLJ program that has not been customized and bound will treat the statically coded SQL as dynamic at execution time and process it via JDBC. This is transparent to the user.

## Embedded dynamic SQL

Embedded dynamic SQL applications place the full SQL statements in language host variables. To prepare and run these SQL statements, these programs issue dedicated embedded statements like PREPARE and EXECUTE. Here we use the term “dedicated” to distinguish these statements from real static SQL statements like DECLARE. The dedicated SQL, although statically coded, help to prepare and execute the actual dynamic SQL statements. Just like an application containing static SQL, an application containing embedded dynamic SQL must be precompiled and bound before it can be executed. All dedicated SQL statements (like the PREPARE and EXECUTE statements) must be bound into an application package or plan. The (dynamic) SQL statements contained in the host variables are prepared and executed when invoked (at runtime).

*Interactive dynamic SQL* on DB2 for z/OS, is a special case of embedded dynamic SQL and refers to the entry of SQL statements (that can be prepared dynamically) by a user through programs such as SPUFI and QMF. DB2 prepares and executes them as dynamic SQL statements. Since the statements you enter using SPUFI or QMF are kept in host variables, no application precompile and BIND is required when the SQL statement is changed. The dedicated SQL statements (PREPARE, DESCRIBE, etc.) that SPUFI and QMF use, are bound into packages and/or plans at DB2 or QMF installation time. More information about how to use SPUFI and QMF can be found in *DB2 Universal Database for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933 and *Query Management Facility Version 7 Using QMF*, SC27-0716.

## Deferred embedded dynamic SQL

This type of dynamic SQL is only used by the DB2 private protocol (DB2 for z/OS only) to access remote data. For example, an application executing on DB2 subsystem DB2A references an object PROD.EMPLOYEE on subsystem DB2B. Using the DB2 private protocol this can be achieved through the use of an ALIAS or by specifying the table as a so-called three-part name (DB2B.PROD.EMPLOYEE). Such a statement has characteristics of static as well as dynamic SQL. Like static SQL, it is embedded and “hard-coded” in an application, but like dynamic SQL it is prepared at runtime on subsystem DB2B. When using the DB2 private protocol, there is no plan or package on DB2B that contains the statements that are executed there (coming from a remote system like DB2A). At execution time, the statement is sent from DB2A to DB2B and is prepared and executed like dynamic SQL.

DB2 private protocol can only be used when communicating between two DB2 for z/OS subsystems. It has not been enhanced to support some of the new functions in DB2 like LOBs and user-defined functions. The recommendation is to use DRDA for new applications, and begin migrating existing private protocol applications to DRDA.

The way DB2 processes SQL which accesses declared temporary tables (DTTs) is very similar to the way that statically coded embedded SQL using the DB2 private protocol is executed. Even though the DB2 manuals do not classify these statements as deferred embedded dynamic SQL, they are also coded as embedded static SQL, but dynamically prepared (incrementally bound) at runtime. DB2 has to use this approach since the table (DTT) is only created at execution time. At bind time, the table does not exist, and therefore DB2 cannot create an executable format of the statement at that time.

## 2.2.2 DB2 callable interfaces to execute SQL statements

An alternative to embedded SQL, is the usage of a callable interface to request DB2 services.



The first two programming interfaces are based on the Open Group (formerly X/Open) call-level interface specification and provide a consistent API for C and C++ applications (ODBC) and for Java applications (JDBC). The third form of callable interface is used by REXX programs to request DB2 services. Actually the DSNREXX interface is not really a callable interface. It has characteristics of both, the embedded SQL interface (the way the SQL is coded) and a callable interface (the way the SQL is executed). For ease of use, we classify it as a callable interface.

When using a call-level interface, SQL statements, whether “hard-coded” in the program or dynamically built at run-time, are always passed to the database management system via the call-level interface at run-time. ODBC, JDBC or REXX programs *themselves* must not be precompiled. At installation time, the system administrator binds a set of general packages and/or plans (usually with different isolation levels) that are invoked by every application.

### **Executing SQL statements via ODBC**

DB2 Open Database Connectivity (DB2 ODBC) is an API available to C and C++ applications via function calls. The application issues a function call at execution time to connect to DB2, to issue SQL statements and to obtain data or status information. No precompilation or binding of the application itself is required.

### **Executing SQL statements via JDBC**

JDBC is an API for Java applications to access any relational database. Like ODBC, it utilizes function calls and does not require precompilation or binding.

### **Executing SQL statements via the REXX interface**

The REXX language interface for DB2 for z/OS provides a simple method for developing an application very quickly. It can use the RRS or CAF attach to connect to the DB2 subsystem. DSNREXX uses a simple set of statements that look very similar to embedded dynamic SQL and is in that respect more like embedded SQL than a callable interface. However, REXX programs accessing DB2 must not be precompiled and come with a predefined set of DB2 packages, very similar to callable interfaces like ODBC and JDBC.

Some vendors also incorporate other APIs that are based on the ODBC API. These will not be discussed in this book.

## **2.3 Classifying SQL based on how the statement is executed**

Another way to classify SQL statements is by the way they are executed by the database management system.

It is important to understand in the explanation below that a dynamically built statement must always be run in the dynamic execution mode (that is, it must be prepared at runtime). A “hard-coded” statement can be executed with either the static execution mode or the dynamic execution mode.

### 2.3.1 Static execution

It is specific to DB2, that the precompiler for embedded SQL generates a *database request module* (DBRM). This DBRM contains the SQL statements which are “compiled” in a **BIND** process (prepare process) generating a *package* or *plan* with the statement strings and the related access paths. Those bound statements are executed directly without additional preparing. This *execution mode* is called *static*. Only precompiled static SQL statements benefit from this BIND. The executable form of a statement that is executed statically is generated at BIND time, during the program preparation process, not at execution time.

### 2.3.2 Dynamic execution

SQL statements that have not been bound before they are executed (either because they are dynamically built and prepared via embedded statements, or passed in via a call interface) are prepared at runtime. This *execution mode* is called *dynamic*. No executable form of the statement exists before the statement is executed. (Here we ignore statement caching on purpose, to avoid overcomplicating the discussion.) As a result, all statements executed by any of the available callable interfaces (ODBC, JDBC, and REXX) are considered dynamic SQL statements from the DBMS point of view. All statements, even those “hard-coded” in the program, are passed to DB2 (prepared) at execution time and are considered dynamic.

### 2.3.3 Combined execution mode

There exist also a combination of both methods for *deferred embedded* SQL statements, which are neither fully static nor fully dynamic. Deferred embedded SQL statements are used for the DB2 private protocol access to remote data. (See Chapter 14, “Remote dynamic SQL” on page 229 for detailed information.)

Another “hybrid” form is when a static SQLJ program does not have a customized profile that was bound against the DB2 system. In this case, even though the statements are coded as static SQLJ statements, they are dynamically prepared and executed using the dynamic JDBC interface.

In this book we focus on both of the following:

- ▶ The use of dynamic SQL statements, like the coding of dedicated embedded statements or specific functions to prepare, describe, and execute statements built at runtime.
- ▶ The use of dynamic execution mode; how statements are dynamically executed against the DB2 subsystem, independently from the API (using the embedded API as well as the callable interfaces).

## 2.4 Persistence of dynamic SQL

In general, the prepared form of a dynamic SQL statement is discarded after the logical unit of work (LUW) containing the statement has been committed, except when you use the **KEEPDYNAMIC** bind option. Depending on whether or not the (global) dynamic statement cache is active, either the statement string or the full prepared statement is kept across **COMMITs**.

There are various forms of caching that either eliminate the need to prepare completely or allow for a cheaper prepare operation. This is controlled by various options including the **CACHEDYN** DSNZPARM, **KEEPDYNAMIC** bind option and the sizes of various pools. This is discussed in detail in Chapter 8, “Dynamic statement caching” on page 141 and is a major factor that affects performance.

## 2.5 Choosing between static and dynamic SQL

Consider an application program that must return a list of employees for a wide variety of user requirements. For example, the user may wish to list all employees for a department or a list of all employees whose last name starts with some set of characters. In general, the user requirements can be captured via a screen that looks something like the one shown in Figure 2-1.

	<u>FROM</u>	<u>TO</u>	<u>SORT ORDER</u>
LAST NAME	_____	_____	_____
FIRST NAME	_____	_____	_____
WORK DEPT.	_____	_____	_____
HIRE DATE	_____	_____	_____
SALARY	_____	_____	_____
BONUS	_____	_____	_____
COMMISSION	_____	_____	_____

Figure 2-1 A typical generic search screen

There are three main alternatives to developing an application to process such requests and we discuss the merits of each approach below:

- Static SQL with a cursor for each combination:

Using this approach we code and process a cursor for each possible combination. Since the user may specify any or all criteria, the number of such cursors is prohibitively large (1 with no criterion specified, 7 with exactly 1 criterion specified, 21 with exactly 2 criteria specified, etc., for a total of 128 without considering the sort order) which makes the code cumbersome, since we need a DECLARE, OPEN, FETCH, and CLOSE for each such cursor.

- Static SQL with generic search criteria:

Using this approach, you code a SELECT statement with all criteria specified as a set of BETWEEN predicates. Ordering the data needs to be handled by other means, or not handled at all. Here, the SQL looks like this:

```
SELECT ...
FROM      DSN8710.EMP
WHERE     LASTNAME      BETWEEN :LO-LASTNAME  AND :HI-LASTNAME
AND       FIRSTNAME     BETWEEN :LO-FIRSTNME  AND :HI-FIRSTNME
AND       WORKDEPT      BETWEEN :LO-WORKDEPT  AND :HI-WORKDEPT
AND       SALARY        BETWEEN :LO-SALARY    AND :HI-SALARY
AND       BONUS         BETWEEN :LO-BONUS     AND :HI-BONUS
AND       COMMISSION    BETWEEN :LO-COMMISSION AND :HI-COMMISSION
```

Prior to the execution of this cursor, the host variables are populated based on user requirements. For example, if a user specifies the range of \$50,000 to \$100,000 for the salary, LO-SALARY contains 50,000 and HI-SALARY contains 100,000. If the user specifies no criteria for salary, LO-SALARY contains 0 and HI-SALARY contains some maximum permissible number.

While this approach is simple to develop and uses static SQL, the performance of such a statement is generally far from optimal, since the access path is unaware of the criteria entered by the user. Substantial performance improvement can generally be obtained using the REOPT(VARS) bind option (see section 7.1, “REOPT(VARS) option” on page 132).

► **Dynamic SQL:**

Depending on the criteria entered and the sort sequence requested by the user, a dynamic SQL statement is built and processed. For example, if the user requested all employees in department ‘E21’ whose last name starts with ‘S’ earning more than \$10,000.00 in commission, sorted by last name and first name, the SQL looks like this:

```
SELECT ...
FROM      DSN8710.EMP
WHERE     WORKDEPT = 'E21'
AND       LASTNAME LIKE 'S%'
AND       COMMISSION > 10000
ORDER BY  LASTNAME, FIRSTNAME
```

A different statement is built for a different request. This is completely flexible and the access path is optimal (but it incurs the cost of PREPARE multiple times).

At least in cases like this, where a highly flexible application is needed, dynamic SQL should be seriously considered as a possible solution. You should also consider the following factors before deciding whether or not dynamic SQL is appropriate for your needs.

## Flexibility

Static SQL can be made more flexible by using host variables instead of literals. For example, the following code fragment in COBOL can be used to update the salaries by various amounts for various employees determined at execution time (both perhaps read from a file):

```
EXEC SQL
  UPDATE  DSN8710.EMP
  SET     SALARY = :NEW-SALARY
  WHERE   EMPNO = :EMPNO
END-EXEC.
```

However, if the user can update any column (perhaps of any table) or can select one or many columns of a table (perhaps in a different sequence), the number of variations of static SQL to handle all such combinations becomes prohibitively large. Such business needs that must be highly flexible can only be met by using dynamic SQL.

## Complexity

In general, dynamic SQL is more complex to develop than static SQL. However, the degree of complexity depends on the type of dynamic SQL that must be handled. In Section 3.6, “Variations of embedded dynamic SQL” on page 24, we discuss the various forms of dynamic SQL in traditional programming languages. Non-SELECT statements without parameter markers add only slightly to the complexity while the most complex variation (varying-list SELECT with parameter markers) adds a considerable amount of complexity, especially in COBOL. However, the additional complexity of dynamic SQL should not be the determining factor in whether or not to choose for dynamic SQL. Reasonably skilled programmers should have no problems coding dynamic SQL programs.

## Performance

In general, a dynamic SQL statement must be prepared before it is executed. The resources consumed for this step have the potential to make dynamic SQL perform worse than static SQL. However, caching of dynamic SQL (see Chapter 8, “Dynamic statement caching” on page 141) can substantially reduce this overhead. Furthermore, when the data is skewed, dynamic SQL can actually perform better than static SQL as explained in the next section.

One important consideration for performance is the cost of preparing the statement as compared to the cost of running it. For example, if the statement is to be prepared once (say 100 ms of CPU) and run once (many seconds of CPU), the prepare cost is negligible.

## Presence of non-uniform distribution (NUD) of data

When static SQL is used for accessing non-uniform data, the access path determined at BIND time can be sub-optimal. Consider, for example, static SQL that accesses all employees for a specified department (as a host variable). If the department has a very small number of employees, an index access may be more efficient, while a table space scan may be more efficient for a department with a substantial number of employees.

Since dynamic SQL knows the value of the department during the prepare phase (assuming literal are passed in the text instead of parameter markers), it can produce a more efficient access path and cost of the prepare can be more than offset. In cases like this, static SQL can still be used and the plan or package bound with REOPT(VARS). This is the focus of section 2.6, “Choosing between REOPT(VARS) and dynamic SQL” on page 16.

## Security

There are two security aspects you need to consider:

- The cost of maintaining authorizations:

When using dynamic SQL, a user may require additional authorizations to the objects accessed. See Chapter 13, “Security aspects of dynamic SQL” on page 221 for details.

The use of the DYNAMICRULES bind option and the use of secondary authorization IDs can help to alleviate the authorization maintenance cost.

- Security exposure:

Let us quickly review the minimum authorization needed for a user to run a program containing static SQL. In this case, the user only needs an EXECUTE privilege on the plan or package containing the SQL. The user does not require authorization to execute these statements outside the control of the program. For example, consider a program containing static SQL that updates the salary information for any employee. A user with EXECUTE authority on the plan or package can issue these statements by running the program, but will not be able to do so via QMF, SPUFI or any external means. For dynamic SQL, the user normally has explicit authorization to UPDATE the EMPLOYEE table (unless the bind option DYNAMICRULES(BIND) is used). This applies only to embedded dynamic SQL and is discussed in detail in Chapter 13, “Security aspects of dynamic SQL” on page 221. This way the user can also access the EMPLOYEE table outside the program, for example via SPUFI.

## Auditing

Unlike static SQL that exists in source libraries, dynamic SQL is built at runtime. The source of such SQL may be external and controlled (for example, a file or a DB2 table) but it is impossible to ensure that the statement is not modified prior to its execution. While methods to trap SQL statements once they have executed exist, it is not possible to determine, in advance, all such statements. If you have very stringent audit requirements for certain applications, do not implement them using dynamic SQL.

## Access path determination

Besides the resources consumed, the fact that access path is determined at runtime for dynamic SQL presents another challenge. For static SQL, it is possible to obtain information about the access path by using EXPLAIN option at bind time at package or plan level. This information is not readily available for dynamic SQL. See Section 7.8, “Obtaining access path information for dynamic SQL” on page 137 for details. Analysis of this information can then complement the similar analysis done for static SQL as a means of quality assurance of embedded dynamic SQL.

## Predictive and reactive governing

Since the exact SQL statement that will be executed is not known in advance, how do you prevent a runaway query? Predictive governing is the ability to estimate the resources needed for a statement and take appropriate action based on thresholds (allow processing, warning, stop execution before the statement starts to execute). In addition, execution of dynamic SQL can be terminated if it consumes more resources than a specified limit. This topic is discussed further in Section 11.2, “Predictive governing” on page 170. To prevent runaway dynamic SQL queries from using too much system resources, implement the DB2 governor functionality.

## Programming dependencies

In a complex operational system, with thousands of tables and programs to manage, it is of paramount importance to keep track of where a certain table or column is referred to. This may be needed, for example, to assess the impact of a change to the length of a column. When dynamic SQL is used, it is difficult (sometimes impossible) to know this without any uncertainty. This means a potential outage if the complete impact is not assessed properly.

Another important type of dependency is the use of indexes. Using static SQL exclusively, it is easy to determine whether or not an index is used and how effective it is. With the presence of dynamic SQL, this determination involves collecting and analyzing performance data which still cannot accurately predict the impact on dynamic SQL that is infrequently executed (a batch job run quarterly or annually, for example), unless the data collection spans a long period of time.

## 2.6 Choosing between REOPT(VARS) and dynamic SQL

The BIND option REOPT(VARS) can work in conjunction with dynamic SQL. This is discussed in Section 7.1, “REOPT(VARS) option” on page 132. Here, we discuss its use with static SQL as an *alternative* to using dynamic SQL.

To overcome the potentially sub-optimal access path that can result for static SQL in the presence of non-uniform data or highly correlated data, the REOPT(VARS) bind option can be used. This makes static SQL behave like dynamic SQL since the access path is selected at runtime *after* the values of the host variables are known. If the sole reason why dynamic SQL is being considered is to overcome the performance issues arising from non-uniform distribution of data or column correlation affecting static SQL, you should consider the use of REOPT(VARS) with static SQL first.

Also note that dynamic SQL can be written at the statement level. The REOPT(VARS) bind option generally applies to all statements within a package (unless optimization hints are used).

## 2.7 Choosing between dynamic SQL and stored procedures

One of the reasons why people are often reluctant to use dynamic SQL is because of the way authorization works. By default, every user needs to be granted access to all objects they access via dynamic SQL. Administering these authorizations can be time consuming and once users have access to the objects, they can access the objects “outside” the application as well.

In some cases, using stored procedures can be an alternative. If the applications needs the use of dynamic SQL, for example, because you cannot code all potential combinations using static SQL, you can get around some of the security issues by using a stored procedure to encapsulate the logic that requires the use of dynamic SQL. When using a stored procedure, with the correct setup, you only have to authorize users to call the stored procedure and no longer grant them access on the individual objects. This is discussed in more detail in 13.3, “Using stored procedures to overcome security issues” on page 223.

## 2.8 Importance of catalog statistics for dynamic SQL

An explanation of how catalog statistics influence the access of an SQL statement is provided in Chapter 32 of *DB2 Universal Database for OS/390 and z/OS Administration Guide*, SC26-9931. This applies to static as well as dynamic SQL. Here we briefly point out why they are especially important for the performance of dynamic SQL.

There are a few important considerations:

- ▶ **Timeliness:** In general, the access paths for static SQL are not impacted until the next bind or rebind takes place. For dynamic SQL, the impact on access paths is immediate. When addressing a performance issue, this means that the result is visible immediately.
- ▶ **Outage:** As a direct by-product of the above, the timing of RUNSTATS becomes critical. With only static SQL, the impact of gathering new statistics using RUNSTATS is not felt until a scheduled bind or rebind occurs - with dynamic SQL this will be immediate. For example, a production environment may use a scenario like the one shown below:
  - RUNSTATS
  - Analyze candidates for REORG
  - Periodic REORG, RUNSTATS and REBIND

While this scenario is harmless in an environment with no dynamic SQL, execution of RUNSTATS can trigger a change in the access path and cause a degradation of performance in an environment containing dynamic SQL.

- ▶ **Catalog statistics:** If dynamic SQL is used to overcome the performance issues arising from the presence of host variables, it is important to gather accurate catalog statistics including column distribution and column correlation information. This is discussed in detail in *DB2 Universal Database for OS/390 and z/OS Utility Guide and Reference*, SC26-9945.

On the other hand, it is not just the statistics themselves that should be considered important. During the prepare process, DB2 has to retrieve these statistics from the DB2 catalog. Therefore the performance of queries accessing catalog tables is equally important. Make sure your catalog is in good shape when you use dynamic SQL by executing REORGs of catalog table spaces when appropriate.

## 2.9 Examples of interactive dynamic SQL programs

DB2 comes with a set of programs to process dynamic SQL which are used as a productivity aid. Appendix C of *DB2 Universal Database for OS/390 and z/OS Application Programming and SQL Guide*, SC26-9933 contains an explanation of the parameters they use and sample JCL used to run them. Table 2-1 below summarizes how they can be used.

Table 2-1 Examples of dynamic SQL programs

Program name	Language	Purpose and types of statements processed	Shipped as
DSNTEP2	PL/I	All statements (SELECT and non-SELECT)	Source and object module
DSNTIAD	Assembler	Processes all non-SELECT statements	Source
DSNTIAUL	Assembler	Unload data. Also processes all non-SELECT statements (1)	Source
SPUFI	PL/X	All statements (SELECT and non-SELECT)	Load modules

(1) Note that you can also use the REORG UNLOAD EXTERNAL and UNLOAD (Version 7) utility to unload data. Those usually provide much better performance than the sample DSNTIAUL program.

## 2.10 General recommendations on the use of dynamic SQL

These are some recommendations on the use of dynamic SQL:

- ▶ Static SQL should generally be the first choice.
- ▶ Before exploiting dynamic SQL to overcome performance issues arising from the presence of host variables and non-uniform distribution of data, consider using the REOPT(VARS) bind option. However, be aware of the fact that this bind option is specified at the package or plan level. If a package contains several statements that do not need to be re-optimized, you incur the extra cost unnecessarily. It is therefore usually a good idea to isolate statements that require re-optimization at runtime in a separate package (which is extra work). Dynamic SQL can be used at the statement level.
- ▶ Fixed list selects add some degree of coding complexity. For varying-list selects, the added complexity is considerable (see Chapter 3, “Developing embedded dynamic SQL applications” on page 19). When starting out with dynamic SQL, start with the less complex forms and gradually move to the difficult variations.
- ▶ When used, dynamic SQL must be managed and controlled properly (see Chapter 12, “Monitoring, tracing, and explaining dynamic SQL” on page 179).
- ▶ Consider the cost of preparing the statement in relation to the total cost. For statements that are infrequently executed and access large amount of data, the prepare cost is generally negligible, thus favoring dynamic SQL.
- ▶ When the number of combinations of static SQL required is prohibitively large, consider using dynamic SQL.
- ▶ When dynamic statement caching and parameter markers are used, the cost of repeated prepares can be reduced significantly.
- ▶ Dynamic SQL can cause a security exposure (see Chapter 13, “Security aspects of dynamic SQL” on page 221 for details) and if authorizations need to be maintained for each user, this can lead to additional work in administering security.



## Developing embedded dynamic SQL applications

In this chapter we focus on developing applications containing embedded dynamic SQL in “traditional” programming languages such as COBOL, C, and PL/I.

We discuss these topics:

- ▶ The basics of coding embedded dynamic SQL
- ▶ What parameter markers are, and how to use them
- ▶ What the SQLDA is, and how and when it must be used
- ▶ The different variations of embedded dynamic SQL
- ▶ Special techniques and considerations when dealing with embedded dynamic SQL

## 3.1 Introduction

One of the most common misconceptions about embedded dynamic SQL is its complexity. Admittedly, while the most general forms are indeed complex, several forms that are generally useful in practice are easy to master and are generally under-utilized due to the myths and paranoia surrounding *any* use of dynamic SQL.

This chapter provides templates for the six most common forms (see section 3.6, “Variations of embedded dynamic SQL” on page 24) along with COBOL examples for each step.

The following sections cover the basics and should be sufficient for most users developing applications using embedded dynamic SQL:

- ▶ 3.2, “Restrictions of dynamic SQL statements” on page 20
- ▶ 3.3, “The basics of using dynamic SQL” on page 21
- ▶ 3.4, “What are parameter markers?” on page 22
- ▶ 3.5, “What is the SQLDA?” on page 23
- ▶ 3.6, “Variations of embedded dynamic SQL” on page 24
- ▶ 3.7, “Embedded dynamic SQL programming samples” on page 25

The following sections are aimed at the more advanced user interested in developing general applications and utility programs:

- ▶ 3.8, “Using column labels” on page 42
- ▶ 3.9, “Single, double, and triple SQLDA” on page 43
- ▶ 3.10, “Special considerations for LOBs and distinct types” on page 43
- ▶ 3.12, “Handling unknown number and type of parameter markers” on page 45
- ▶ 3.13, “Handling special attributes” on page 47
- ▶ 3.14, “Handling SELECT and non-SELECTs together” on page 48

**Note:** Complete sample programs in COBOL and PL/I can be downloaded from the Web as additional material. Download instructions can be found in Appendix B, “Additional material” on page 251.

Most of the examples in this chapter are provided for COBOL for two reasons. First, it is perhaps the most commonly used host language for embedded SQL. Second, it is also the most difficult language for dealing with dynamic SQL, because of the difficulty of managing pointer-based navigation and a lack of dynamic storage allocation facilities.

## 3.2 Restrictions of dynamic SQL statements

Table 3-1 contains a list of the most common SQL statements that cannot be prepared and therefore cannot be executed dynamically. See appendix B of *DB2 Universal Database for OS/390 and z/OS Application Programming and SQL Guide*, SC26-9933, for a complete list and explanations.

Table 3-1 What dynamic SQL cannot do

Purpose	Statement(s)	Description
Stored procedure invocation	CALL	This pertains to the ability invoke a stored procedure. While the CALL statement itself cannot be prepared dynamically, a static CALL statement can exist that uses a host variable name for the procedure. For example, a COBOL program may issue: EXEC SQL CALL :ws-proc END-EXEC. This makes the CALL essentially dynamic, without the statement containing dynamic SQL.
Cursor manipulation	OPEN FETCH CLOSE	Cursors processing dynamic SQL statements must be declared, opened, fetched and closed (just like static SQL) but the operations themselves (OPEN, FETCH, CLOSE) cannot be dynamically prepared.
Remote connection	CONNECT RELEASE SET CONNECTION	Not permitted via dynamic SQL.
Handling dynamic SQL	PREPARE DESCRIBE EXECUTE	While these will be found in programs with embedded dynamic SQL, the statements themselves cannot be prepared dynamically.
Singleton SELECT	SELECT INTO	It is important to realize the implications of this limitation. A SELECT statement that expects to return only one row (either due to the presence of unique indexes or due to the nature of the predicates) does not require a CURSOR to process when using static SQL. In DB2 V7 you can use the FETCH FIRST 1 ROW ONLY clause to obtain a similar result (which can be dynamically prepared)
Change collection ID	SET CURRENT PACKAGESET	Cannot change the collection ID since this has implications for authorizations used.
Other	SET :host-variable =	Not allowed.

**Restriction:** When preparing a SELECT statement dynamically, a CURSOR must be used even though a maximum of only one row is expected.

### 3.3 The basics of using dynamic SQL

In this section we introduce some concepts and terminology associated with dynamic SQL. More specific details are covered in the following sections.

In general, a dynamic SQL statement must be *prepared* before it can be *executed*.

For example, to prepare and run the following SQL statement:

```
DELETE FROM    DSN8710.EMP
WHERE         EMPNO = '000100'
```

You need to populate a variable (for example, STMTBUFF) with this string value within the host language first, like this:

```
MOVE 'DELETE FROM DSN8710.EMP WHERE EMPNO = '''000100''' ' TO STMTBUFF.
```

At this point, you can issue the SQL statement:

```
PREPARE      S1
FROM         :STMTBUFF
```

followed by:

```
EXECUTE      S1
```

If the statement is not a SELECT statement and has no parameter markers (see 3.4, “What are parameter markers?” for more details on parameter markers) — that is, if it is an INSERT, UPDATE or DELETE statement containing literals or special registers only — it is also possible to accomplish this in one step via an EXECUTE IMMEDIATE statement.

For the statement above, this would look something like the following:

```
EXECUTE IMMEDIATE :STMTBUFF
```

This achieves the same result as the two-step PREPARE followed by the EXECUTE, but is a more convenient form. In general, EXECUTE IMMEDIATE is recommended for statements that will be executed only once in an application.

Note that EXECUTE IMMEDIATE only applies to non-SELECT statements using no parameter markers.

### 3.4 What are parameter markers?

Dynamic SQL statements cannot contain host variables. The only permissible method to obtain the same effect is to use parameter markers (designated with a “?”) as part of the text (*statement string*) that is prepared. When executing the prepared statement, the corresponding host variable is associated with the parameter marker.

There are two types of parameter markers — *typed* and *untyped*. A parameter marker included in a CAST function is typed, without it, it is untyped. Using a typed parameter marker is more efficient and is highly recommended. The following example illustrates how a parameter marker can be used.

To prepare this statement:

```
DELETE FROM    DSN8710.EMP
WHERE          EMPNO =:EMPNO
```

You need to prepare a string like this (typed):

```
DELETE FROM    DSN8710.EMP
WHERE          EMPNO = CAST(? AS CHAR(6))
```

Or like this (untyped):

```
DELETE FROM    DSN8710.EMP
WHERE          EMPNO =?
```

The host variable is associated when executing the statement as follows:

```
EXECUTE      S1
USING        :EMP
```

## 3.5 What is the SQLDA?

The SQLDA is a host language structure that describes input variables, output variables, or the columns of a result table.

A “*receiving-SQLDA*” is needed for a program that contains varying-list SELECT statements (see 3.6, “Variations of embedded dynamic SQL” on page 24 for details) since the list and types of columns are not known in advance. The content of this structure is set by DB2 *after* the execution of the SQL statement.

A program may contain more than one SQLDA. In the general case, a “*sending-SQLDA*” is needed for a program containing parameter markers. The content of this structure is INPUT in section 3.12, “Handling unknown number and type of parameter markers” on page 45 for an explanation of how DB2 can help with the population of the SQLDA).

The use of a “*sending-SQLDA*” is optional in most cases, and is required only if the number or type of parameter markers is unknown in advance (see section 3.12, “Handling unknown number and type of parameter markers” on page 45 for details).

The structure of the SQLDA is shown in Figure 3-1.

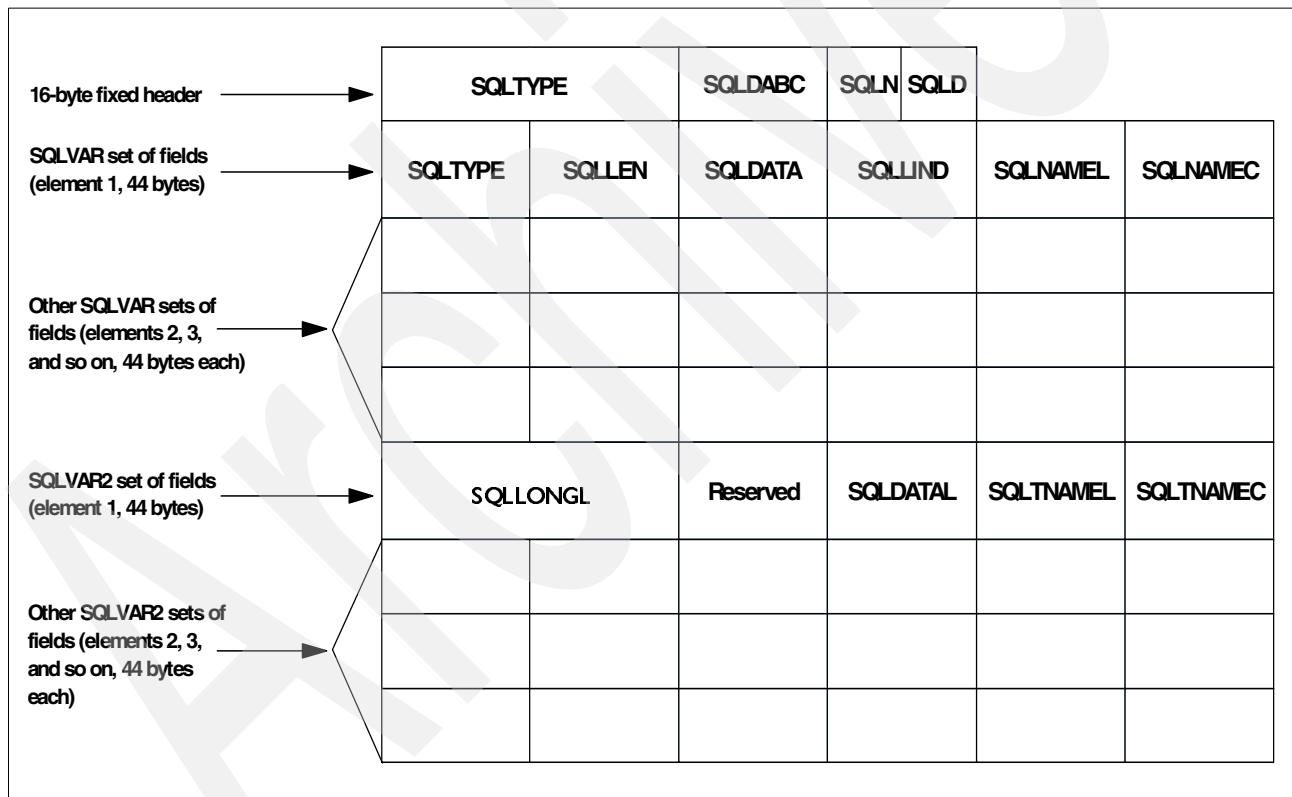


Figure 3-1 The structure of SQLDA

A complete explanation of the contents of the SQLDA structure is available in Appendix C of *DB2 Universal Database for OS/390 and z/OS SQL Reference*, SC26-9944. It consists of a header (SQLDAID), three variables (SQLDABC, SQLN, and SQLD), and an arbitrary number of occurrences of variables, collectively called SQLVAR. A brief overview follows.

<b>SQLDAID</b>	An 8-byte text field containing the text “SQLDA” as an eye-catcher. It is also used to determine if a single, double, or triple SQLDA is needed (see Section 3.9, “Single, double, and triple SQLDA” on page 43 for details).
<b>SQLDABC</b>	Contains the length of the SQLDA structure. For n variables, it must be $16 + 44*n$ (also see Section 3.9, “Single, double, and triple SQLDA” on page 43 for conditions where double and triple SQLDA are needed).
<b>SQLN and SQLD</b>	Specifies the number of occurrences of SQLVAR — SQLN determines how much storage is allocated and SQLD determines the actual number used (It can be less than SQLN).
<b>SQLVAR</b>	A repeating structure that contains:
<b>SQLTYPE</b>	Data type of the column or parameter marker (see appendix C of the <i>DB2 Universal Database for OS/390 and z/OS SQL Reference</i> , SC26-9944 for a list of the different SQLTYPES).
<b>SQLLEN</b>	Length of column or parameter marker.
<b>SQLDATA</b>	Pointer to host variable.
<b>SQLIND</b>	Pointer to the indicator variable of the host variable (indicates if value is NULL or not).
<b>SQLNAME</b>	Name or label of the column.

The number of occurrences of SQLVAR depends on the type of SQLDA. When used as a “sending-SQLDA”, it is the number of parameter markers in the SQL statement. When used as a “receiving-SQLDA”, it is the number of columns returned by the SELECT statement.

## 3.6 Variations of embedded dynamic SQL

As suggested by its title, this book focuses on dynamic SQL, especially where dynamic SQL differs from static SQL. We assume that the reader is familiar with static SQL. For this reason, we do not discuss in detail the areas where static SQL and dynamic SQL are coded and used the same way, for example, the usage of the OPEN CURSOR, FETCH INTO, and CLOSE CURSOR statements.

There are three basic variations of dynamic SQL that you need to be aware of: non-SELECT statements, fixed-list SELECT statements, and varying-list SELECT statements. For each basic variation, the presence or absence of parameter markers requires some additional steps, leading to six variations, as illustrated here:

- Non-SELECT statements without parameter markers:

These refer to INSERT, UPDATE or DELETE statements that do not contain any parameter markers. This is the simplest variation. For example:

```
UPDATE      DSN8710.EMP
SET         SALARY = SALARY * 1.10
WHERE      EMPNO = '000100'
```

- Non-SELECT statements with parameter markers:

These refer to INSERT, UPDATE or DELETE statements that contain a parameter marker. This is only slightly more complex. For example:

```
UPDATE      DSN8710.EMP
SET         SALARY = SALARY * 1.10
WHERE      EMPNO = CAST(? AS CHAR(6))
```

- Fixed-list SELECT statements without parameter markers:

These refer to SELECT statements that return a fixed list of columns — *not a fixed number of rows*. For example:

```
SELECT      LNAME, SALARY
FROM        DSN8710.EMP
WHERE       EMPNO = '000100'
```

- Fixed-list SELECT statements with parameter markers:

As before, these refer to SELECT statements that return a fixed list of columns — *not a fixed number of rows*. The only difference is the presence one or more parameter markers. For example:

```
SELECT      LNAME, SALARY
FROM        DSN8710.EMP
WHERE       EMPNO = CAST(? AS CHAR(6))
```

The next two variations require a host language that can dynamically acquire storage. These include assembler, C, PL/I, and versions of COBOL other than OS/VS COBOL.

- Varying-list SELECT statements without parameter markers:

A varying-list SELECT statement returns rows containing an unknown number of columns of unknown type in the SELECT clause. The number and type of host variables needed to store the results is therefore not known in advance. For example:

```
SELECT      LNAME, SALARY
FROM        DSN8710.EMP
WHERE       EMPNO = '000100'
```

The user could just as well have issued a statement containing a different number or set of columns, such as these:

```
SELECT      FNAME, COMMISSION, SALARY, DEPT
FROM        DSN8710.EMP
WHERE       EMPNO = '000100'
```

- Varying-list SELECT statements with parameter markers:

This is similar to the variation above with the additional presence of parameter markers. For example:

```
SELECT      LNAME, SALARY
FROM        DSN8710.EMP
WHERE       EMPNO = CAST(? AS CHAR(6))
```

## 3.7 Embedded dynamic SQL programming samples

We now discuss in detail the specific steps an application must take to handle the six variations outlined above. Implementation details vary with the host language chosen. This section assumes that COBOL or a similar host language is used.

**Note:** Complete sample programs in COBOL, PL/I, REXX, JDBC and ODBC C are available on the Web. Instructions on how to download this sample code can be found in Appendix B, “Additional material” on page 251. We strongly encourage you to download the sample program in a language you are familiar with before reading this section.

### 3.7.1 Non-SELECT statements without parameter markers

Handling non-SELECT statements without parameter markers is the easiest type of statement to process dynamically. Figure 3-2 shows the steps involved.

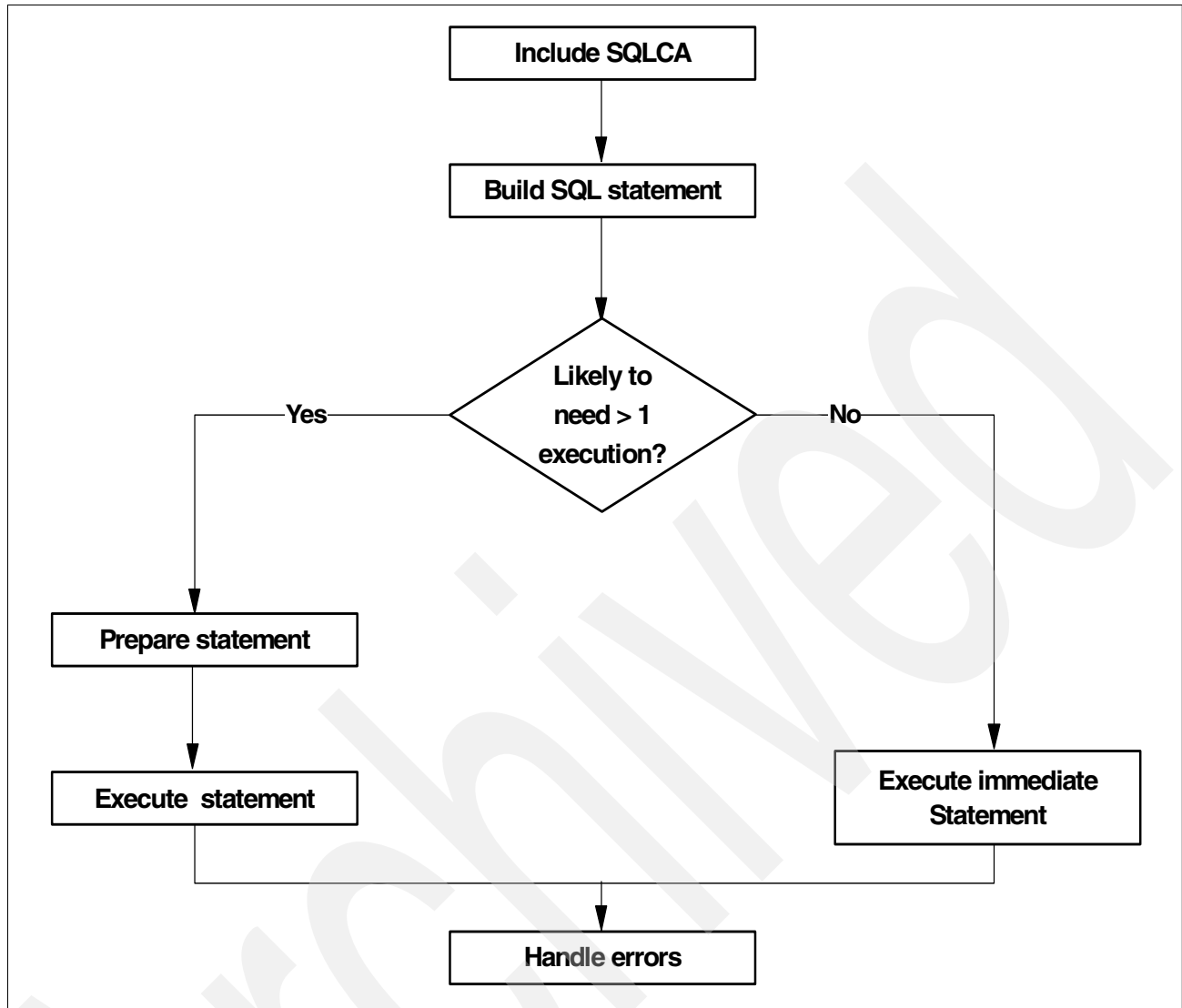


Figure 3-2 Steps for non-SELECT statements without parameter markers

1. Include an SQLCA:

This requirement is the same as for static SQL. The method is dependent on the host language. For example, a COBOL program includes it as follows:

```
EXEC SQL
INCLUDE      SQLCA
END-EXEC.
```

2. Build the SQL statement:

This step populates the contents of a host variable with the full SQL statement to be built and executed. This is dependent on the source of information and the language. For example, the input may be obtained as a full or partial SQL statement from the terminal or a data set. A template statement that is further manipulated by the language may also be used. The end result is a text variable containing the full SQL statement. This string must be a varying-length character variable. For example, in COBOL, it can look something like:

```
01 STMTBUF.
49 STMTLEN          PIC S9(4) COMP VALUE +398.
49 STMTTXT          PIC X(398).
```



It is also recommended that you declare a variable as a statement. For example, a COBOL program can issue:

```
EXEC-SQL
DECLARE      STMT STATEMENT
END-EXEC.
```

**Note:** The DECLARE ... STATEMENT is optional, but we recommend its use since it documents all names used to identify prepared SQL statements.

### 3. Execute the statement:

As discussed in section 3.3, “The basics of using dynamic SQL” on page 21, this is accomplished by one of two methods — either by issuing an EXECUTE IMMEDIATE or by issuing a PREPARE followed by an EXECUTE. A COBOL program can issue either:

```
EXEC SQL
PREPARE      STMT
FROM         :STMTBUFF
END-EXEC
```

followed by:

```
EXEC SQL
EXECUTE      STMT
END-EXEC.
```

or issue:

```
EXEC SQL
EXECUTE IMMEDIATE STMT
FROM             :STMTBUFF
END-EXEC.
```

### 4. Handle any resulting errors:

Any resulting errors are handled by interrogating the contents of the SQLCA. A typical error-handling routine may be called depending on the host language. For example, for COBOL it can look like this:

```
EVALUATE SQLCODE
  WHEN 0
    CONTINUE
  WHEN OTHER
    PERFORM 9000-DBERROR THRU 9000-EXIT
END-EVALUATE.
```

The 9000-DBERROR paragraph should contain standard methods (identical to those used for static SQL) to format the error message via a call to DSNTIAR and externalize the error if the program is unable to continue after the error.

## 3.7.2 Non-SELECT statements with parameter markers

This type of statement is handled in a similar fashion. Since we have parameter markers, we need to associate the statement with a host variable that has been populated before the execution of the statement. Figure 3-3 shows the steps involved.

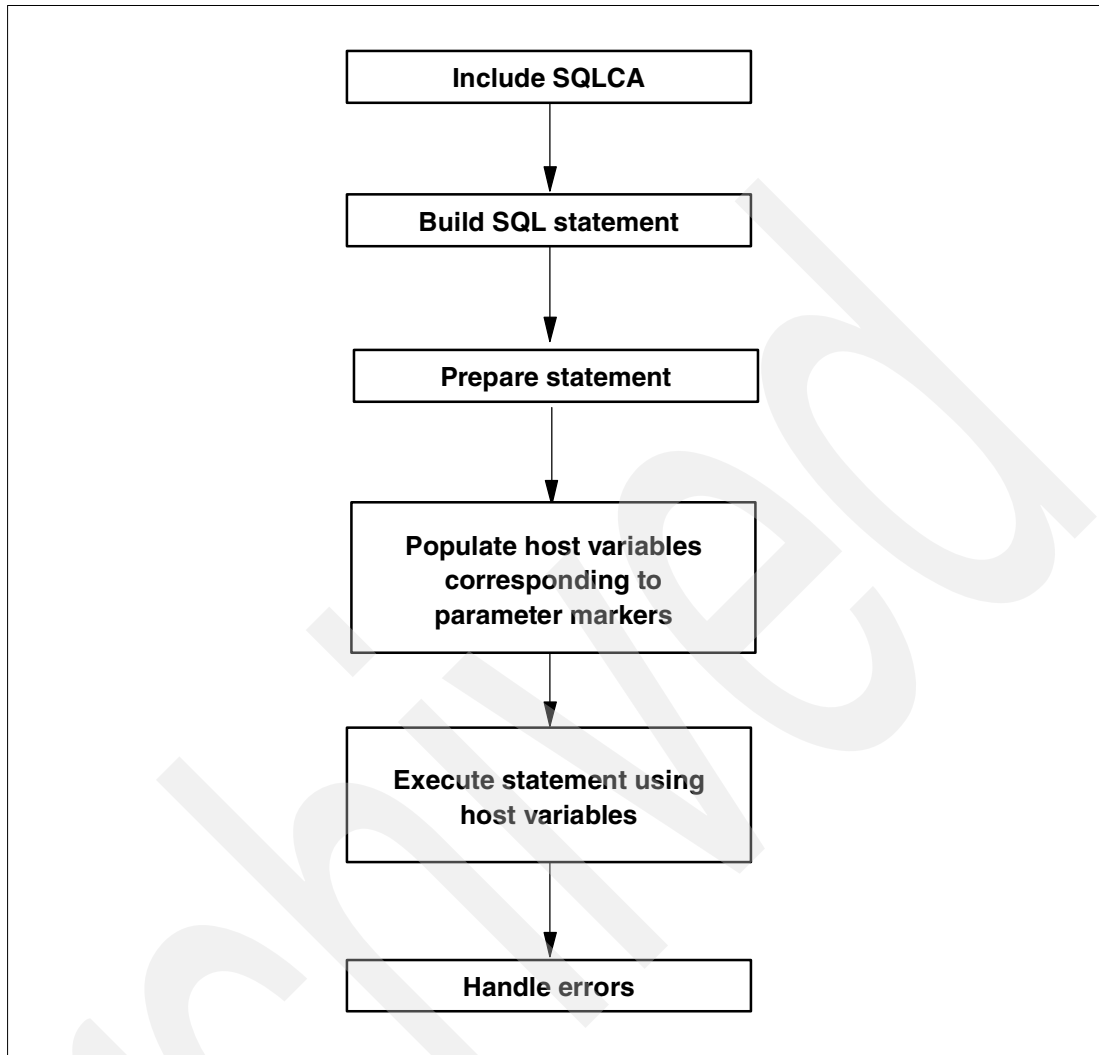


Figure 3-3 Steps for non-SELECT statements with parameter markers

These steps outline the sequence of actions required:

1. Include an SQLCA:

This is required, as discussed before in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25.

2. Build the SQL statement:

This is populated as discussed before in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25. The only difference is that the statement is built with parameter markers. The string may contain, for example:

```

UPDATE      DSN8710.EMP
SET         SALARY = SALARY * 1.10
WHERE       EMPNO = CAST(? AS CHAR(6))
  
```

The COBOL declaration of STMTBUFF can look something like:

```

01 STMTBUF.
   49 STMTLEN          PIC S9(4) COMP VALUE +398.
   49 STMTTXT          PIC X(398).
  
```

It is also recommended that you declare a variable as a statement. For example, a COBOL program can issue:

```
EXEC-SQL  
DECLARE      STMT STATEMENT  
END-EXEC.
```

3. Populate host variables corresponding to parameter markers:

For each parameter marker specified in the statement, a corresponding host variable of the correct type must be populated. For example, for the statement above a string variable must be declared and populated. In COBOL, this is done as follows:

```
10 EMPNO          PIC X(6).  
...  
MOVE '000100' TO EMPNO.
```

4. Execute the statement:

This step is similar to the prepare and execute in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25 except for the need to associate each parameter with a corresponding host variable before execution. This is accomplished with a USING clause as shown here for COBOL:

```
EXEC SQL  
PREPARE      STMT  
FROM         :STMTBUFF  
END-EXEC
```

followed by:

```
EXEC SQL  
EXECUTE      STMT  
USING        :EMPNO  
END-EXEC.
```

When the number of input variables is unknown, a more general (but complex) method that utilizes the SQLDA to link the statement to the host variables via pointers is required. This is discussed in section 3.12, “Handling unknown number and type of parameter markers” on page 45.

Note that because we use parameter markers in this type of statement, EXECUTE IMMEDIATE cannot be used.

Once prepared, if the statement must be executed again within the same logical unit of work (LUW) using different values for the host variables, the statement does not have to be prepared again. Also see Chapter 8, “Dynamic statement caching” on page 141 to assess the impact of statement caching on the ability to avoid the prepare.

5. Handle any resulting errors:

This step is identical to the error-handling discussed in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25.

### 3.7.3 Fixed-list SELECT statements without parameter markers

Figure 3-4 shows the steps involved.

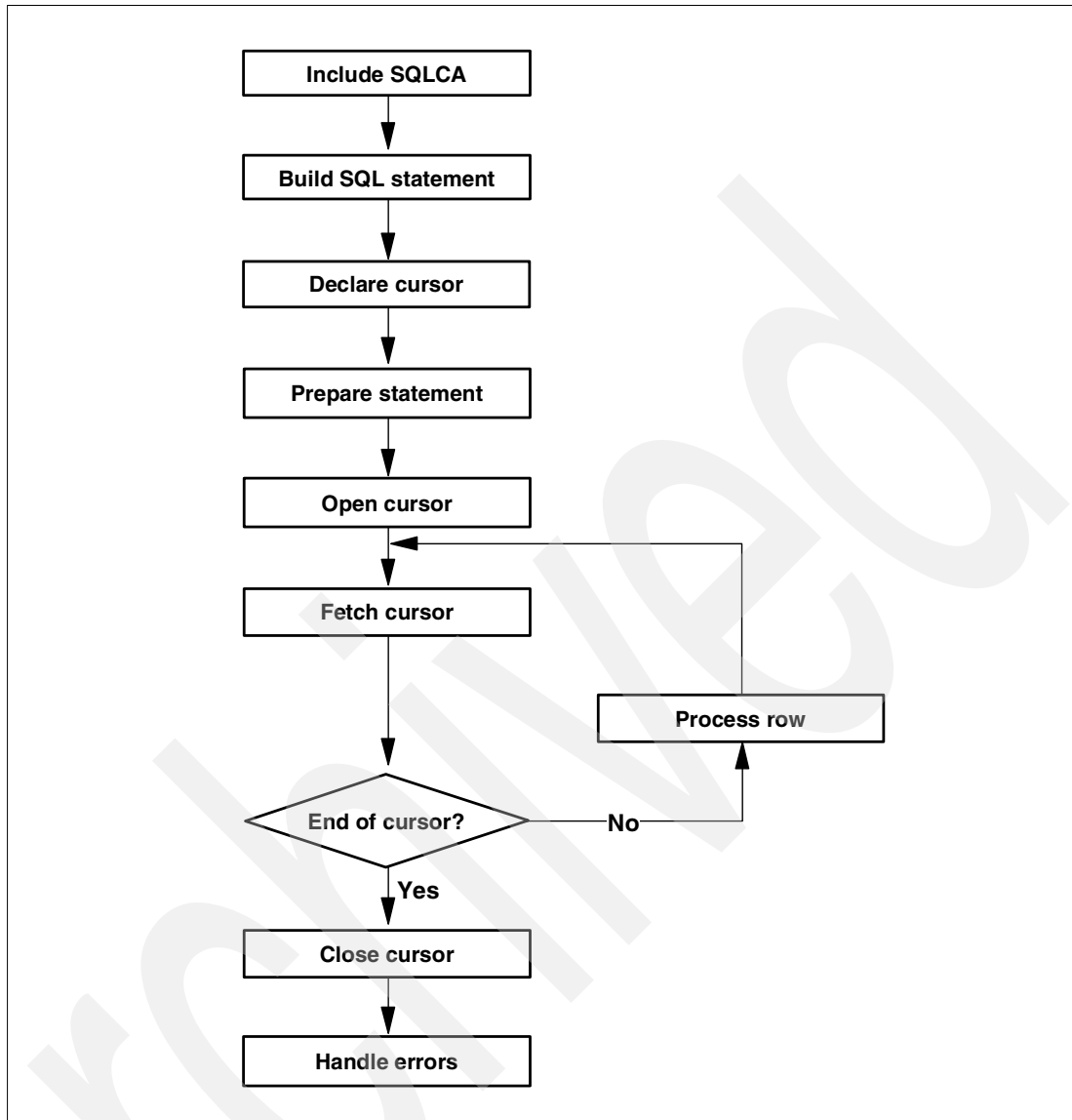


Figure 3-4 Steps for fixed-list *SELECT* statements without parameter markers

1. Include an SQLCA:

This is required, as discussed before in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25.

2. Build the SQL statement:

This is populated as discussed before in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25. The statement may look like this:

```

SELECT      LASTNAME, SALARY
FROM        DSN8710.EMP
WHERE       EMPNO = '000100'
  
```

The COBOL declaration of STMTBUFF can look something like:

```

01 STMTBUF.
   49 STMTLEN          PIC S9(4) COMP VALUE +398.
   49 STMTTXT          PIC X(398).
  
```

It is also recommended that you declare a variable as a statement. For example, a COBOL program can issue:

```
EXEC-SQL  
DECLARE      STMT STATEMENT  
END-EXEC.
```

3. Declare a cursor:

As discussed in section 3.2, “Restrictions of dynamic SQL statements” on page 20, a cursor is required even if the SQL cannot return multiple rows. In a COBOL program this can be done as follows:

```
EXEC SQL  
DECLARE      C1 CURSOR FOR STMT  
END-EXEC.
```

4. Prepare the statement:

As before, in a COBOL program this can be done as follows:

```
EXEC SQL  
PREPARE      STMT  
FROM         :STMTBUFF  
END-EXEC
```

5. Open the cursor:

From this point on, the processing is identical to how a cursor is processed in a static SQL program. In a COBOL program this can be done as follows:

```
EXEC SQL  
OPEN         C1  
END-EXEC.
```

6. Fetch rows from cursor (looping until end of cursor) and process them:

Since this a fixed-list SELECT, the number and type of columns returned is known in advance and the FETCH statement simply obtains their values in compatible host variables, just like with static SQL. In a COBOL program, the statement may look like this:

```
EXEC SQL  
FETCH       C1  
INTO       :FIRSTNME, :LASTNAME, :SALARY  
END-EXEC.
```

The COBOL declaration of the output host variables can look like this:

```
10 FIRSTNME.  
   49 FIRSTNME-LEN      PIC S9(4) USAGE COMP.  
   49 FIRSTNME-TEXT     PIC X(12).  
10 LASTNAME.  
   49 LASTNAME-LEN      PIC S9(4) USAGE COMP.  
   49 LASTNAME-TEXT     PIC X(15).  
10 SALARY  
   49 SALARY            PIC S9(7)V9(2) USAGE COMP-3.
```

7. Close the cursor:

This is the same as for static SQL, this step looks like this in a COBOL program:

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

8. Handle any resulting errors:

This step is identical to the error-handling discussed in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25.

### 3.7.4 Fixed-list SELECT statements with parameter markers

Most of the processing here is identical to that of the previous section, except for step 5 (new step) and step 6 (modified to include the USING clause on the OPEN). All steps are included and shown here for completeness.

Figure 3-5 shows the steps involved.

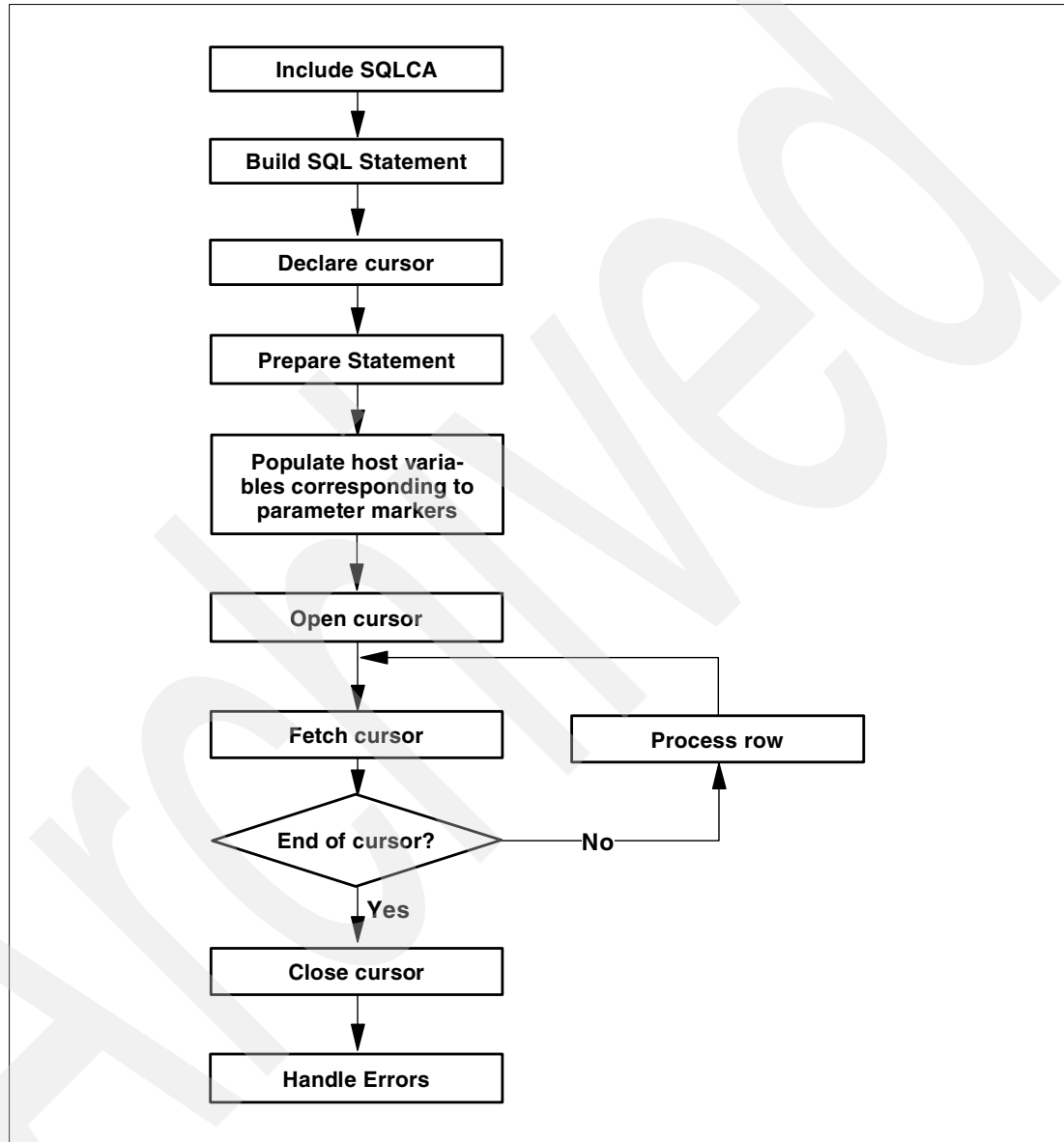


Figure 3-5 Steps for fixed-list SELECT statements with parameter markers

1. Include an SQLCA:

This is the same as the previous section.

2. Build the SQL statement:

This is the same as the previous section. The statement may look like this:

```
SELECT      LASTNAME, FIRSTNME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE ? AND SALARY > ?
```

The COBOL declaration of STMTBUFF can look something like:

```
01 STMTBUF.  
  49 STMTLEN          PIC S9(4) COMP VALUE +398.  
  49 STMTTXT          PIC X(398).
```

It is also recommended that you declare a variable as a statement. For example, a COBOL program can issue:

```
EXEC-SQL  
DECLARE      STMT STATEMENT  
END-EXEC.
```

3. Declare a cursor:

This is the same as the previous section.

4. Prepare the statement:

Identical to the previous section.

5. Populate the host variables corresponding to parameter markers:

For each parameter marker specified in the statement, a corresponding host variable of the correct type must be populated. For example, for the statement above, variables using the appropriate data type are declared and populated. In COBOL, this can be done as follows:

```
10 SALARY              PIC S9(7)V9(2) USAGE COMP-3.  
10 LASTNAME.  
  49 LASTNAME-LEN      PIC S9(4) USAGE COMP.  
  49 LASTNAME-TEXT     PIC X(15).  
...  
MOVE 2 TO LASTNAME-LEN.  
MOVE 'S%' TO LASTNAME-TEXT.  
MOVE 25000 TO SALARY.
```

6. Open the cursor:

A COBOL program that processes the SQL statement from step 2 contains an OPEN statement with a USING clause like this:

```
EXEC SQL  
OPEN          C1  
USING         :LASTNAME, :SALARY  
END-EXEC.
```

7. Fetch rows from cursor (looping until end of cursor) and process them:

This is the same as the previous section.

8. Close the cursor:

This is the same as the previous section.

9. Handle any resulting errors:

This is the same as previous section.

### 3.7.5 Varying-list SELECT statements without parameter markers

Figure 3-6 shows the steps involved.

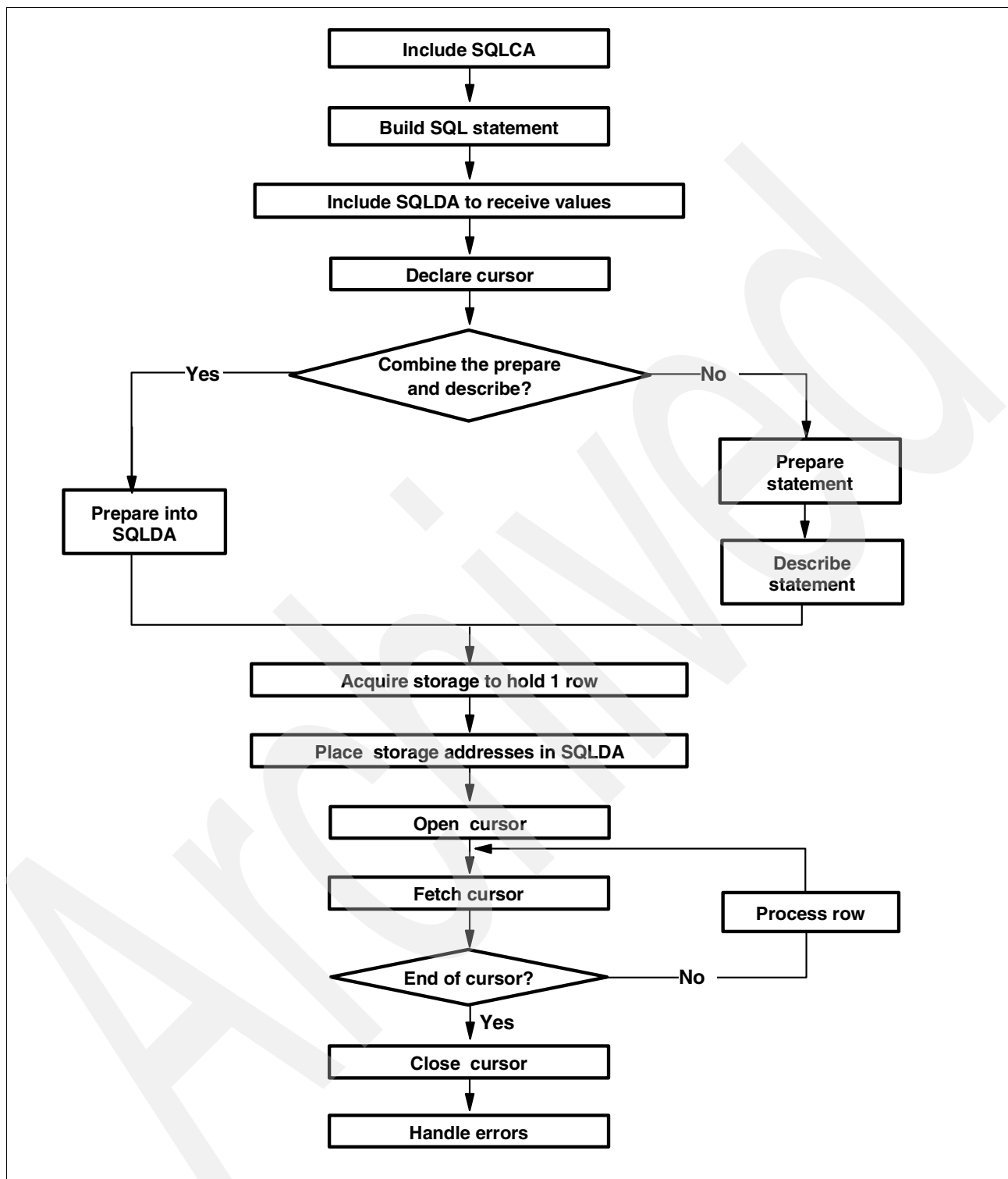


Figure 3-6 Steps for varying-list SELECT statements without parameter markers

1. Include an SQLCA:

This is required, as discussed before in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25.



2. Build the SQL statement:

This is populated as discussed before in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25.

The COBOL declaration of STMTBUFF can look something like:

```
01 STMTBUF.  
   49 STMTLEN          PIC S9(4) COMP VALUE +398.  
   49 STMTTXT          PIC X(398).
```

It is also recommended that you declare a variable as a statement. For example, a COBOL program can issue:

```
EXEC-SQL  
DECLARE      STMT STATEMENT  
END-EXEC.
```

3. Include a “receiving-SQLDA”:

Whether an SQLDA must be explicitly declared as a structure or can be included via the INCLUDE SQLDA command depends on the host language. The INCLUDE SQLDA command is applicable only in COBOL, PL/I, C or assembler programs. For OS/VS COBOL and FORTRAN, this command is not permitted and the structure must be declared using the host language definitions.

A COBOL program includes it as follows:

```
EXEC SQL  
INCLUDE      SQLDA  
END-EXEC.
```

4. Declare a cursor:

As discussed in section 3.2, “Restrictions of dynamic SQL statements” on page 20, a cursor is required even if the SQL cannot return multiple rows. In a COBOL program this can be done as follows:

```
EXEC SQL  
DECLARE      C1 CURSOR FOR STMT  
END-EXEC.
```

5. Prepare the statement:

Prior to the execution of this step, the program must determine the maximum number of columns the SELECT statement can return. A technique described in section 3.11, “Conserving storage — right-sizing the SQLDA” on page 44 allows for the conservation of storage by declaring a minimum SQLDA and later declaring a larger SQLDA if needed. Here, we assume that the statement can return up to 100 columns. The variable SQLN in the declared SQLDA must therefore be set to 100 before the PREPARE is issued. Note that without this, DB2 will *not* populate the SQLDA correctly.

We cannot just prepare the SQL statement, but also have to ask DB2 to populate the contents of SQLDA, to inform the program about the number and type of columns selected. This can be done in one of two ways, either with a PREPARE and a separate DESCRIBE statement, or with a single PREPARE INTO statement.

- If the PREPARE-DESCRIBE option is chosen, you can code it in a COBOL program as follows:

```
EXEC SQL  
PREPARE      STMT  
FROM         :STMTBUFF  
END-EXEC
```

Step 6 must be completed following this to populate the contents in SQLDA.

- Alternatively, the program may use the PREPARE INTO as follows:

```
EXEC SQL
PREPARE      STMT
INTO         :SQLDA
FROM         :STMTBUFF
END-EXEC
```

In this case, step 6 is not required.

6. Obtain information about the data type of each column of the result table:

See the discussion above to determine if this step is required. If required, this is how it can be done in a COBOL program:

```
EXEC SQL
DESCRIBE     STMT
INTO         :SQLDA
END-EXEC
```

At this point, the contents of SQLDA have been populated by DB2. For example, if the SQL SELECT statement contains:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
```

The contents of SQLDA looks like Figure 3-7.

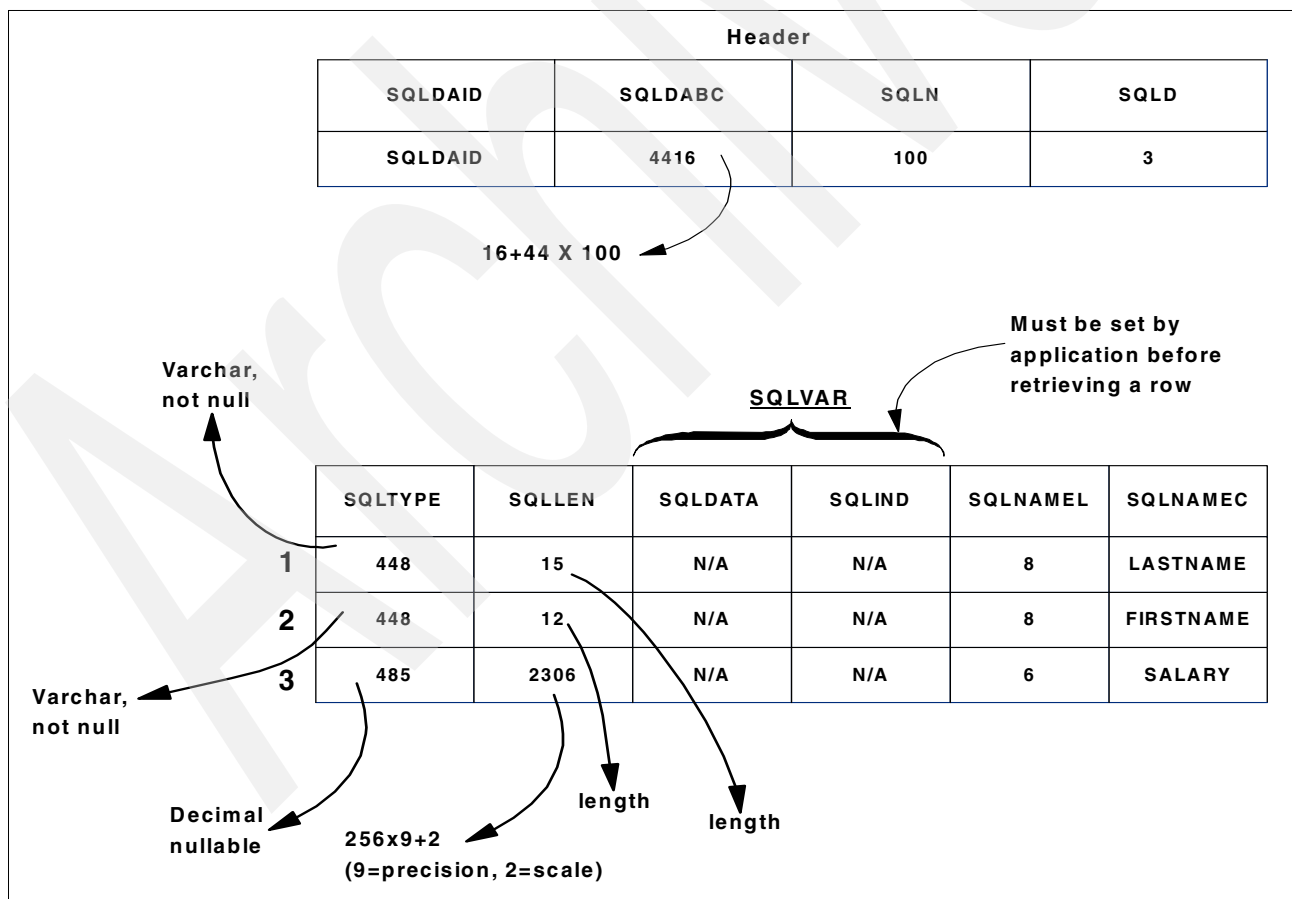


Figure 3-7 SQLDA as populated by PREPARE INTO or DESCRIBE

7. Acquire main storage required to hold a row:

This step varies depending on the host language, following the normal methods of acquiring storage. This can be a static declaration as the following COBOL example shows: (we assume that the SELECT statement can return up to 750 columns and the total length of each row returned does not exceed 1000 bytes)

```
WORKING-STORAGE SECTION.  
...  
01 OUTAREA.  
   02 OUTAREA-LEN          PIC S9(8) COMP VALUE +1000.  
   02 OUTAREA-CHAR         PIC X(1) OCCURS 1000 TIMES.  
...  
01 OUTNULLS.  
   02 OUTIND               PIC S9(4) COMP OCCURS 750 TIMES.
```

The OUTNULLS structure is required to handle the null values (if any) returned by the execution.

8. Place the storage addresses in the SQLDA:

This is the most complex and most important step. Since COBOL does not allow a pointer to address an item in the working-storage section, this step is specially complex since it requires the invocation of a subroutine. We now discuss this in detail. Recall that all fields of the SQLDA have been populated by the earlier PREPARE or DESCRIBE step (step 5 and/or step 6) except for the SQLVAR pointers SQLDATA and SQLIND. This section deals with the procedure to calculate and set these correctly.

For the first output variable, the pointers simply point to the start of the output data area and the start of the area containing the null indicators. To determine the starting position for each of the other variables (2 through n), we use the length of the previous column and add it to the address of the previous variable. The null indicators are assumed to be contiguous in storage. To make life even more complicated, the length of a column is not available directly for decimal, varchar, graphic, and vargraphic column types and has to be calculated.

In COBOL, setting the storage addresses in the SQLDA can be accomplished as follows:

```
CALL 'DSCOBOLS' USING OUTAREA, OUTNULLS, OUTPTR, NULLPTR.
```

The DSCOBOLS subroutine linkage section looks similar to the calling program's working storage:

```
01 OUTAREA.  
   02 OUTAREA-LEN          PIC S9(8) COMP.  
   02 OUTAREA-CHAR         PIC X(1) OCCURS 1000 TIMES  
                           DEPENDING ON OUTAREA-LEN.  
01 OUTNULLS.  
   02 OUTIND               PIC S9(4) COMP OCCURS 750 TIMES.
```

The declaration of the pointers looks like this in COBOL. Notice the need to redefine them to allow us to add lengths to them.

```
01 OUTPTR                  POINTER.  
01 OUTPTR-NUM REDEFINES OUTPTR PIC S9(9) COMP.  
  
01 NULLPTR                 POINTER.  
01 NULLPTR-NUM REDEFINES NULLPTR PIC S9(9) COMP.
```

The processing within the subroutine is quite simple:

```
SET OUTPTR TO ADDRESS OF OUTAREA-CHAR(1).  
SET NULLPTR TO ADDRESS OF OUTIND(1).
```

The mainline program now needs to calculate the offsets, add them to the displacement and set the addresses in SQLDA as follows:

```
PERFORM 2530-SET-EACH-COL
THRU    2530-EXIT
VARYING VARNUM FROM 1 BY 1
UNTIL   VARNUM > SQLD.
```

The processing within the 2530-SET-EACH-COL paragraph should be something like this:

```
SET SQLDATA(VARNUM) TO OUTPTR.
SET SQLIND(VARNUM) TO NULLPTR.
MOVE SQLLEN(VARNUM) TO COLUMN-LEN.
DIVIDE SQLTYPE(VARNUM) BY 2 GIVING DUMMY
REMAINDER COLUMN-IND.
MOVE SQLTYPE(VARNUM) TO COLTYPE.
SUBTRACT COLUMN-IND FROM COLTYPE.
...
```

For a complete list of column types, see appendix C of the *DB2 Universal Database for OS/390 and z/OS SQL Reference*, SC26-9944.

```
EVALUATE COLTYPE
  WHEN CHARTYPE CONTINUE
  WHEN DATETYP  CONTINUE
  WHEN TIMETYP  CONTINUE
  WHEN Timestmp CONTINUE
  WHEN FLOATYPE CONTINUE
  WHEN VARCTYPE
    ADD 2 TO COLUMN-LEN
  WHEN VARLTYPE
    ADD 2 TO COLUMN-LEN
  WHEN GTYPE
    MULTIPLY COLUMN-LEN BY 2 GIVING COLUMN-LEN
  WHEN VARGTYPE
    PERFORM 2531-VARG-LENGTH
    THRU    2531-EXIT
  WHEN LVARGTYP
    PERFORM 2531-VARG-LENGTH
    THRU    2531-EXIT
  WHEN HWTYPE
    MOVE 2 TO COLUMN-LEN
  WHEN INTTYPE
    MOVE 4 TO COLUMN-LEN
  WHEN DECTYPE
    PERFORM 2532-DEC-LENGTH
    THRU    2532-EXIT
  WHEN OTHER
    PERFORM 9000-DBERROR
    THRU    9000-EXIT
END-EVALUATE.
```

```
ADD COLUMN-LEN TO OUTPTR-NUM.
ADD 1 TO NULLPTR-NUM.
```

The VARGRAPHIC data type is handled as follows:

```
MULTIPLY COLUMN-LEN BY 2 GIVING COLUMN-LEN.
ADD 2 TO COLUMN-LEN.
```

The decimal data type requires special processing. The length column consists of 2 bytes. The first byte contains the scale and the second contains the precision. We need to extract this information and calculate the bytes required as follows:

```
DIVIDE COLUMN-LEN BY 256 GIVING COLUMN-PREC
REMAINDER COLUMN-SCALE.
MOVE COLUMN-PREC TO COLUMN-LEN.
ADD 1 TO COLUMN-LEN.
DIVIDE COLUMN-LEN BY 2 GIVING COLUMN-LEN.
```

#### 9. Open the cursor:

From this point on, the processing is identical to how a cursor is processed in a static SQL program. In a COBOL program this looks like this:

```
EXEC SQL
OPEN                C1
END-EXEC.
```

#### 10. Fetch rows from the cursor (looping until end of cursor) and process them:

Unlike previous cases, the list of host variables where the values need to be fetched is known only via the SQLDA. For this reason, a different form of the FETCH statement must be used. It has to specify the USING DESCRIPTOR clause. In a COBOL program this can look like this:

```
EXEC SQL
FETCH                C1
USING DESCRIPTOR    :SQLDA
END-EXEC.
```

The contents of the SQLDA allows us to process the returned row. Since we have placed the addresses of each host variable where we want the data placed as a result of the fetch, the host variable will be populated with the data. For example, if we selected two columns of length 10 and 20 respectively, the data for the first column will start at location 1 and data for the second column will start at location 11 since SQLDATA(1) is 1 and SQLDATA(2) is 11.

#### 11. Close the cursor

The process is the same as for static SQL and looks like this in a COBOL program:

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

#### 12. Handle any resulting errors

This step is identical to the error-handling discussed in section 3.7.1, “Non-SELECT statements without parameter markers” on page 25.

### 3.7.6 Varying-list SELECT statements with parameter markers

This variation required the most complex type of processing. Figure 3-8 shows the steps involved.

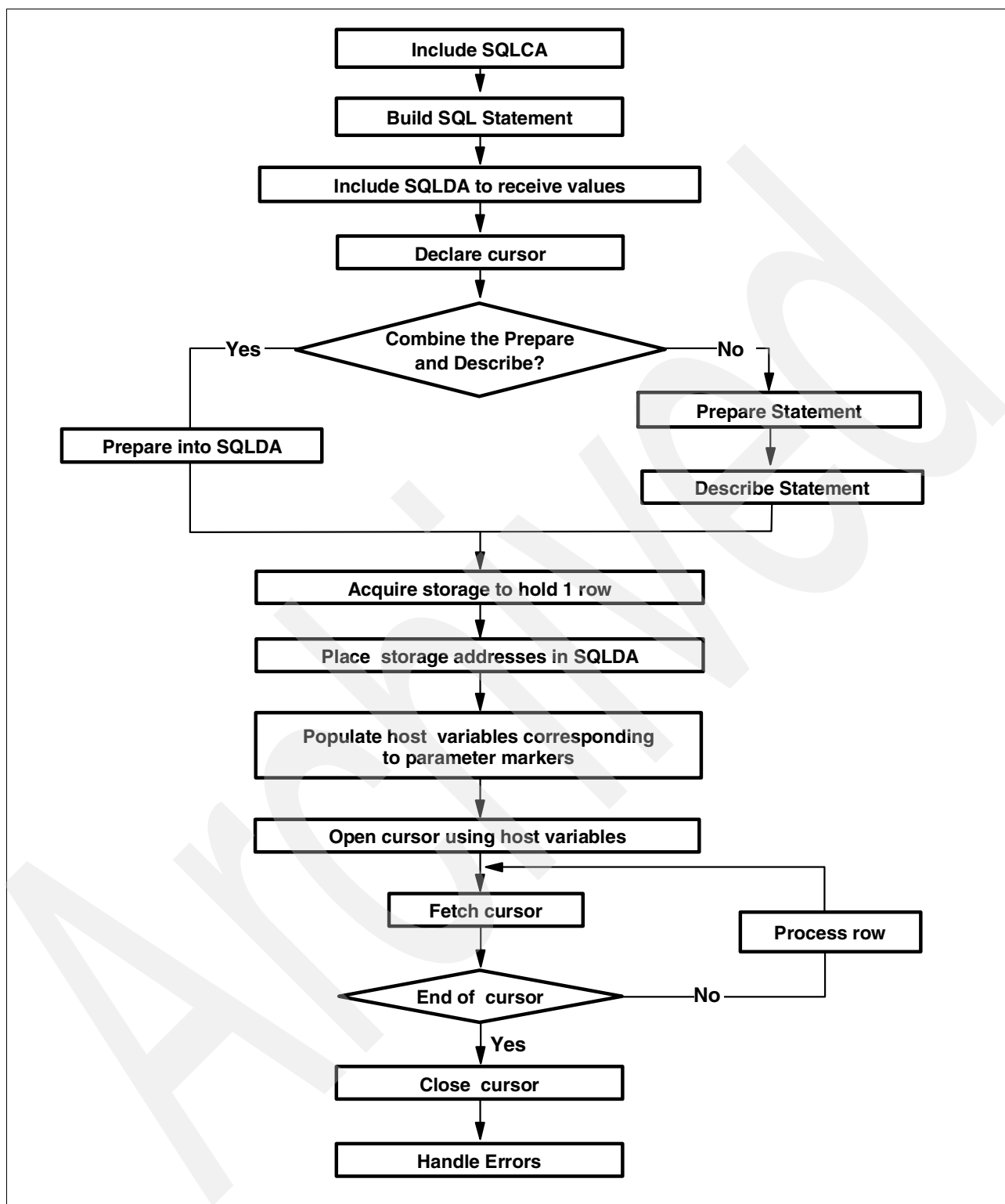


Figure 3-8 Steps for varying-list SELECT statements with parameter markers

1. Include an SQLCA:

This is required, as discussed in the previous section.

2. Build the SQL statement:

This is populated as discussed in the previous section.

The COBOL declaration of STMTBUFF can look something like:

```
01 STMTBUF.  
   49 STMTLEN          PIC S9(4) COMP VALUE +398.  
   49 STMTTXT          PIC X(398).
```

It is also recommended that you declare a variable as a statement. For example, a COBOL program can issue:

```
EXEC-SQL  
DECLARE      STMT STATEMENT  
END-EXEC.
```

3. Include a “receiving-SQLDA”:

This is the same as the previous section. Since the list of values returned by DB2 is unknown in advance a “receiving-SQLDA” is required to process the statement. In addition, a program can contain several SQLDAs and, in general, handling an unknown number or type of parameter markers requires an additional SQLDA to accommodate the parameters (see section 3.12, “Handling unknown number and type of parameter markers” on page 45 for details).

4. Declare a cursor:

This is the same as in the previous section.

5. Prepare the statement:

Identical to the previous section.

6. Obtain information about the data type of each column of the result table:

This is the same as in the previous section.

7. Acquire main storage required to hold a row:

This is the same as in the previous section.

8. Place the storage addresses in SQLDA

This is the same as in the previous section.

9. Populate host variables corresponding to parameter markers

For each parameter marker specified in the statement, a corresponding host variable of the correct type must be populated. For example, for the statement like this:

```
SELECT      LASTNAME, FIRSTNAME, SALARY  
FROM        DSN8710.EMP  
WHERE       LASTNAME LIKE ? AND SALARY > ?
```

Variables that correspond to each parameter marker are declared and populated. In COBOL, this can be done as follows:

```
10 SALARY          PIC S9(7)V9(2) USAGE COMP-3.  
10 LASTNAME.  
   49 LASTNAME-LEN  PIC S9(4) USAGE COMP.  
   49 LASTNAME-TEXT PIC X(15).  
...  
MOVE 2 TO LASTNAME-LEN.  
MOVE 'S%' TO LASTNAME-TEXT.  
MOVE 25000 TO SALARY.
```

#### 10. Open the cursor:

Similar to the previous section, but a USING clause must point to the host variables that correspond to the parameter markers in the correct sequence. A COBOL program that processes the SQL statement from the step 9 has to contain an OPEN statement like this:

```
EXEC SQL
OPEN          C1
  USING       :LASTNAME, :SALARY
END-EXEC.
```

#### 11. Fetch rows from the cursor (looping until end of cursor) and process them

This is the same as previous section.

#### 12. Close the cursor

This is the same as previous section.

#### 13. Handle any resulting errors

This is the same as previous section.

## 3.8 Using column labels

By default, the DESCRIBE statement and PREPARE INTO statement populate the name of each SELECTed column in the SQLNAME field of the SQLDA. Various choices exist with respect to usage of labels instead of or in addition to the column names:

##### ► Using labels instead of column names

This is accomplished by using the USING LABELS clause. For example:

```
DESCRIBE      STMT
INTO          :FULLSQLDA
USING         LABELS
```

If the label is blank, SQLNAME will also contain blanks.

##### ► Using labels, but use column names if no labels are present

This is accomplished by using the USING ANY clause. For example:

```
DESCRIBE      STMT
INTO          :FULLSQLDA
USING         ANY
```

##### ► Using both

This is accomplished by using the USING BOTH clause. For example:

```
DESCRIBE      STMT
INTO          :FULLSQLDA
USING         BOTH
```

In this case, you must allocate a longer SQLDA to contain a second set of SQLVARs with a total length of  $16 + \text{SQLD} * 88$  bytes instead of  $16 + \text{SQLD} * 44$  bytes and the number of columns possible (SQLN) must be set to  $\text{SQLD} * 2$  instead of SQLD. Failure to set any of these correctly can result in an SQLDA that is not sufficiently large. In this case, DESCRIBE does not enter descriptions for *any* of the columns.



## 3.9 Single, double, and triple SQLDA

The number of occurrences of SQLVAR depends on a combination of how many columns you DESCRIBE, whether or not BOTH option is used for column names (see 3.8, “Using column labels”) and whether or not any of the columns described are LOB columns or distinct types. Table 3-2 shows the minimum number of SQLVARs needed.

Table 3-2 How many SQLVARs do I need?

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	$2*n$	$3*n$
No distinct types but LOBs	$2*n$	$2*n$
LOBs and distinct types	$2*n$	$3*n$

An SQLDA containing  $n$  occurrences is called a single SQLDA, one containing  $2*n$  occurrences is called a double SQLDA and one containing  $3*n$  is a triple SQLDA.

## 3.10 Special considerations for LOBs and distinct types

When a LOB column or a column based on a distinct type is selected, you will need a double or triple SQLDA as discussed in the previous section. What exactly is contained in the base SQLDA (first set), extended SQLDA (second set) and the second extended SQLDA (third set)?

Table 3-3 shows the contents of the SQLDA when USING BOTH is not specified on the DESCRIBE statement.

Table 3-3 Contents of SQLDA when USING BOTH is not specified

LOBs	Distinct types	7th byte of SQLDAID	SQLD	Min for SQLN	SQLVAR 1st set	SQLVAR 2nd set	SQLVAR 3rd set
No	No	Blank	$n$	$n$	Column names or labels	Not used	Not used
Nos	Yes	2	$n$	$2n$	Column names or labels	LOBs, distinct types or both	Not used
Yes	No	2	$n$	$2n$	Column names or labels	LOBs, distinct types or both	Not used
Yes	Yes	2	$n$	$2n$	Column names or labels	LOBs, distinct types or both	Not used

Table 3-4 shows the contents of SQLDA when USING BOTH is specified on the DESCRIBE statement.

Table 3-4 Contents of SQLDA when USING BOTH is specified

LOBs	Distinct types	7th byte of SQLDAID	SQLD	Min for SQLN	SQLVAR1st set	SQLVAR 2nd set	SQLVAR 3rd set
No	No	Blank	2n	2n	Column names	Labels	Not used
Nos	Yes	3	n	3n	Column names	Distinct types	Labels
Yes	No	2	n	2n	Column names	LOBs and labels	Not used
Yes	Yes	3	n	3n	Column names	LOBs and distinct types	Labels

A program must be aware of the options used and the contents of the SQLDA before it can process the results of the dynamic SQL statement:

- ▶ For distinct types, nothing special needs to happen except that you need a double/triple SQLDA. This is because the *coltype* is already set to the proper values. For example, if a column is based (sourced) on a type that is decimal, *sqltype* in the SQLDA is 485 (not so in *syscolumns*).
- ▶ For LOBs a double/triple SQLDA is needed. In addition, you need to use the *sqlen* as specified in the second *sqldvar*.

### 3.11 Conserving storage — right-sizing the SQLDA

When the number of columns of the result table is unknown in advance (“varying-list SELECT”), we have three choices of how large to declare the SQLDA:

- ▶ Allow for the maximum (750 columns).
- ▶ Allow for a reasonable number (say 40) and make it larger if needed.
- ▶ Allow for zero, find out exactly how many are needed and acquire just what is needed.

We now discuss the merits of each of these options.

A SELECT statement can return up to 750 columns. To accommodate this, a program may declare an SQLDA to contain 750 columns. This occupies 33,016 bytes for a single SQLDA, 66,016 for a double SQLDA and 99,016 for a triple SQLDA (see section 3.9, “Single, double, and triple SQLDA” on page 43 for an explanation). In general, this leads to a lot of wasted space since a typical SELECT returns far fewer than the maximum 750 permissible. However, if the number chosen is too small, DB2 returns no descriptions.

The best technique specifies a reasonable number (say 40) and handles the situation when this is exceeded by acquiring a larger SQLDA if needed. This is more complex to program but conserves storage and would be recommended for the general case.

Since acquiring and releasing storage is not done easily in COBOL, an alternative is to always use the 2-step process shown here.

We start with a declaration of a minimum SQLDA that does not need to contain any occurrences of SQLVAR. A COBOL declaration looks something like this:

```

01 MSQ LDA.
   05 MSQ LDAID          PIC X(8) .
   05 MSQ LDAABC          PIC S9(9) BINARY.
   05 MSQ LN              PIC S9(4) BINARY.
   05 MSQ LD              PIC S9(4) BINARY.
```

The value of SQLN is set to zero. At this point, we issue a PREPARE as shown here for COBOL:

```
EXEC SQL
PREPARE          STMT
INTO             :MSQLDA
FROM             :STMTBUF
END-EXEC.
```

The intention of this PREPARE is simply to determine the number of SQLVAR entries needed. DB2 will not describe any column since the SQLDA is known to be too small (zero occurrences of SQLVAR).

There are two items of interest after the execution of the PREPARE as shown in Table 3-5.

*Table 3-5 Contents of minimum SQLDA*

Item	Value and Description
7th byte of SQLDAID	blank = single SQLDA
	2 = double SQLDA
	3 = triple SQLDA
SQLD	required number of sqlvars

Also note that when any column containing LOB data type or a distinct type is present in the SELECT clause, the PREPARE issues an SQLSTATE 01005 (SQLCODE of +238 or +239) that must be handled.

Following this, an SQLDA with the required number of entries is acquired, SQLN set to the proper value (=SQLD for single, = 2\*SQLD for double and = 3\*SQLD for triple SQLDA) and the statement is processed as shown in 3.7, “Embedded dynamic SQL programming samples” on page 25.

**To summarize:** Techniques to minimize storage come at a price. Since an additional DESCRIBE is needed (when the number of columns returned in the SELECT clause exceeds the preset SQLN value), the cost of the extra SQL statements and a potential trip across the wire must be weighed against the savings of using less storage. Note that no second PREPARE is necessary, a DESCRIBE is sufficient to provide the SQLDA with the necessary information.

Due to the added coding complexity, we recommend that this technique be used only for programs for which:

- ▶ They can process a wide range of columns (for example, from one to several hundred)
- ▶ It is difficult to predict some reasonable upper limit.
- ▶ Storage benefits are worth the cost of incurring the extra prepare occasionally.
- ▶ The added coding complexity can be managed.

## 3.12 Handling unknown number and type of parameter markers

A dynamic SQL statement may contain more than one parameter marker. If the number is known prior to the execution of the statement, the USING clause of the EXECUTE statement specifies a list of host variables or a host structure that matches the number and type of parameters in the correct order. If this is not known in advance, the application program may choose to issue a DESCRIBE INPUT command to let DB2 populate information about the variables in an SQLDA. This is discussed here.

Consider the SQL statement shown here (the procedure is identical for a non-SELECT statement, as well):

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE ? AND SALARY > ?
```

A program processing such a statement contains an OPEN statement like this:

```
EXEC SQL
OPEN        C1
USING       :LASTNAME, :SALARY
END-EXEC.
```

Instead, we issue the following:

```
EXEC SQL
DESCRIBE INPUT STMT
INTO        :PSQLDA
END-EXEC.
```

This allows DB2 to place the number of input variables and their format in a structure defined in the program as PSQLDA. At this point, it contains the information, shown in Figure 3-9.

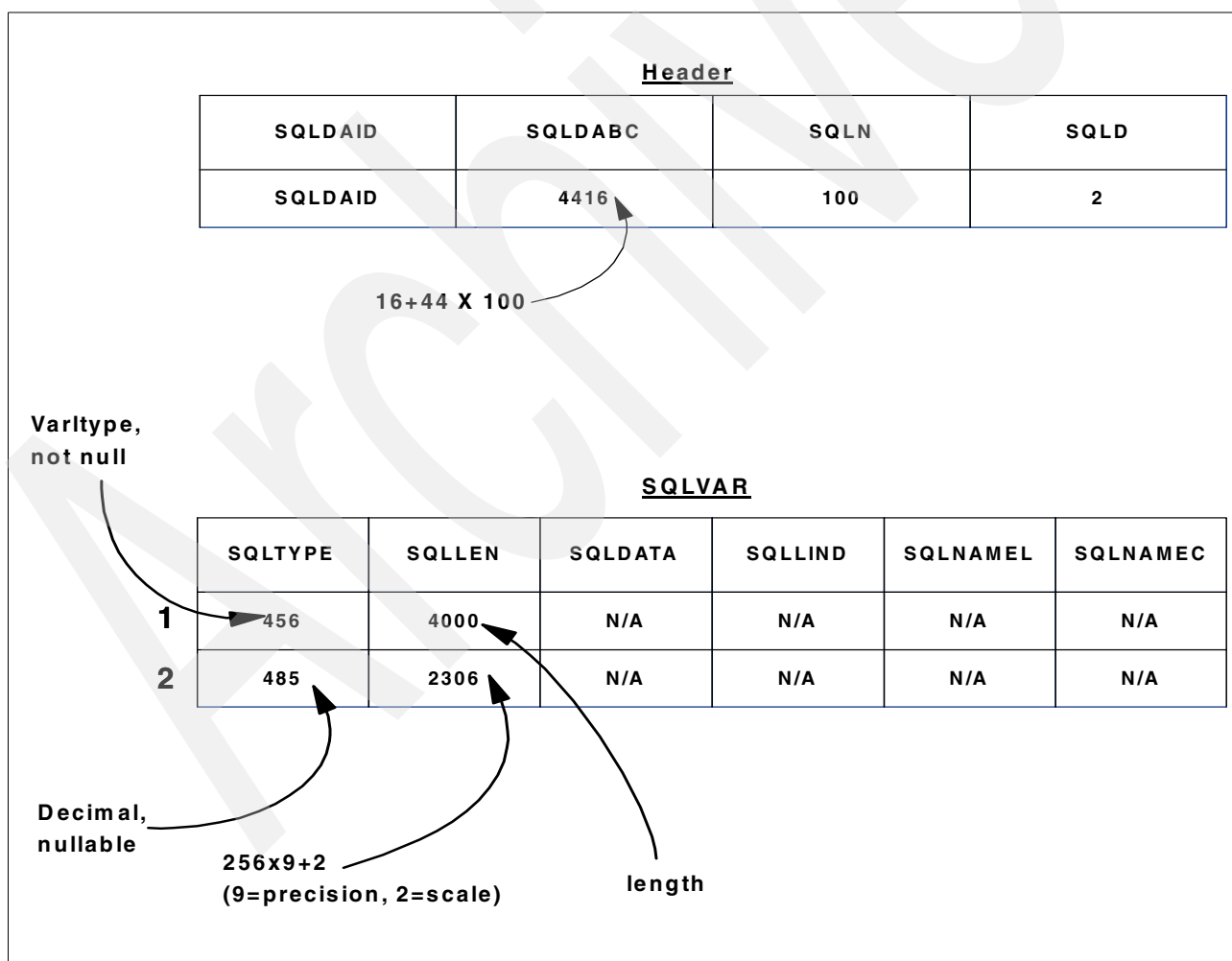


Figure 3-9 SQLDA as populated by DESCRIBE INPUT

Next, we need to set the addresses SQLDATA and SQLIND for each input host variable. In host languages that support pointer manipulation of local storage, this would be easy and accomplished by code similar to:

```
SET PRMPTR = ADDRESS(LASTNAME)
SET NULLPTR = ADDRESS(PRMIND(1))
```

In COBOL, the process is more cumbersome and requires the use of a subroutine, as shown.

We call a subroutine for each input host variable (only LASTNAME is shown here) as follows:

```
MOVE 'LASTNAME' TO INPVARNAME.
CALL 'DSCOBOLP' USING INPVARNAME, EMP, PRMNULLS, PRMPTR, NULLPRMPTR.
```

The subroutine contains a linkage section as shown here:

```
01 INPVARNAME                PIC X(18).
01 EMP.
  10 EMPNO                   PIC X(6).
  10 SALARY                   PIC S9(7)V9(2) USAGE COMP-3.
  10 LASTNAME.
    49 LASTNAME-LEN          PIC S9(4) USAGE COMP.
    49 LASTNAME-TEXT         PIC X(15).

    rest of the variables ....

01 PRMNULLS.
  02 PRMIND                   PIC S9(4) COMP OCCURS 750 TIMES.
01 PRMPTR                     POINTER.
01 NULLPRMPTR                 POINTER.
```

and contains procedural logic such as:

```
PROCEDURE DIVISION USING INPVARNAME, EMP, PRMNULLS, PRMPTR, NULLPRMPTR.
....
EVALUATE INPVARNAME
  WHEN 'LASTNAME'
    SET PRMPTR TO ADDRESS OF LASTNAME
    SET NULLPRMPTR TO ADDRESS OF PRMIND(1)
  WHEN 'SALARY'
    SET PRMPTR TO ADDRESS OF SALARY
    SET NULLPRMPTR TO ADDRESS OF PRMIND(2)
... similar code for every other variable that must be handled ...
...
END-EVALUATE.
```

### 3.13 Handling special attributes

Additional clauses not specified in a statement such as OPTIMIZE FOR 10 ROWS or FETCH FIRST 10 ROWS ONLY can also be included during the PREPARE of a dynamic SQL statement using the ATTRIBUTES clause. For example:

```
PREPARE          STMT
FROM             :STMTBUF
ATTRIBUTES       :ATTRVAR
```

In general, only attributes that were *not* specified in the SQL statement itself can be incorporated during the PREPARE. For example, a SELECT... OPTIMIZE FOR 10 ROWS cannot be overridden during PREPARE by a clause such as OPTIMIZE FOR 1000 ROWS.

A complete list of clauses that can be specified as ATTRIBUTES and their explanation is found in chapter 5 of *DB2 Universal Database for OS/390 and z/OS SQL Reference*, SC26-9944. They are listed here for ease of reference:

- ▶ INSENSITIVE or SENSITIVE STATIC
- ▶ SCROLL
- ▶ WITH HOLD
- ▶ WITH RETURN
- ▶ fetch-first-clause
- ▶ read-only-clause or update-clause
- ▶ optimize-clause
- ▶ isolation-clause

These attributes affect dynamic SQL caching (see Chapter 8, “Dynamic statement caching” on page 141) since they are considered to be part of the SQL statement text. Two SQL statements that vary *only* in their attributes are considered to be *different* for caching purposes.

### 3.14 Handling SELECT and non-SELECTs together

If the program does not know in advance whether or not the statement to be processed is a SELECT, the following technique can be used. This is a by-product of the “minimum-SQLDA” technique discussed in section 3.11, “Conserving storage — right-sizing the SQLDA” on page 44. Following the PREPARE into the minimum SQLDA, we interrogate the SQLD value returned by DB2. For a SELECT statement it is set to the number of columns SELECTed, otherwise it is zero. We can utilize this as the following example shows:

```
PREPARE      STMT
INTO         :MSQLDA
FROM         :STMTBUF;

IF MSQLD = 0
    process the non-SELECT (already prepared),
    supplying host variables for parameter markers if any
ELSE
    process the SELECT
```



## Developing ODBC applications

In this chapter we explain how to develop and control DB2 ODBC applications. We cover the following topics:

- ▶ ODBC, an alternative to embedded SQL
- ▶ ODBC concepts and facilities
- ▶ The ODBC API
- ▶ ODBC programming samples
- ▶ Specific features of DB2 ODBC
- ▶ Performance considerations of ODBC applications

## 4.1 Introduction

DB2 ODBC is a callable interface to DB2, that can be used by C or C++ programs to execute SQL statements against DB2 data. It is based on both the Microsoft Open Database Connectivity (ODBC) specification and the Open Group (formerly called X/Open) Call Level Interface (CLI) specification (almost entirely accepted by the ISO organization).

The ODBC standard has evolved and has been enhanced during the last years. The four specifications (ODBC, DB2 ODBC, and Open Group CLI and ISO CLI) are now evolving together, providing one of the most reliable and successful standards, widely used in client/server applications.

By nature, ODBC applications make use of dynamic SQL. However, their underlying techniques differ fundamentally from those implemented in embedded dynamic SQL. Therefore, we first provide a description of the basic differences between the ODBC and the embedded SQL approach.

We then introduce the general ODBC concepts and facilities, to help us understand how SQL applications operate in an ODBC environment. Next, we describe the standard ODBC API, provide programming samples, and explain the features specific to DB2 ODBC. Finally, we provide several performance considerations for a DB2 ODBC environment.

## 4.2 ODBC, an alternative to embedded SQL

In the embedded SQL approach, SQL statements are mixed with other statements, in a host programming language; and moreover:

- ▶ Program variables can be directly included in SQL statements.
- ▶ Special SQL statements must be embedded in the program to process data row by row (for example, statements related to declare and open cursors).

As the program contains 2 types of languages, it cannot be directly processed by a compiler. A language precompiler is needed to extract the SQL statements (or the compiler has to call a coprocessor to do this extraction). The SQL statements are replaced by calls to the DBMS routines. As a consequence, the program is tied to the DBMS and cannot be ported to another one without (at least) being precompiled again using the new DBMS' precompiler or coprocessor.

However, embedding SQL in an existing language has some advantages:

- ▶ A BIND operation can be used to “compile” the extracted SQL into a plan or package, managed by the DBMS. Thanks to this operation, static SQL statements can be prepared in advance and performance can be highly improved at execution time.
- ▶ The programmer does not have to worry about the interface to DBMS routines, as the precompiler manages it.

The ODBC approach does not blend SQL with other programming language statements. It provides the developer with a *callable interface*, that is, a library of callable functions to interact with the DBMS. SQL statements are passed to the DBMS as string arguments of these functions. A precompiler or coprocessor is therefore not needed.



Following are some other general characteristics of callable interfaces:

- ▶ Specialized embedded SQL statements are unnecessary. Cursors for example, must not be explicitly declared or opened, as they are directly generated and managed by the called functions.
- ▶ Host variables cannot be embedded into SQL statements (as opposed to embedded SQL).
- ▶ Instead, specialized functions are used to associate program variables to parameter markers.
- ▶ Parameter markers can be used when asking for the direct execution of an SQL statement (as opposed to the EXECUTE IMMEDIATE embedded statement).
- ▶ Parameter and result sets descriptions are obtained via function calls. There is no need for the application to allocate and manage a structure like the SQLDA.
- ▶ An explicit connection to the DBMS is mandatory. (Note that when using embedded SQL to access DB2, the connection can be implicitly done if there is only one available DBMS.)
- ▶ Errors are reported to the program via diagnostic functions, as opposed to the DB2 embedded approach, where an SQLCA is needed.

The main advantage of ODBC is the portability it offers to applications:

- ▶ The same compiled code can be used on any DBMS that supports ODBC.
- ▶ The developer 's knowledge is portable. Knowledge of embedding techniques specific to each DBMS product is not necessary. However, since SQL statements are just string arguments of functions, DBMS specific SQL extensions can be used. However, they make the code less portable. ODBC allows you to use *escape* clauses (see 4.3.3, “Miscellaneous ODBC features” on page 54) to implement DBMS specific SQL and keep a comfortable level of portability.
- ▶ Thanks to a set of functions dedicated to catalog queries, applications can be used on any release of any DBMS 's catalog.

These are the main drawbacks of ODBC:

- ▶ ODBC, as any other callable interface, does not support static SQL, which *could* be a major issue from the performance point of view. However, ODBC incorporates several optimization techniques, as explained in 4.4, “The ODBC API” on page 55.
- ▶ The ODBC API specifications are public and largely used by end-user tools. Security can therefore be an issue. If end-users are granted the right to execute the ODBC packages on DB2 to run an application, they must be given authorizations on tables. They can possibly misuse these authorizations via any end-user tool able to call ODBC (or any other tool they are entitled to use, such as SPUFI or QMF).

The two aforementioned issues are treated in more detail in 4.7, “Performance considerations for ODBC applications” on page 84 and Chapter 13, “Security aspects of dynamic SQL” on page 221.

## 4.3 ODBC concepts and facilities

In this section we describe the environment and the concepts used by ODBC based applications.

### 4.3.1 The driver manager, the drivers, and the data sources

ODBC provides an environment where applications are not directly linked to function libraries. As shown in the upper side of Figure 4-1, applications are actually interfacing with the ODBC driver manager, which in turn interfaces with DBMS specific drivers.

DBMS drivers are components that implement the ODBC functions for a specific DBMS. A driver must be provided by each DBMS supplier supporting the ODBC standard. IBM provides it for NT and UNIX platforms as well as for the OS/390 or z/OS platform. To comply with other IBM publications, each time we don't explicitly specify the platform, "DB2 CLI" stands for the NT and UNIX version of the driver, and "DB2 ODBC" stands for the z/OS and OS/390 version.

Function calls are made locally from the program, but a layer is needed to manage the connection between the function and the DBMS across the network. In the case of an application running on NT, and accessing DB2 on z/OS, this layer is implemented by DB2 Connect. When running on OS/390 or z/OS, the ODBC application can access data on a local DB2 through CAF or RRSF, or can access remote databases through DRDA or the DB2 private protocol.

The driver manager is not supplied by the DBMS supplier. On the Windows platform, for example, it is delivered by Microsoft or any other vendor. The driver manager loads DBMS drivers, receives function calls and directs them to the right driver. A configuration file indicates to the driver manager which drivers are installed.

Note that DB2 ODBC on z/OS does not support a real driver manager, unlike DB2 ODBC on NT platforms (see the lower side of Figure 4-1). Actually, some functions of the driver manager are implemented in the driver itself. Other differences between DB2 ODBC on z/OS and on NT platforms are explained in 4.6.3, "Differences between the DB2 ODBC driver on z/OS and NT" on page 81.

Before being accessible through ODBC, a database must first be registered within the ODBC driver manager as a *data source*. A data source is an object that holds information about the database such as its name, the driver type to be used, and the address where the database server is located. If the target DBMS is DB2 on z/OS, the name of the database is actually the location name of the DB2 subsystem.

When requesting a connection, the program must provide the data source name to the driver manager. The latter is then able to load and interface with the appropriate driver.

Data sources can be defined through the user interface provided by the driver manager, or directly specified in an initialization file (called DB2CLI.INI on NT platforms. On z/OS platforms, the initialization file is defined by the DSNAINI DD card or UNIX environment variable). For each data source, ODBC parameters can be overridden, to dictate the behavior of ODBC transactions. In the ODBC terminology, these parameters are called *keywords*. Some of the keywords can help in managing SQL performance, as explained in 4.7, "Performance considerations for ODBC applications" on page 84. For more information about setting the initialization file to configure data sources, see Chapter 4 of *DB2 Universal Database for OS/390 and z/OS Version 7 ODBC Guide and Reference*, SC26-9941.

For DB2 ODBC applications running on z/OS:

- ▶ The data source name of a target DB2 is the location name in the DB2 SYSIBM.LOCATION table.
- ▶ The local database name to which the application is connected is the subsystem name given by the keyword MVSDEFAULTSSID.

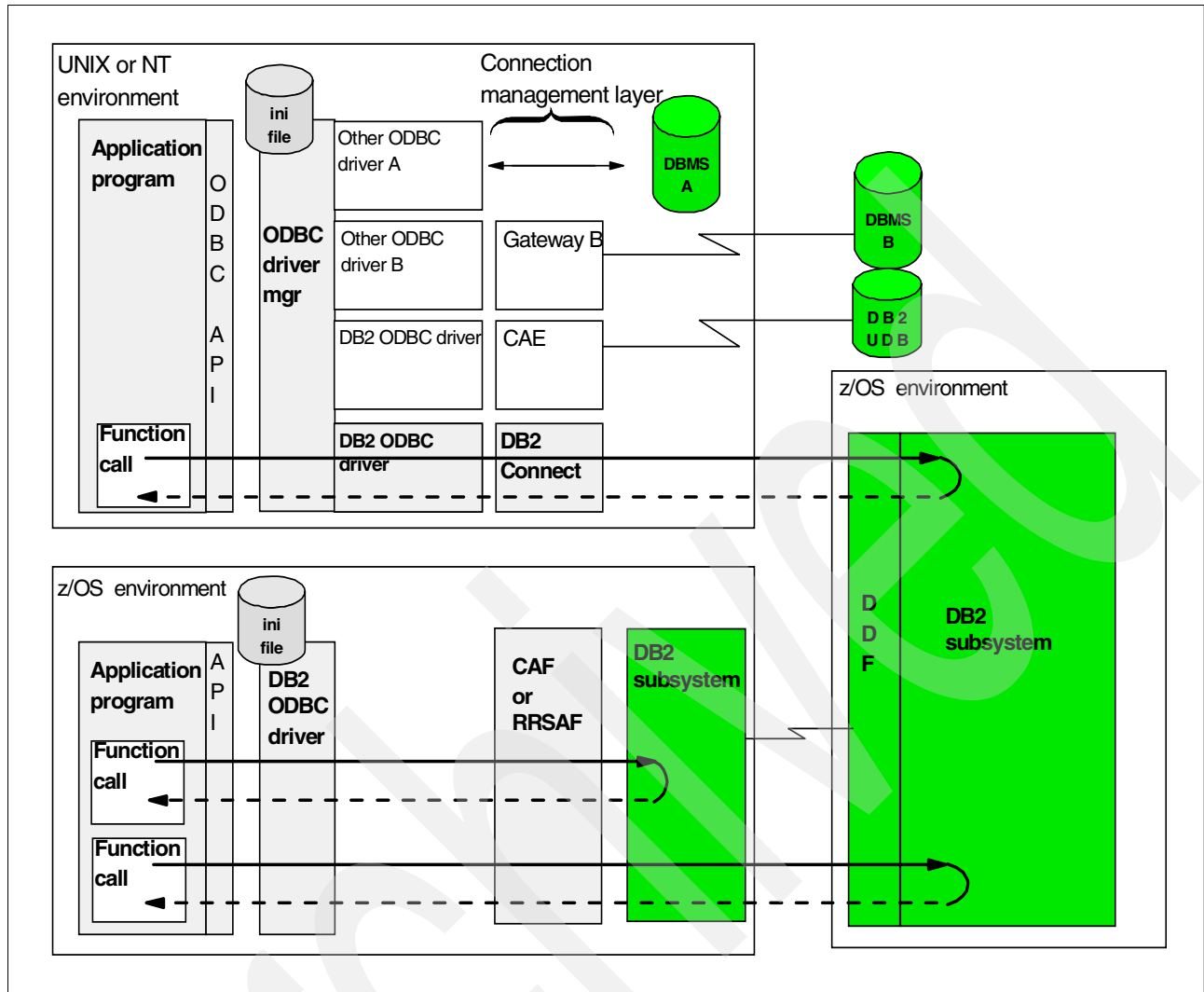


Figure 4-1 The ODBC environment

### 4.3.2 ODBC handles

To manage connections and SQL statements, ODBC uses so-called “handle” objects. There are three types of handles.

#### Environment handle

This can be seen as an abstract object containing global information about the application (for example, the version of ODBC currently used, the maximum number of connections allowed for the application). A program must first create an environment handle, before asking for a database connection.

#### Connection handle

This is an object representing a connection, and containing information about a specific data source (for example, the current schema name, the isolation level used within the connection). Such a handle must be created before asking for a connection to a data source. Connections handles must be created in reference to an already existing environment handle.

**Statement handle**

This is an object representing the SQL statement, and containing information about its behavior (for example, maximum number of rows returned to the application from a query, information specifying whether the cursor position is maintained after a query). A statement handle is created in reference to an existing connection handle. The statement handle must exist before a SQL statement can be launched.

An information field belonging to a handle is called an *attribute* of the handle. Using ODBC functions acting on handles to get or set their attribute values, programs can be informed about the status of connections and statements, or can change their behaviors.

When connecting to a data source, the application must specify a connection handle. An application can connect concurrently to several data sources, and can have several concurrent connections on the same data source, but it must allocate one connection handle per connection.

Each ODBC connection is independent and represents a transaction, maintaining its own unit of work. There are some restrictions and some enhancements on this connection model when using DB2 ODBC. See 4.6.4, “Connections supported by DB2 ODBC on z/OS or OS/390” on page 82 and 4.6.5, “Connections supported by DB2 ODBC on UNIX or NT” on page 83 for more information.

### 4.3.3 Miscellaneous ODBC features

Here are some more ODBC features worth noting:

- ▶ ODBC supports asynchronous execution of SQL, allowing programs to call functions and continue processing without waiting for the answer. This facility is requested by setting an attribute of the connection handle, and is foreseen for application running on single-threaded operating systems. Application running on multi-threaded operating systems should execute functions on separate threads instead of using asynchronous SQL.
- ▶ Programs can use descriptors, which are logical views on the information that ODBC must maintain about SQL parameters and retrieved columns. This information is also accessible through function calls, but in some cases, using the descriptors can be more efficient. For example, a descriptor describing a fetched row can be used to describe a row to be inserted in the same table.
- ▶ ODBC makes an automatic default conversion between SQL and C data types wherever it is possible. ODBC can also apply conversion explicitly requested by the programmer when assigning the value of a variable to a parameter, or assigning the value of fetched columns to variables. Chapter 3 of *DB2 Universal Database for OS/390 and z/OS Version 7 ODBC Guide and Reference*, SC26-9941 gives more information on conversion supported by DB2 ODBC.
- ▶ ODBC supports batch operations, whose goal is to optimize performance. In the ODBC context, batch operations include stored procedure calls, compound SQL (that is, group of multiple independent SQL statements), bulk operations that use single statement with an array of parameters, and bulk operations to retrieve multiple rows in an array.

- ▶ ODBC provides the ability to exploit SQL extensions provided by DBMS vendors. An SQL statement can contain *escape* clauses to ask for such extensions. The statement must be scanned by the driver to search for escape clauses that translate them to equivalent clauses supported by the target DBMS. If the ODBC application access only DB2 data sources, escape clauses are not needed. If the application must access other data sources that provide the same support with another syntax, then the escape clauses improve portability.
- ▶ ODBC supports unicode (the UCS-2 encoding form), a 16-bit encoding standard which use double-wide characters to support international character sets.
- ▶ ODBC supports the four isolation levels defined in SQL92 (read uncommitted, read committed, repeatable read, and serialized read) and adds versioning as a fifth isolation level. This latter is not applicable to DB2.
- ▶ ODBC defines a set of facilities to manipulate scrollable cursors.

These facilities are not all supported by the DB2 driver on z/OS. For more information, see 4.6.3, “Differences between the DB2 ODBC driver on z/OS and NT” on page 81.

## 4.4 The ODBC API

This section gives an overview of common ODBC functions and the general flow of calls used in an ODBC program. Section 4.5, “ODBC programming samples” on page 63 gives more details, including several programming examples.

### 4.4.1 Functions overview

Although the ODBC API differs completely from the embedded SQL API, any SQL statement that can be dynamically prepared in embedded SQL can be executed by ODBC as a string argument of a function. (The only exceptions are COMMIT and ROLLBACK, as explained later on.)

Section 4.5.9, “Mapping between embedded statements and ODBC functions” on page 78, gives a summary of ODBC functions to use if there is a need to replace embedded SQL that cannot be dynamically prepared.

The ODBC specification provides several categories of functions, some of which offer optimization mechanisms. The following categories are not official. They are introduced for pedagogical reasons to illustrate the programming steps described in section 4.5, “ODBC programming samples” on page 63.

#### ▶ Functions to manipulate handles

These functions provide the interface to allocate or free handles, and to get or set their attributes. These functions must be used to initialize or terminate the ODBC application, connect (or disconnect) to (from) data sources, and to allocate or free SQL statements. We also place in this category the function to commit or rollback a transaction, because it acts on a connection or environment handle.

These functions are illustrated in Figure 4-2 and Figure 4-3, and explained in these programming steps:

- “Programming step 1: Initializing the application” on page 64
- “Programming step 2: Manipulating statement handles” on page 66
- “Programming step 6: Terminating the ODBC application” on page 76

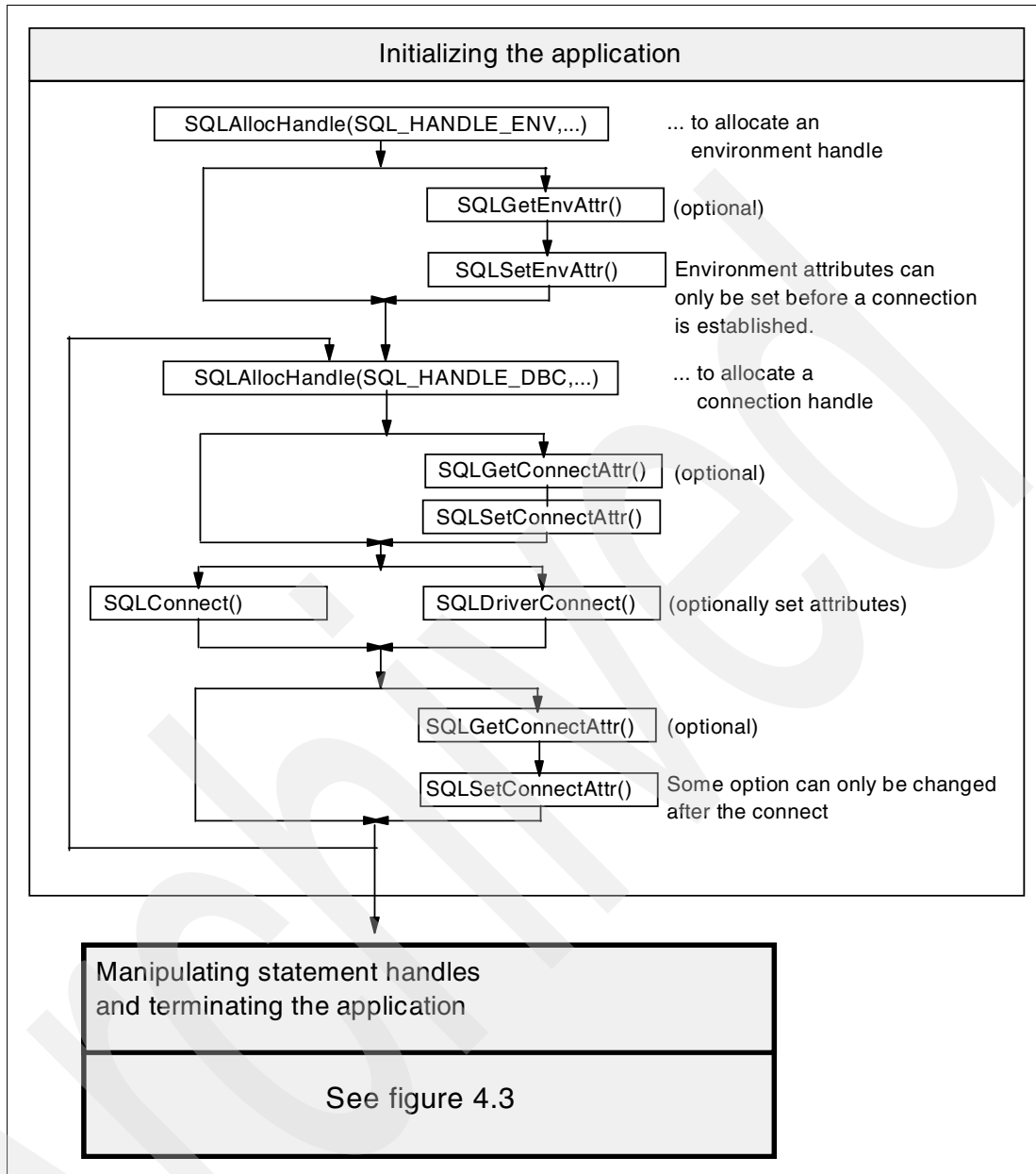


Figure 4-2 ODBC functions to manipulate handles (Part 1)

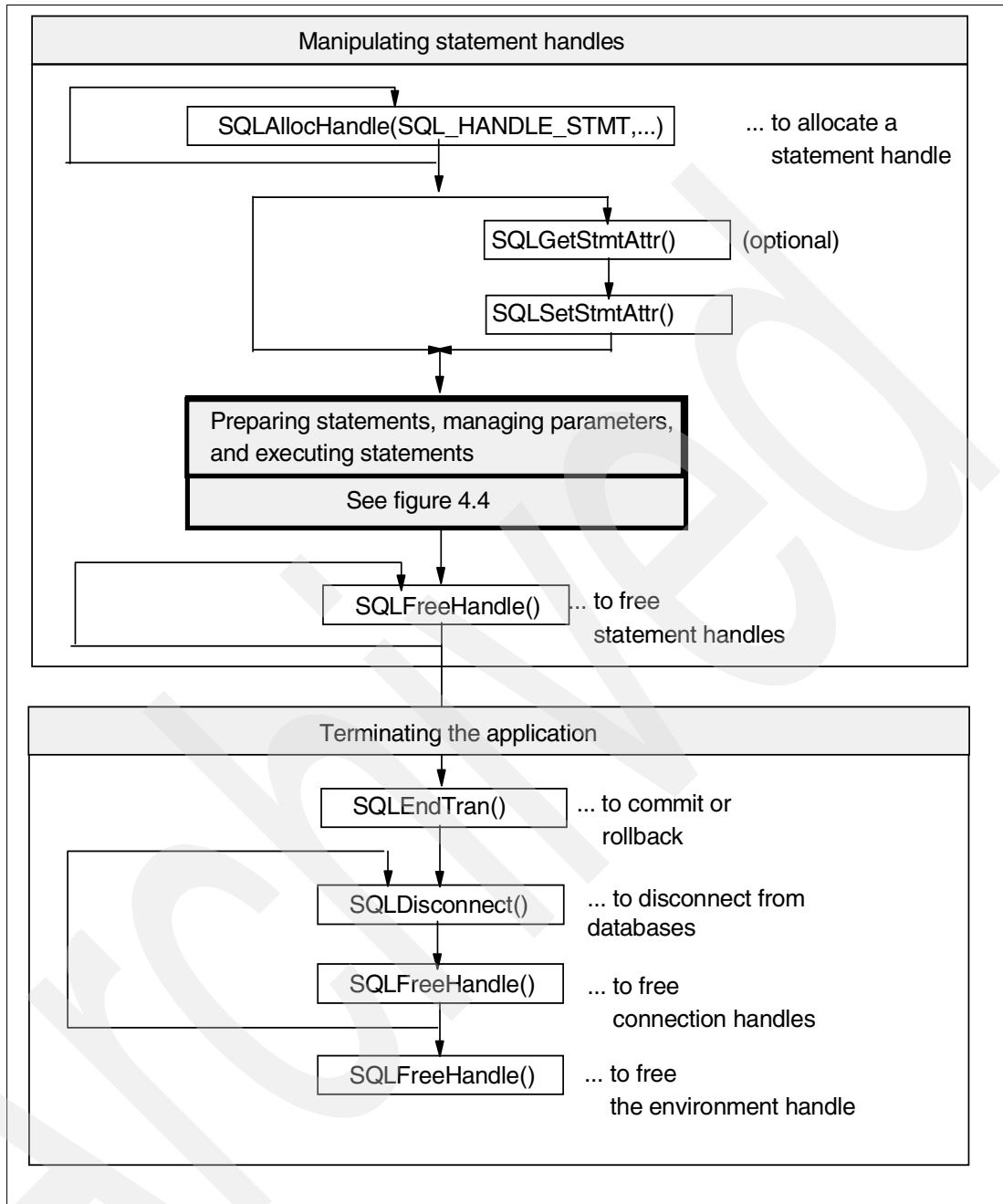


Figure 4-3 ODBC functions to manipulate handles (Part 2)

#### ► Functions to prepare the execution

In this category are functions to prepare SQL statements and to manage parameters. A prepared statement can be executed several times, possibly with different values assigned to the parameters. Functions exist to ask for the number of parameters, to describe parameters or to associate program variables with parameters. Some options allow the use of arrays as input parameter values. This technique avoids repeated executions of the same statement with different parameter values, and can save network flows.

These functions are illustrated in Figure 4-4 and explained in the programming steps:

- “Programming step 3: Preparing the execution” on page 66
- “Programming step 5: Managing execution results” on page 71

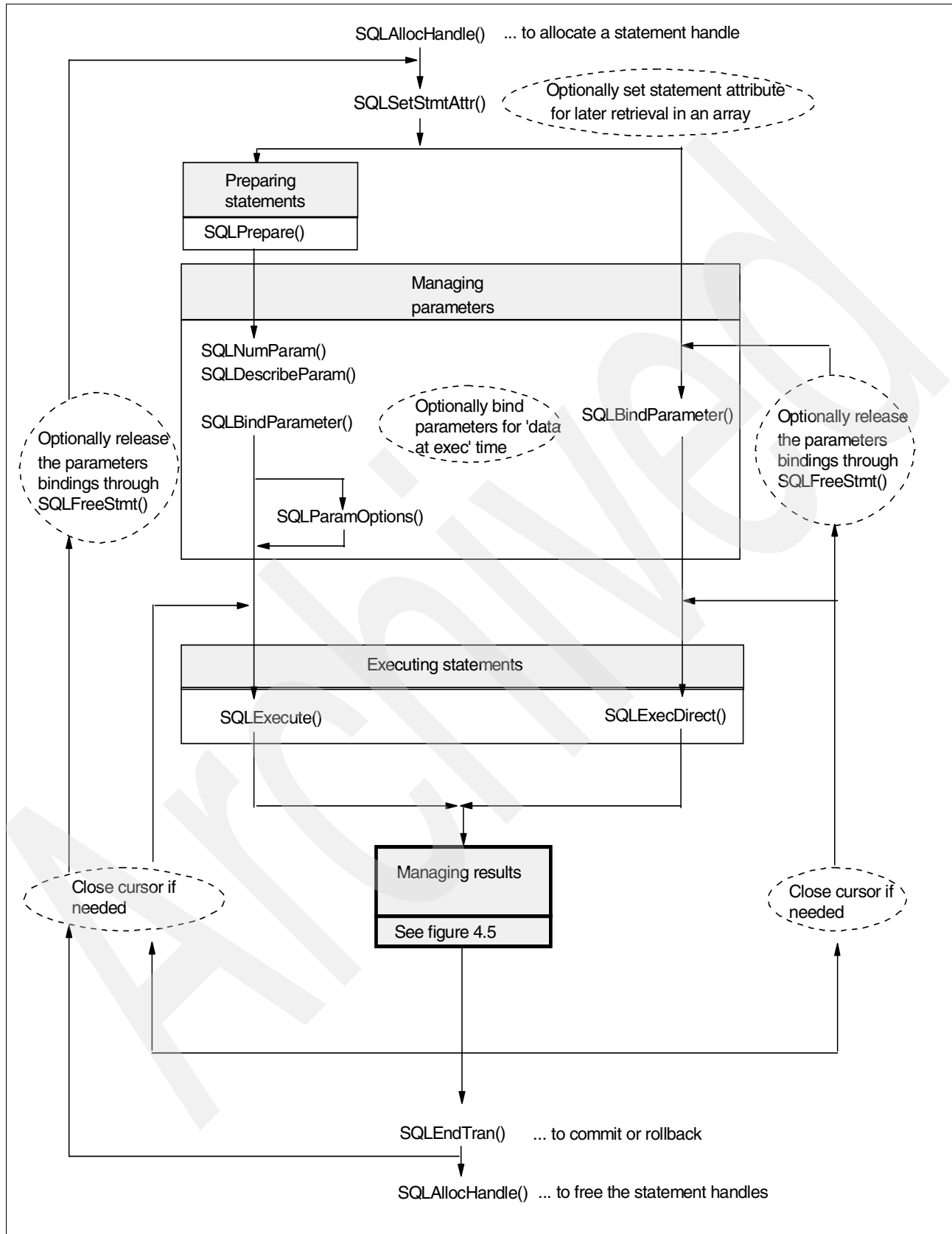


Figure 4-4 ODBC functions to prepare the execution and execute SQL statements



► **Functions to execute SQL statements**

It is possible to execute an already prepared statement or to ask for the prepare operation and the execution in one call. As long as a statement handle is not freed, it can be reused to prepare or execute another statement.

These functions are illustrated in Figure 4-4, and explained in the programming step:

- “Programming step 4: Executing statements” on page 70

► **Functions to manage execution results**

If the statement updates, deletes or inserts data, a specific function can be called to retrieve the number of rows updated. No special functions are related to the execution of DDL statements.

If the statement is a query, functions can be used to retrieve result sets, to describe columns and to associate columns with program variables. ODBC creates and opens a cursor for each result set. If an input array is used as parameter, multiple result sets will be received and must be processed one by one.

Some options allow you to retrieve multiple rows at a time into an array. Note that specific techniques must be used to process positioned updates or deletes.

These functions are illustrated in Figure 4-5 through Figure 4-8, and explained in the programming step:

- “Programming step 5: Managing execution results” on page 71.

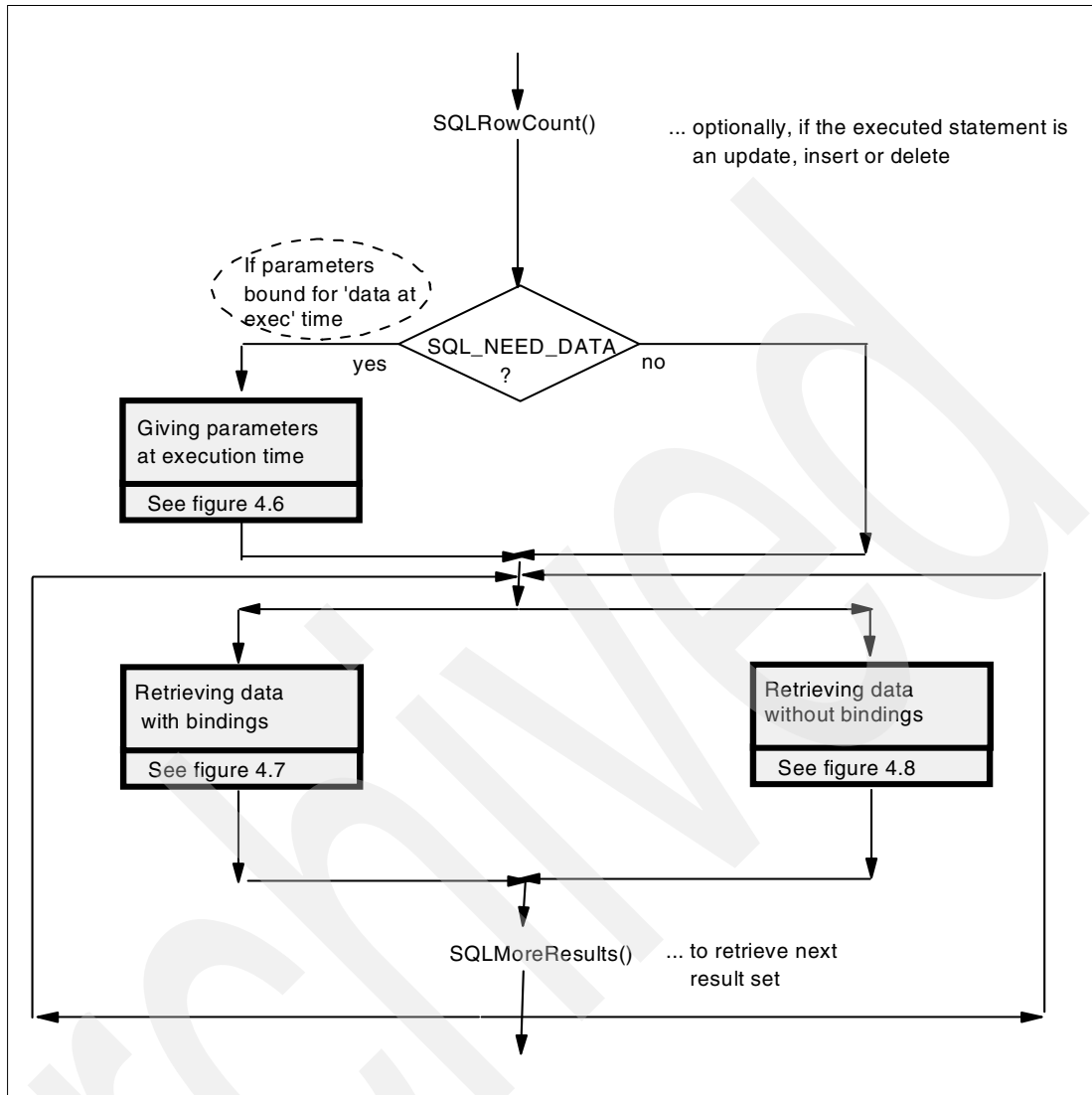


Figure 4-5 ODBC functions to manage execution results (Part 1)

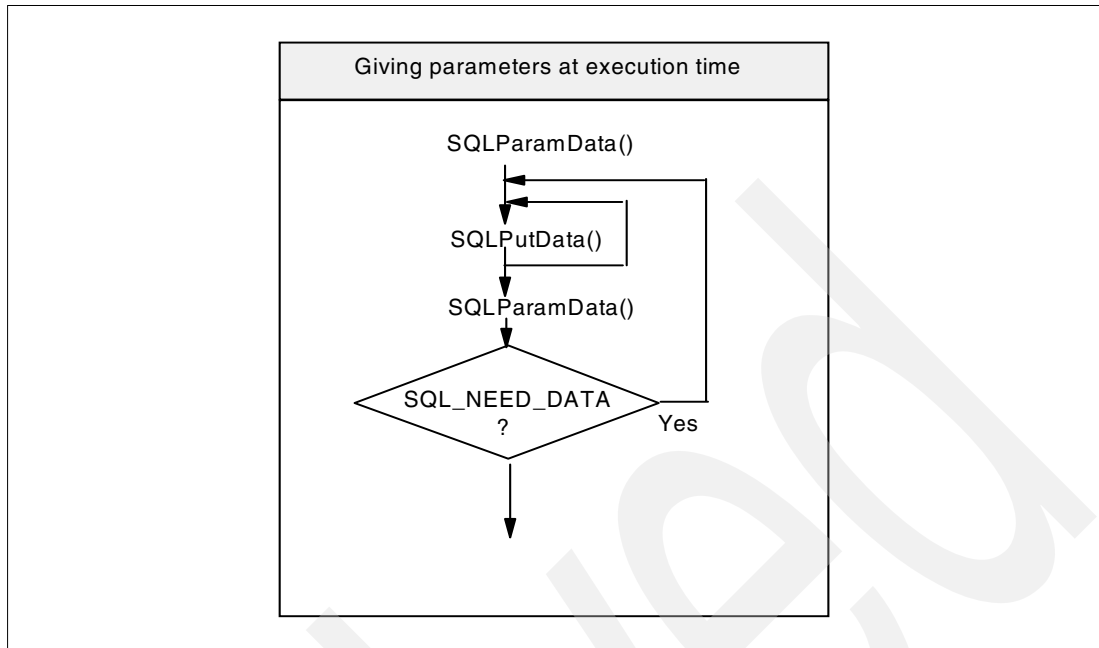


Figure 4-6 ODBC functions to manage execution results (Part 2)

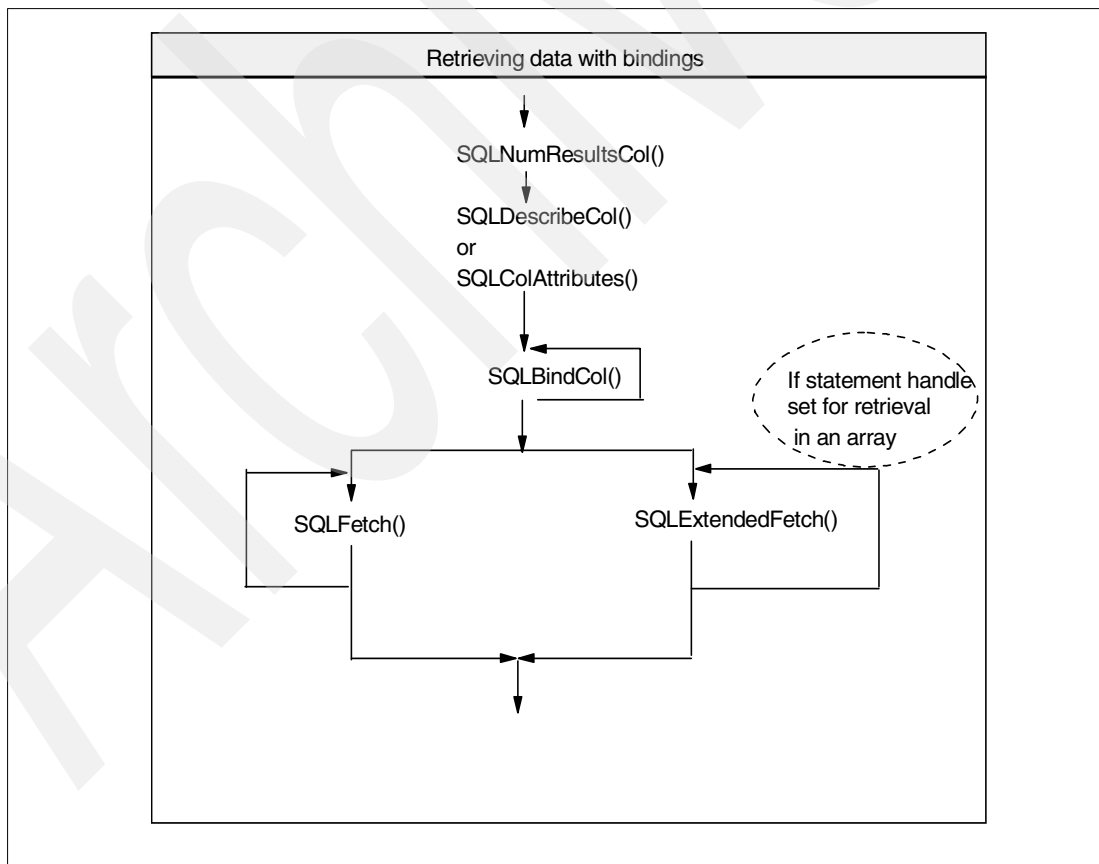


Figure 4-7 ODBC functions to manage execution results (Part 3)

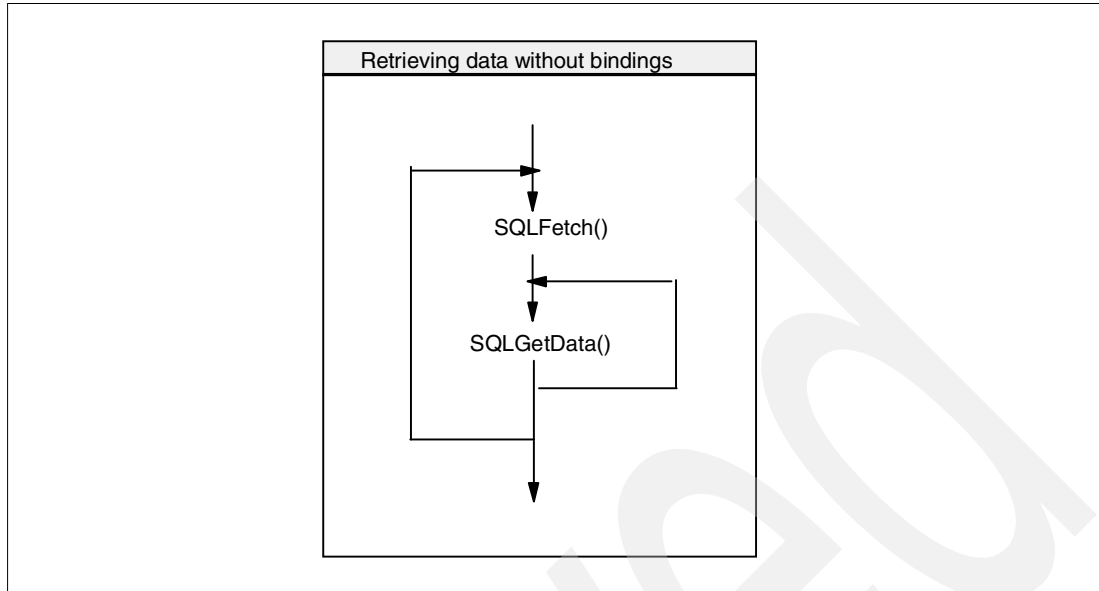


Figure 4-8 ODBC functions to manage execution results (Part 4)

► **Functions to retrieve diagnostics**

Each call to a function returns a code. When more detailed diagnostic information is available, it can be received by calling a specific function, which provides the `SQLCODE` and `SQLSTATE` with its associated text.

These functions are illustrated in Figure 4-9 and explained in programming step:

- “Programming step 7: Retrieving diagnostics” on page 78.

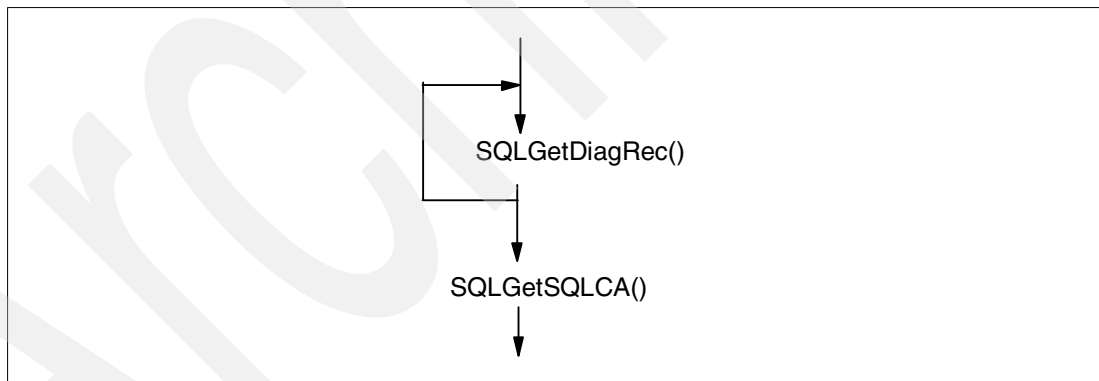


Figure 4-9 ODBC functions to retrieve diagnostics

► **Functions to retrieve information about drivers and data sources**

These functions don't really handle relational data. They are related to the ODBC driver manager. Such functions can for example be used to ask for the list of available data sources before deciding which database to connect to.

► **Functions to manipulate large objects**

In this category we find functions to handle LOB locators. These functions are provided by DB2 ODBC, but do not belong to the current ODBC standard.

► **Functions to query the catalog**

ODBC provides the developer with a set of functions to query the catalog. Using these functions increase the portability of the applications, since the driver does the translation between the standard ODBC function and the targeted catalog.

The three last sets of functions are not specifically related to dynamic SQL and are consequently not illustrated in this book. For more information on these functions, see chapters 5 and 6 of *DB2 Universal Database for OS/390 and z/OS Version 7 ODBC Guide and Reference*, SC26-9941.

## 4.5 ODBC programming samples

In this section we explain, step-by-step, how calls must be programmed to connect to databases and to execute SQL statements. Standard ODBC functions not supported by the DB2 driver on z/OS are not described. Not all of the other numerous ODBC functions are analyzed, but details are given about functions to prepare, execute, and retrieve results of dynamic SQL. We especially focus on:

- ▶ The dynamic aspect of SQL, like the management of unknown SQL parameters and unknown result sets
- ▶ Special bulk ODBC operations intended to make dynamic SQL more efficient in client server environments

The walk-through of the programming steps gives a solid understanding of ODBC programming techniques. An ODBC C program, implementing the different steps can be downloaded from the web. Instructions are in Appendix B, “Additional material” on page 251.

### 4.5.1 General format of ODBC function calls

The ODBC function call has the same general format as any other function call in C:

```
rc = FunctionName(argument1, ..., argumentn)
```

The return code takes the form of a symbolic constant, for instance: `SQL_SUCCESS` indicating that the function completed with success or `SQL_ERROR` indicating that the function failed.

The function can have input arguments which values must be provided by the program or output arguments, which values are provided by ODBC when the function completes successfully. An argument can for example be a constant, a variable defined in the program, a pointer or a *string* argument.

An input string is represented by two arguments in the argument list:

- ▶ A buffer where the program must place the value of the string
- ▶ A length argument given by the program, indicating one of the following:
  - The length of the argument
  - `SQL_NTS` to indicate that the string is terminated by a null character (in which case ODBC determines the length of the argument)
  - Or `SQL_NULL_DATA` to pass a NULL value.

An output string is represented by three arguments in the argument list:

- ▶ A buffer where ODBC must return the output
- ▶ An input length argument, giving the length of the allocated output buffer
- ▶ An output length argument where ODBC puts the actual length of the string.

## 4.5.2 Programming step 1: Initializing the application

Initializing an application is done through handle allocations, which are requested by the `SQLAllocHandle()` function, irrespective of the type of the handle. This function has 3 arguments:

- ▶ The type of the handle to be allocated:
  - Use `SQL_HANDLE_ENV` to allocate an environment handle
  - Use `SQL_HANDLE_DBC` to allocate a connection handle
  - Use `SQL_HANDLE_STMT` to allocate a statement handle
- ▶ An input handle, on which the requested handle depends
- ▶ A pointer to the buffer in which the allocated handle will be returned

The returned handle is actually a variable that refers to the handle object managed by ODBC. This variable is used in statements where a reference to the handle is needed.

### Allocating the environment handle

The first handle to be created is the environment handle. DB2 ODBC on z/OS doesn't allow more than one environment handle per application, but DB2 ODBC on NT or UNIX does.

Example 4-1 shows how to allocate an environment handle. The "henv" variable represents the handle allocated by ODBC. `SQLHENV` and `SQLRETURN` are the C symbolic data types of handles and codes returned by ODBC functions. These symbolic types are derived from basic C type and described in the chapter 3 of *DB2 Universal Database for OS/390 and z/OS ODBC Guide and Reference*, SC26-9941.

The constant `SQL_HANDLE_ENV` specifies that the handle to be allocated is an environment handle. `SQL_NULL_HANDLE` must be used to represent a null input handle, since environment handles do not depend on any other handle.

`SQLGetEnvAttr()` is used to read the default connection type into a variable. The argument to receive this information is represented by a pointer to the variable "contype", and its associated input and output lengths. These lengths are set to 0 because the retrieved value is not a character type.

If the connect type is not `SQL_COORDINATED_TRANS`, then `SQLSetEnvAttr()` is used to force this value. The argument used to set this value is actually represented by a pointer to the desired value, followed by the length of the value. This length is set to 0 because the `SQL_ATTR_CONNECTTYPE` attribute is not a character.

More information about the meaning of `SQL_COORDINATED_TRANS` can be found in 4.6.4, "Connections supported by DB2 ODBC on z/OS or OS/390" on page 82.

#### Example 4-1 Creating the environment handle

---

```
SQLHENV      henv;  
SQLRETURN    rc;  
SQLINTEGER   contype;  
  
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv );  
rc = SQLGetEnvAttr(henv, SQL_ATTR_CONNECTTYPE, &contype, 0, 0);  
  
if (contype != SQL_COORDINATED_TRANS)  
    {rc=SQLSetEnvAttr(henv, SQL_ATTR_CONNECTTYPE,(void*) SQL_COORDINATED_TRANS, 0);
```

---

## Allocating connection handles and connecting to databases

After the environment handle, connection handles can be allocated. The connection itself is requested through either of the following:

- ▶ The `SQLConnect()` function with arguments to represent the connection handle, the data source, a userid, and a password
- ▶ The `SQLDriverConnect()` function, which gives the possibility to override several keywords of the initialization file.

When creating the allocation handle with DB2 ODBC on z/OS, the DB2 subsystem name must be available to ODBC. DB2 finds it either in the initialization file (keyword `MVSDEFAULTSSID`) or in the `DSNHDECP` (default SSID) application programming defaults module. If ODBC does not find a valid subsystem name, the connection allocation fails.

Example 4-2 allocates a connection handle by specifying `SQL_HANDLE_DBC` as input argument for the type of the handle and specifying the “`henv`” variable for the environment handle. The connection handle is returned in the third argument, “`hdbc`”.

A connection is done to the data source specified by the variable “`dsn`”. `SQL_NTS` specifies that the 3 string arguments are ended with a null string.

The example also checks the value of the default commit mode and forces it to the manual mode. For information about the commit mode, see “Programming step 6: Terminating the ODBC application” on page 76.

*Example 4-2 Creating a connection handle and connecting to databases*

---

```
SQLHDBC          hdbc;
char *           dsn="DB2G";
char *           uid="xxx";
char *           pwd="yyy";
SQLINTEGER       autocommit;

rc=SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc);

rc = SQLGetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, &autocommit, 0, NULL);
if (autocommit == SQL_AUTOCOMMIT_ON)
{rc=SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT,(void*) SQL_AUTOCOMMIT_OFF,SQL_NTS);
rc=SQLGetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, &autocommit, 0, NULL); }

rc=SQLConnect(hdbc, dsn,SQL_NTS,uid, SQL_NTS,pwd, SQL_NTS);
```

---

Example 4-3 shows the usage of `SQLDriverConnect()` to connect to a data source and overrides the value of the connect type and the default commit mode given in the initialization file. The first argument of this function is the connection handle. The second argument must always be `NULL` (argument not used). The other arguments are:

- ▶ A pointer to the connection string
- ▶ The length of the connection string
- ▶ A pointer to a buffer to contain the connection string when completed by ODBC
- ▶ The maximum size of this buffer
- ▶ A pointer to the length of the connect string completed by ODBC
- ▶ `SQL_DRIVER_NOPROMPT` (the only value supported by DB2. Other values could specify whether the user must be prompted at run time to complete the connect string)

*Example 4-3 Connecting to database through SQLConnectDrive()*

---

```
SQLHDBC          hdbc;
char              ConnStrIn="dsn=DB2G;uid="xxxx";          \
                  pwd="yyyy";autocommit=0,connecttype=2";
char              ConnStrOut[200];
SQLSMALLINT       LengthConnStrOut;

rc=SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc)
rc=SQLConnectDrive(hdbc, NULL, (SQLCHAR *) ConnStrIn, strlen(ConnStrIn),
                  SQLCHAR *) ConnStrOut, sizeof(ConnStrOut), &LengthConnStrOut,
                  SQL_DRIVER_NOPROMPT);
```

---

### 4.5.3 Programming step 2: Manipulating statement handles

After the connection handles, statement handles can be created. SQLGetStmtAttr() and SQLSetStmtAttr() can be used to read or to override statement attributes.

Example 4-4 shows how statement handles are allocated. It also shows the setting of two attributes: SQL\_ATTR\_NOSCAN and SQL\_ATTR\_CURSOR\_HOLD. For more information about these attributes, see 4.7.1, "ODBC usage and programming hints" on page 84.

*Example 4-4 Manipulating statement handles*

---

```
SQLHSTMT          hstmt;

rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);

rc=SQLSetStmtAttr( hstmt, SQL_ATTR_NOSCAN, (void *) SQL_NOSCAN_ON, 0);
rc=SQLSetStmtAttr( hstmt, SQL_ATTR_CURSOR_HOLD, (void *) SQL_CURSOR_HOLD_OFF, 0);
```

---

### 4.5.4 Programming step 3: Preparing the execution

Here we cover the various activities that are needed in preparing the execution.

#### Preparing statements

Associating an SQL statement with a statement handle and asking the DBMS to PREPARE the statement can be done in one step, by using the SQLPrepare() function. The prepared statement is then referenced by the statement handle. Preparing a statement makes the description of parameters and columns of the result set available to the program via specific function calls. A result set is the set of rows obtained from a stored procedure call *or* from a query. If the statement handle has previously been used with a statement returning a result, the cursor opened by ODBC must be closed by using SQLCloseCursor() or SQLFreeStmt(). SQLCloseCursor() returns an error if no cursor is open, while SQLFreeStmt() does not.

Example 4-5 below shows the use of SQLFreeStmt() to close the cursor before calling the SQLPrepare() function.

*Example 4-5 Preparing statements*

---

```
SQLCHAR          stmt1="SELECT EMPNO FROM EMP WHERE JOB='XX' AND SEX='M';
SQLCHAR          stmt2="SELECT EMPNO FROM EMP WHERE JOB='XX' AND SEX='Y';
```

---



```
rc=SQLPrepare(hstmt, stmt1, SQL_NTS);
/** execute the statement here, and process the result set **/

rc=SQLFreeStmt(hstmt, SQL_CLOSE);
rc=SQLPrepare(hstmt, stmt2, SQL_NTS);
/** execute the statement here, and process the result set **/
```

---

## Describing parameters

Once the statement is prepared, the function `SQLNumParams()` can be called to determine the number of parameter markers, and `SQLDescribeParam()` can be called for each parameter, to find out its type, length (precision and scale), and whether it is nullable, as shown in Example 4-6. This information is used later on to bind parameters to program variables. Calling these functions is not necessary if the number and type of parameters are known.

*Example 4-6 Describing parameters*

---

```
SQLSMALLINT      numparam, sqltype, sqlscale, sqlnullable;
SQLUINTEGER      sqlprecision;
SQLINTEGER       i;

numparam=0;
rc=SQLNumParams(hstmt, &numparam);

for (i = 0 ; i < numparam ; i++)
    { rc=SQLDescribeParam(hstmt, i+1, &sqltype, &sqlprecision, &sqlscale, &sqlnullable); }
```

---

## Binding parameters

Before running a statement, each parameter (represented by a question mark) must be bound:

- ▶ Either to a program variable. In this case, a value must be assigned to the variable before running the statement.
- ▶ Or to the constant `"SQL_DATA_AT_EXEC"`. In this case the parameter is not known and ODBC will inform the program at execution time, that a value is missing.

*Binding parameters* is done by calling the `SQLBindParameter()` function for each parameter. Once a parameter is bound, the binding can be reused for several assignments and several executions, at least as long as the statement handle is not freed. If the statement handle is reused for another SQL statement with other parameter specifications, the `SQLFreeStmt()` function must be called to reset the parameter bindings.

Binding parameters can be done before or after the call to `SQLPrepare()`. If the number and type of parameters is unknown, special functions can be used to determine them, but the statement must have been prepared before.

`SQLBindParameter()` needs the following arguments:

- ▶ The statement handle associated to the statement.
- ▶ The number of the parameters to be bound. The parameters are ordered sequentially from left to right, starting with 1.
- ▶ The type of the parameter. If the type is `SQL_PARAM_INPUT`, it means that the program deals with a parameter marker belonging to an SQL statement. Other types are related to parameter markers in a stored procedure call.
- ▶ The C data type of the parameter.

- ▶ The SQL data type of the parameter. These two last arguments allows ODBC to explicitly perform a data conversion.
- ▶ Precision of the parameter (ignored if the parameter does not need it).
- ▶ Scale of the parameter (ignored if the parameter does not need it).
- ▶ A pointer to the variable that must contain the parameter value.
- ▶ The length of this variable (ignored for non-character and non-binary data). The application variable and its length are called *deferred input arguments*, because their values must not be known at the bind time, but later on, at the statement execution time.
- ▶ A pointer to a variable which must contain the length of the parameter value. This variable can also contain:
  - NULL, to specify that the parameter value is a “null-terminated” string. In this case ODBC is able to calculate the length.
  - SQL\_NULL\_DATA to indicate a null value for the parameter.
  - SQL\_DATA\_AT\_EXEC to specify that the parameter value must be prompted at execution time.

Example 4-7 shows how to bind an SQL parameter to a program variable. The SQLDescribeParameter() function is called before the SQLBindParameter() in order to get the type, precision, and scale needed for the binding.

The statement “stmt1” is executed with different values, but reusing the same binding. The statement “stmt2” is associated to the statement handle, but executed with a new binding. The previous binding must first be released by SQLFreeStmt(), with argument SQL\_RESET\_PARAMS.

Because the length of the parameter given in the SQLBindParameter() function is NULL (last argument), the variable must be a “null terminated string”. This is the reason why the length of the variable is equal to the length of the DB2 column + 1 (length of the null character terminating the string).

*Example 4-7 Binding parameters to program variables*

---

```

SQLSMALLINT      sqltype, sqlscale, sqlnullable;
SQLINTEGER       sqlprecision;
char             empno_prm[7];
char             workdept_prm[4];
SQLCHAR          stmt1[40]=[SELECT WORKDEPT FROM EMP WHERE EMPNO=?];
SQLCHAR          stmt2[40]=[SELECT EMPNO FROM EMP WHERE WORKDEPT=?];

/** prepare the statement **/
rc=SQLPrepare(hstmt, stmt1,SQL_NTS);

rc=SQLDescribeParam(hstmt, 1, &sqltype, &sqlprecision,&sqlscale,&sqlnullable);
rc=SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, sqltype,
sqlprecision, sqlscale,empno_prm, 7, NULL);

/** associate a value to the bound parameter **/
strcpy((char *)empno_prm,"000099");

/** execute the statement here, and process the result set **/

/** associate another value to the bound parameter **
strcpy((char *)empno_prm,"000010");

/** execute the statement here, and process the result set **/

rc=SQLFreeStmt(hstmt, SQL_CLOSE);

```

```

/** reset the parameter binding */
rc=SQLFreeStmt(hstmt, SQL_RESET_PARAMS);

/** prepare the statement */
rc=SQLPrepare(hstmt, stmt2, SQL_NTS);

rc=SQLDescribeParam(hstmt, 1, &sqltype, &sqlprecision, &sqlscale, &sqlnullable);
rc=SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, sqltype,
sqlprecision, sqlscale, workdept_prm, 4, NULL);

/** associate a value to the bound parameter */
strcpy((char *)workdept_prm, "A00");

/** execute the statement here, and process the result set */

```

---

## Binding parameters to an array of program variables

One call can bind one parameter to an array of variables. This is an efficient way to process the same statement with different parameter values within the same set of prepare, bind, and execute calls. If the statement is a query, the output contains multiple result sets after the execution.

Example 4-8 shows the use of `SQLParamOptions()` to inform ODBC that the parameter will be bound to an array containing 3 elements. In the example, this function is called before `SQLBindParameter`, but could be called after. The variable “pirow” is an output argument which contains the index of the last array element processed by ODBC at the end of the execution. It can be used in case an error occurs to find out which element of the array failed.

*Example 4-8 Binding a parameter to an array of variables*

```

SQLCHAR      stmt_array[50]="SELECT FIRSTNME FROM DSN8710.EMP WHERE WORKDEPT=?";
SQLSMALLINT  sqltype, sqlscale, sqlnullable;
SQLINTEGER   sqlprecision, pirow;
char         workdept_prm[3][4] = {"A00", "D11", "E11"};

rc=SQLPrepare(hstmt, stmt_array, SQL_NTS);
rc=SQLParamOptions(hstmt, 3, &pirow);

sqltype=SQL_CHAR;
sqlprecision=3;
sqlscale=0;
rc=SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, sqltype, sqlprecision,
sqlscale, workdept_prm, 4, NULL);

/** execute the statement here, and process the result set */

```

---

## Binding parameters to enter values at execution time

Assigning `SQL_DATA_AT_EXEC` to the input data length in `SQLBindParameter()` means that the value of a parameter is only known at execution time.

ODBC must inform the program during the execution that the parameter value is missing, and must wait for a value.

Giving the parameter value at execution time can be useful when the parameter is too long to be specified before the execution. ODBC provides a method to enter too long values in pieces during the execution of the statement. This method is explained in “Specifying a parameter value at execution time” on page 71.

Example 4-9 shows how the SQLBindParameter() function can be used to notify ODBC that the parameter value must be entered at execution time. Note that the parameter that normally points to the program variable, points to a number value which represent the parameter. The number of the parameter is usually used. In the example, "(SQLPOINTER) 1" is used, meaning that the missing parameter is the first one. This value is returned to the program at execution time to help in the identification of the missing variable, as shown in Example 4-12.

*Example 4-9 Binding parameters to enter value at execution time*

---

```

SQLINTEGER      atexec=SQL_DATA_AT_EXEC;
SQLSMALLINT     sqltype, sqlscale, sqlnullable;
SQLUINTEGER     sqlprecision;
SQLCHAR         stmt_atexec[80]=["SELECT FIRSTNME FROM DSN8710.EMP WHERE      \
                                WORKDEPT=?"];

sqltype=SQL_CHAR;
sqlprecision=3;
sqlscale=0;
rc=SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, sqltype,
sqlprecision, sqlscale, (SQLPOINTER) 1, 0 , &atexec);
/** execute the statement here, and process the result set **/

```

---

## 4.5.5 Programming step 4: Executing statements

There are two ways of executing SQL statements:

- ▶ Prepare and then execute :
  - Call SQLPrepare() with an SQL statement as an argument
  - Potentially call SQLNumParams() to receive the number of parameters, and SQLDescribeParams() to know the type of parameters
  - Potentially call SQLBindParameter for each parameter
  - Call SQLExecute()
- ▶ Execute directly:
  - Call possibly SQLBindParameter() for each parameter
  - Call SQLExecDirect() with an SQL statement as an argument

The first method must be used if the number and type of parameters is not known. It should also be used when the statement is executed frequently. In that case, the PREPARE operation must be done only once for all subsequent executions.

The second method cannot be used if the number of parameters is not known, except if all parameters are bound to "SQL\_DATA\_AT\_EXEC".

As it prepares and executes the statement in one call, SQLExecDirect() should be used whenever the statement must run only once, in order to avoid calling two functions (and potentially save one message flow to DB2). Unlike the EXECUTE IMMEDIATE statement in embedded SQL, SQLExecDirect() allows the direct execution of SELECT statements.

If the statement is a query, the execution generates a cursor with an internal name. The programmer can associate a name to this cursor via SQLSetCursorName(), or can obtain the generated name via SQLGetCursorName(). Cursors on the statement handle must be closed before executing another statement.

Example 4-10 shows the 2 execution methods.

---

*Example 4-10 Executing statements*

---

```
SQLCHAR      stmt1[43]=[SELECT EMPNO FROM EMP WHERE WORKDEPT='A00''];
SQLCHAR      stmt2[43]=[SELECT EMPNO FROM EMP WHERE WORKDEPT='E11''];

rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
rc=SQLPrepare(hstmt,stmt2,SQL_NTS);
rc=SQLExecute(hstmt);

rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmt);

rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
rc=SQLExecDirect(hstmt,stmt1,SQL_NTS);
```

---

## 4.5.6 Programming step 5: Managing execution results

In this step we consider how to manage the execution results.

### Counting the number of rows involved in the execution

As illustrated in Example 4-11, the function `SQLRowCount()` can be used to count the number of rows involved in an UPDATE, a DELETE or an INSERT operation. This function returns -1 for a mass delete in a DB2 segmented table space. There are no functions to count the number of rows selected by a SELECT statement, and there are no special functions to be used after a DDL statement.

---

*Example 4-11 Counting the number of updated, deleted, or inserted rows*

---

```
SQLINTEGER    numrows;
SQLCHAR      stmtin[43]=[DELETE EMPNO FROM EMP WHERE WORKDEPT='E11''];

rc=SQLExecDirect(hstmt,stmtin,SQL_NTS);
rc=SQLRowCount(hstmt,&numrows);
```

---

### Specifying a parameter value at execution time

Example 4-12 executes the statement defined in Example 4-9. It illustrates the use of `SQLParamData()` and `SQLPutData()`.

- ▶ `SQLParamData()` returns `SQL_NEED_DATA` if a parameter value is missing, and it supplies the number of the missing parameter in an output argument (here, "prgbvalue").
- ▶ `SQLPutData()` is used to assign the value to the parameter. It can be used in a loop to assign a value in pieces. In the example, the value of the work department ("A00") is given in pieces of 1 character.

---

*Example 4-12 Specifying a parameter value at execution time*

---

```
SQLPOINTER    prgbvalue ;
char          workdept_exe[4] = "A00";
char          put_exe[1] ;

rc=SQLExecDirect(hstmt,stmt_atexec,SQL_NTS);
if (rc == SQL_NEED_DATA)
{
    rc=SQLParamData(hstmt, &prgbvalue);
    while (rc== SQL_NEED_DATA)
    {
        if ((SQLPOINTER) prgbvalue == (SQLPOINTER) 1 )
        {
            for ( i=0 ; i<3 ; i++)
```

```

        {
            put_exe[0]=workdept_exe[i] ;
            rc=SQLPutData(hstmt,put_exe,1);
        }
    }
    if ((SQLPOINTER) prgbvalue == (SQLPOINTER) 2 )
    {
        /**enter here a value for the next parameter **/
    }
    rc=SQLParamData(hstmt,&prgbvalue);

}
}

```

---

## Retrieving data with bindings

Before retrieving columns of a result set, the program should associate them with program variables. This operation is named *binding columns* and is done by calling the `SQLBindCol()` function. The binding can be done before the execution of the statement, if the number and type of columns is known. If this is not the case, `SQLNumResultCols()` and `SQLColAttribute()` can be used to get the required information, but they must be called after the `SQLPrepare()` or the `SQLExecDirect()` function.

When a query is executed, a cursor is created by ODBC. `SQLFetch()` is used to go forward into the cursor and to transfer data to the program variables.

The arguments of `SQLBindCol()` are:

- ▶ The statement handle
- ▶ The number of the column (columns are ordered sequentially from left to right, beginning with 1)
- ▶ The C data type of the column
- ▶ A pointer to a variable where DB2 ODBC must put the column data
- ▶ The size of the variable
- ▶ A pointer to a value indicating the actual length of the column data

The outputs that point to the data value and its actual length are called *deferred* outputs, since they are not available after the binding but only after the fetch. Fetching rows can be done with the same binding. The binding can therefore only be executed once for each column.

While transferring data to variables, `SQLFetch()` does the conversion to the C data type provided in `SQLBindCol()`.

Example 4-13 illustrates the 2 functions `SQLBindCol()` and `SQLFetch()`. After the last row, `SQLFetch()` returns `SQL_NO_DATA_FOUND`.

For performance reasons, the variable which receives the column and the variable which receives its length are placed contiguously in memory, in a C structure. Thanks to this technique, ODBC fetches the 2 values in one operation.

### Example 4-13 Retrieving data with bindings

---

```

SQLCHAR          stmtin[47]=[SELECT FIRSTNME FROM EMP WHERE WORKDEPT='E11'];
struct
{
    SQLINTEGER     firstnme_len;
    char           firstnme_var[14];
} col_firstnme;
SQLINTEGER        numrows;

```

```
rc=SQLExecDirect(hstmt,stmtin,SQL_NTS);

rc=SQLBindCol(hstmt, 1, SQL_C_CHAR, &col_firstnme.firstnme_var, 14,
              &col_firstnme.firstnme_len);

numrows=0;
while ((rc=SQLFetch(hstmt)) != SQL_NO_DATA_FOUND)
{
    numrows++ ;
    printf("output #i : %s\n",numrows,col_firstnme.firstnme_var,);
}
printf("num of fetched rows %i\n",numrows);
```

---

## Retrieving several result sets

When the executed statement returns several result sets (for example if the parameters are bound to an array of variables, or if the statement called a stored procedure), the function `SQLMoreResults()` must be used to retrieve all the result sets sequentially. When a set is entirely fetched, the function closes the associated cursor and opens a cursor for the next set. This function is illustrated in Example 4-14 and Example 4-15.

## Retrieving result sets in an array

To retrieve columns in an array of variables, two statement attributes must be set before the first fetch:

- ▶ The `SQL_ATTR_ROWSET_SIZE` attribute must be set to the number of elements in the array.
- ▶ The value of `SQL_ATTR_BIND_TYPE` depends on the type of binding:
  - If columns are bound to separate arrays, the attribute must have the value `SQL_BIND_BY_COLUMN`. This technique is known as *column-wise binding*.
  - If columns are bound to an array of structures in which one element represent a whole row, this attribute must contain the length of the structure. This technique is known as *row-wise binding*.

When these two attributes are set, columns can be bound and rows must be fetched in the array with `SQLExtendedFetch()`. If the array is too small, several calls to `SQLExtendedFetch()` are needed to fetch all rows. This function returns in its fourth argument, the number of rows it has fetched.

Note that `SQLExtendedFetch()` is deprecated. `SQLFetchScroll()` should be used instead, but this function is not supported on DB2 ODBC for z/OS.

Example 4-14 and Example 4-15 are follow-ons of Example 4-8 and illustrate the *column-wise binding* and the *row-wise binding* techniques to fetch data in arrays. This also illustrates the use of `SQLMoreResults()` to retrieve multiple sets. Example 4-15 shows how to use `SQLFreeStmt()` to unbind parameters.

### Example 4-14 Retrieving result sets in an array (column-wise binding)

---

```
char          firstnme_var[4][13];
SQLINTEGER    firstnme_len[4];
SQLINTEGER    numrows, numset, i ;
SQLUINTEGER   pcrows;

rc=SQLSetStmtAttr(hstmt, SQL_ATTR_ROWSET_SIZE, (void *) 4, 0);
rc=SQLSetStmtAttr(hstmt, SQL_ATTR_BIND_TYPE,SQL_BIND_BY_COLUMN,0);
```

```

rc=SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) firstnme_var, 13,
               &firstnme_len[0]);

/** retrieve all rows for all set */
numset = 0 ;
do
{
    numset++;
    printf("number of the result set: %i\n", numset);
    numrows=0;
    numset=0;
    rc=SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrows, NULL);
    while (rc != SQL_NO_DATA_FOUND)
    {
        numset++;
        for (i = 0 ; i < pcrows ; i++)
        {
            numrows++;
            printf("output set #%i : %s\n", numset,
                  firstnme_var[i]);
        }
        rc=SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrows, NULL);
    }
    printf("num of fetched rows %i\n", numrows);
}
while ( SQLMoreResults(hstmt) == SQL_SUCCESS);

```

---

*Example 4-15 Retrieving result sets in an array (row-wise binding)*

---

```

struct {
    SQLINTEGER    firstnme_L_str;    /* for row-wise retrieval*/
    SQLCHAR       firstnme_str[13]; /* for row-wise retrieval*/
} R[4];
SQLINTEGER       numrows, numset, i ;
SQLUINTEGER      pcrows;

rc=SQLFreeStmt(hstmt,SQL_UNBIND);

rc=SQLExecDirect(hstmt,stmtin,SQL_NTS);

rc=SQLSetStmtAttr(hstmt, SQL_ATTR_ROWSET_SIZE, (void *) 4, 0);
rc=SQLSetStmtAttr(hstmt, SQL_ATTR_BIND_TYPE,(void *) (sizeof(R)/4),0);
rc=SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER) R[0].firstnme_str,
               13, &R[0].firstnme_L_str);

/** retrieve all rows for all set */
numset = 0 ;
do
{
    numset++;
    printf("number of the result set: %i\n", numset);
    numrows=0;
    numset=0;
    rc=SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrows, NULL);
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc );
}

```



```

while (rc != SQL_NO_DATA_FOUND)
{
    numset++;
    for (i = 0 ; i < pcrows ; i++)
    {
        numrows++;
        printf("output set #%i : %s\n",numset,
            R[i].firstme_str,R[i]);
    }
    rc=SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &pcrows, NULL);
}
printf("num of fetched rows %i\n",numrows);
}
while ( SQLMoreResults(hstmt) == SQL_SUCCESS);

```

---

## Retrieving data without bindings

It is possible to retrieve data without column bindings via the `SQLGetData()` function. This function also allows to retrieve data in small pieces, which could be an advantage to receive long data. `SQLGetData()` must be used in conjunction with `SQLFetch()` or `SQLExtendedFetch()` with a rowset size=1. Example 4-16 illustrates this function.

*Example 4-16 Retrieving data without bindings*

```

SQLCHAR          stmt[80]="SELECT FIRSTME FROM DSN8710.EMP WHERE \
                                WORKDEPT='A00'";

char              get_exe[2] ;
char              concat_exe[12] ;
char              firstme_exe[13] ;

while ((rc=SQLFetch(hstmt)) != SQL_NO_DATA_FOUND)
{
    strcpy((char *) concat_exe,"");
    rc=SQLGetData(hstmt, 1, SQL_C_CHAR, get_exe, 2, NULL);
    for ( i=0 ; (i<12 && rc != SQL_NO_DATA_FOUND); i++)
    {
        concat_exe[i]= get_exe[0];
        rc=SQLGetData(hstmt, 1, SQL_C_CHAR, get_exe, 2, NULL);
    }
    strncpy( firstme_exe, concat_exe, 12);
    printf("Firstname=%s\n",firstme_exe);
}

```

---

## Positioned updates and deletes

ODBC declares and opens cursors for the developer, but accepts positioned updates or deletes. For this purpose, statements are provided to set/get the cursor name. However different statement handles must be used to fetch the cursor and to run the positional update or delete.

Example 4-17 shows the use of `SQLSetCursorName()` and `SQLGetCursorName()` to manage names of cursor, and illustrates how positioned updates and deletes can be executed. Note that many tools do not use this technique, but use a selective statement instead, where the selection is done on a primary key identifying the row to update or delete. This is the reason why many tools do not accept delete and update operations on a table that does not have a primary key.

#### Example 4-17 Use of positioned updates and deletes

```
char          empno_pos[7];
SQLINTEGER    empno_L_pos;
char          cursor_nme[20] ;
WORD          cbCursor;
SQLCHAR       stmtpos[399];
SQLHSTMT      hstmt,hstmtpos;
SQLINTEGER    comp ;
SQLCHAR       stmtin[54]=["SELECT EMPNO FROM EMP WHERE WORKDEPT='A00' FOR UPDATE"];

rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt);
rc=SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmtpos);

rc=SQLSetCursorName(hstmt,(SQLCHAR *) "POSUPD", SQL_NTS);

rc=SQLExecDirect(hstmt,stmtin,SQL_NTS);

rc=SQLBindCol(hstmt, 1, SQL_C_CHAR, empno_pos, 7, &empno_L_pos);

rc=SQLGetCursorName(hstmt, (SQLCHAR *) cursor_nme, 20,
                    &cbCursor);

strcpy((char *) stmtpos, "UPDATE DSN8710.EMP SET SALARY = 9999999.99 \
WHERE CURRENT OF CURSOR ");
strncat((char *) stmtpos, cursor_nme,20);

while ((rc=SQLFetch(hstmt)) != SQL_NO_DATA_FOUND)
{
    comp = strcmp(empno_pos,"000099");
    if (comp == 0)
    { rc=SQLExecDirect(hstmtpos,stmtpos,SQL_NTS);
      if (rc == SQL_SUCCESS) {printf("successful positioned update\n");} }
}

rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmtpos);
```

### 4.5.7 Programming step 6: Terminating the ODBC application

In this section we describe ways to terminate the ODBC application.

#### Ending transactions

Normally, a new transaction is implicitly started when the first SQL statement is executed on the current data source. But the end of the transaction depends on the commit mode used by DB2 ODBC:

- ▶ In auto-commit mode, every SQL statement is considered as a transaction, and is automatically committed at the end of its execution, or when the cursor is closed, for query statements.
- ▶ In manual-commit mode, transactions (normally) begin at the first access to a data source, and must be explicitly ended by a call to `SQLEndTran()`, to rollback or commit. The value `SQL_ROLLBACK` or `SQL_COMMIT` is passed as argument to this function. `ROLLBACK` and `COMMIT` are the only 2 SQL statements that are not directly passed as string arguments.

The developer must consider two points:

- ▶ The influence of `CONNECTTYPE` and `MULTICONTEXT` keywords on the start of a transaction and the default commit mode (see 4.6.4, “Connections supported by DB2 ODBC on z/OS or OS/390” on page 82 for more information).
- ▶ The auto-commit mode can be expensive in terms of flows across the network.

Note that after the end of a transaction:

- ▶ Prepared SQL statements are still available for new `SQLExecute()` calls. It doesn't mean that the prepared statement is cached. Actually, the statement string is still associated with the handle, and ODBC is able to implicitly call an `SQLPrepare()` if the statement is re-executed. If DB2 ODBC is informed that DB2 keeps the prepared statement after a commit, it can avoid the submission of a `PREPARE` (see 4.7.3, “Using features of DB2 on z/OS” on page 85).
- ▶ Statement handles are still available to begin another transaction.
- ▶ Cursor names, bound parameters, and column bindings also survive transactions.
- ▶ Cursor positions are maintained after a commit if the hold option is set in the statement attribute (see 4.7.1, “ODBC usage and programming hints” on page 84 for more details).
- ▶ Cursor positions are closed after a rollback.

A handle is passed as an argument of `SQLEndTran()`. It can be a connection handle or an environment handle. In this latter case, DB2 ODBC terminates all transactions on all connections related to the environment.

Ending a transaction is as simple as shown in the following statements:

```
rc=SQLEndTran(SQL_HANDLE_DBC,hdbc,SQL_COMMIT);  
or  
rc=SQLEndTran(SQL_HANDLE_DBC,hdbc,SQL_ROLLBACK);
```

## Disconnecting from databases and freeing handles

Example 4-18 below shows how to disconnect from a database, and how to free allocated connection or environment handles. There is a sequence to follow when freeing handles and disconnecting from databases:

1. The application must first free all statement handles. Freeing a statement handle frees all resources allocated to the statement.
2. Then the application must disconnect from the database. Disconnecting from a database drops all statements that are not freed on the connection.
3. When a disconnection is done from a database, the application is allowed to free the related connection handle.
4. If all connection handles are freed, the application can free the environment handle.

Note that a connection or statement handle that is not freed can be reused to request another connection or submit another statement.

Just like the function to allocate handles, the function to free handles is the same for all types of handles.

*Example 4-18 Disconnecting from databases and freeing handles*

---

```
rc=SQLFreeHandle(SQL_HANDLE_STMT, hstmt);  
rc=SQLDisconnect(hdbc);  
rc=SQLFreeHandle(SQL_HANDLE_DBC, hdbc);  
rc=SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

---

## 4.5.8 Programming step 7: Retrieving diagnostics

Example 4-19 shows how to use the SQLGetDiagRec() and SQLGetSQLCA() function to display error information about the execution of a statement.

- ▶ SQLGetDiagRec() is used to read fields of a diagnostic record provided by ODBC.
- ▶ SQLGetSQLCA() is available with DB2 ODBC, to read the DB2 SQLCA.

*Example 4-19 Retrieving diagnostics*

---

```
SQLCHAR          buffer[SQL_MAX_MESSAGE_LENGTH+1];
SQLCHAR          sqlstate[SQL_SQLSTATE_SIZE+1] ;
SQLINTEGER       sqlcode ;
SQLSMALLINT      length, i;
struct sqlca     sqlca;

i = 1 ;
while ( SQLGetDiagRec( htype, hstmt, i, sqlstate, &sqlcode, buffer,
                      SQL_MAX_MESSAGE_LENGTH + 1, &length ) == SQL_SUCCESS )
{
    printf("error --- SQLSTATE: %s\n", sqlstate ) ;
    printf("error --- Native Error Code: %ld\n", sqlcode ) ;
    printf("error --- buffer: %s \n", buffer ) ;
    i++ ;
}

rc = SQLGetSQLCA(henv, hdbc, hstmt, &sqlca);
printf("error --- printing sqlca\n");
printf("error --- SQLCAID .... %s",sqlca.sqlcaid);
printf("error --- SQLCABC .... %d",sqlca.sqlcabc);
printf("error --- SQLCODE .... %d",sqlca.sqlcode);
printf("error --- QLERRML ... %d",sqlca.sqlerrml);
printf("error --- QLERRMC ... %s",sqlca.sqlerrmc);
printf("error --- QLERRP ... %s",sqlca.sqlerrp);
for (i = 0; i < 6; i++)
    {printf("error --- QLERRD%d ... %d",i+1,sqlca.sqlerrd[i]);}
for (i = 0; i < 10; i++)
    {printf("error --- SQLWARN%d ... %c",i,sqlca.sqlwarn[i]);}
printf("error --- SQLWARNA ... %c",sqlca.sqlwarn[10]);
printf("error --- SQLSTATE ... %s",sqlca.sqlstate);
```

---

To retrieve error information related to a connection or to the environment, replace the statement handle variable by a connection handle or by the environment handle in the SQLGetDiagRec() call.

## 4.5.9 Mapping between embedded statements and ODBC functions

Any SQL statement that can be dynamically prepared in an embedded program can be executed either by SQLExecDirect() or by SQLPrepare() and SQLExecute() (except for COMMIT and ROLLBACK, as explained in “Ending transactions” on page 76).

The embedded SQL statements that cannot be dynamically prepared are listed in Table 4-1 and mapped to their ODBC equivalent (when available).

Table 4-1 Mapping between embedded non-dynamic statements and ODBC functions

Purpose	Embedded SQL statement	ODBC equivalent
Stored procedure management	CALL	Although this statement cannot be dynamic in an embedded SQL program, it can be specified in <code>SQLExecDirect()</code> , or in <code>SQLPrepare()</code> and <code>SQLExecute()</code> functions. Procedure parameters can be described and bound via the usual ODBC functions.
	DESCRIBE PROCEDURE, DESCRIBE CURSOR	Each result set from a stored procedures is described by the usual ODBC functions. <code>SQLMoreResults()</code> must be used to browse subsequent result sets sequentially.
Cursor management	DECLARE CURSOR (this statement is not executable)	No need and no equivalent for such statements in ODBC environments. ODBC assigns a cursor and opens it automatically when the query is executed. Functions <code>SQLSetCursorName()</code> and <code>SQLGetCursorName()</code> can be used if the name of the cursor is needed (for example, in a positioned update).
	OPEN	
	FETCH	<code>SQLFetch()</code> , <code>SQLExtendedFetch()</code> , or <code>SQLGetData()</code>
	CLOSE	Either <code>SQLCloseCursor()</code> or <code>SQLFreeStmt()</code> with option <code>SQL_CLOSE</code> . <code>SQLFreeHandle()</code> also closes cursors associated with the handle to be freed.
Remote connection	CONNECT	<code>SQLConnect()</code> or <code>SQLDriverConnect()</code>
	RELEASE	No ODBC equivalent.
	SET CONNECTION	<code>SQLSetConnection()</code> , which is a DB2 ODBC and DB2 CLI feature, to be used when the application mixes embedded SQL and ODBC function calls.
Handling dynamic SQL	PREPARE	<code>SQLPrepare()</code>
	DESCRIBE INPUT	<code>SQLNumParams()</code> to retrieve the number of parameters. <code>SQLDescribeParam()</code> to describe a parameter. <code>SQLBindParameter()</code> must be used to associate a value with a parameter.
	DESCRIBE	<code>SQLNumResultCols()</code> to find out the number of retrieved columns, and <code>SQLDescribeCol()</code> or <code>SQLColAttribute()</code> to describe these columns. <code>SQLBindCol()</code> must be used to associate a retrieved value with a program variable.
	EXECUTE	<code>SQLExecute()</code>
	EXECUTE IMMEDIATE	<code>SQLExecDirect()</code> . Unlike the <code>EXECUTE IMMEDIATE</code> statement, <code>SQLExecDirect()</code> allows the direct execution of a <code>SELECT</code> statement.
Singleton select	SELECT INTO	No equivalent for such a statement in ODBC environments. <code>SQLBindCol()</code> can be used to bind a column to a program variable.

Purpose	Embedded SQL statement	ODBC equivalent
Other	BEGIN DECLARE SECTION (this statement is not executable)	No ODBC equivalent.
	END DECLARE SECTION (this statement is not executable)	No ODBC equivalent.
	DECLARE STATEMENT	No ODBC equivalent.
	DECLARE TABLE	No ODBC equivalent.
	INCLUDE (this statement is not executable)	No ODBC equivalent, but the SQLCA structure can be declared in the program, and the DB2 ODBC or DB2 CLI function SQLGetSQLCA() can be called to read the SQLCA.
	SET host variable	No ODBC equivalent.
	SET CURRENT APPLICATION ENCODING SCHEME	No ODBC equivalent.
	SET CURRENT PACKAGESET	No ODBC equivalent. The CURRENTPACKAGESET KEYWORD can be used on NT.
	VALUES INTO	No ODBC equivalent.
	WHENEVER (this statement is not executable)	No ODBC equivalent.

## 4.6 Specific features of DB2 ODBC

In this section we discuss features that are specific to the DB2 for z/OS ODBC implementation.

### 4.6.1 DB2 ODBC installation

DB2 ODBC must be installed on every target DB2, as a set of packages. There is a set of specific packages for each type of client platform needing to access the DB2 server through ODBC. The bind is valid for all applications, but specific binds can be done for specific applications.

A DB2 ODBC plan must also be bound on the local DB2 on z/OS if local ODBC access is to be used.

With DB2 ODBC on z/OS, the PLANNAME keyword must be specified to identify the ODBC plan. The COLLECTIONID keyword can be specified to identify the collection identifier to be used.

(With DB2 ODBC on NT, the CURRENTPACKAGESET keyword can be used to identify the collection.)

For more information about installing and binding ODBC, refer to chapter 4 of *DB2 Universal Database for OS/390 and z/OS ODBC Guide and Reference*, SC26-9941, and *DB2 Universal Database Call Level Interface Guide and Reference*, SC09-2950.

## 4.6.2 Overview of non-standard features

DB2 ODBC also provides the following non-standard features, on z/OS and NT:

- ▶ The possibility to mix embedded SQL and ODBC calls in the same program. However, the connection and transaction management must be performed completely either by ODBC or by embedded SQL. If embedded SQL is used to connect and to commit/rollback, the DB2 ODBC functions must use a null connection (a null connection is created by invoking `SQLConnect()` with the data source, userid and password arguments pointing to `NULL`).
- ▶ You must also avoid the use of the 2 interfaces in the processing of one statement. For example, opening a cursor in embedded SQL and calling the `SQLFetch()` function to retrieve the rows is not allowed.
- ▶ A set of functions to set or get attributes of environment handles (for example, the type of connection).
- ▶ The possibility to have detailed diagnostic information from the `SQLCA` structure.
- ▶ Specific facilities to improve performance, as explained in 4.7, “Performance considerations for ODBC applications” on page 84.
- ▶ Support of LOBs and LOB locators.
- ▶ Support of Open Group specification for the `SQLSTATEs`, which can differ from the ODBC specification.
- ▶ Some DB2 specific keywords and extension to attributes of handles, to exploit DB2 features, like the support of coordinated transactions, described in 4.6.4, “Connections supported by DB2 ODBC on z/OS or OS/390” on page 82.

## 4.6.3 Differences between the DB2 ODBC driver on z/OS and NT

Unlike the DB2 ODBC driver on NT, DB2 ODBC on z/OS does not support the following standard possibilities:

- ▶ Manipulation of descriptors.
- ▶ Asynchronous SQL.
- ▶ Compound SQL.
- ▶ Scrollable cursors.
- ▶ Multiple environment handles. The request to create a new environment handle if one already exists, returns the existing handle.
- ▶ Interactive data source connection, like the ability to prompt the user for connection information during the connection process.
- ▶ The driver manager. As a consequence, DB2 ODBC applications running on z/OS don't benefit from several functions provided by the driver manager such as for instance:
  - Connection pooling
  - Functions to retrieve information about data sources

We must also emphasize the following characteristics:

- ▶ DB2 ODBC on z/OS does not support all IBM extensions implemented in the DB2 UDB engine on NT, like the datalinks functions.

- ▶ There are more keywords and attributes available for DB2 ODBC on NT.
- ▶ DB2 ODBC on z/OS needs to be connected to a local DB2. Data sources must be defined in the SYSIBM.LOCATIONS table of the local catalog. However, the initialization file is still needed to define keywords associated to data sources.
- ▶ Functions not supported by DB2 ODBC on z/OS could possibly be used in an application with DB2 ODBC on NT, and connected to DB2 on z/OS. This is for example the case for:
  - Scrollable cursors (which are actually supported by DB2 on z/OS but not by DB2 ODBC on z/OS)
  - Non-atomic compound SQL
- ▶ The level of ODBC supported on z/OS is not the same as the level supported on NT.

For more detailed information about supported functions and keywords on z/OS and NT, see *DB2 Universal Database for OS/390 and z/OS ODBC Guide and Reference*, SC26-9941, and *DB2 Universal Database Call Level Interface Guide and Reference*, SC09-2950.

#### 4.6.4 Connections supported by DB2 ODBC on z/OS or OS/390

The way the connections are managed depends on the value of two keywords or attributes:

- |                      |   |
|----------------------|---|
| <b>CONNECTTYPE</b>   | <p>This option is an IBM extension, specified as a keyword in the initialization file, or as an attribute of the environment or connection handle. It is used to specify the type of DB2 connection.</p> <p>With type 1 connection (or with the corresponding attribute set to SQL_CONCURRENT_TRANS), the application can connect to only one database server at a time, and process remote units of work.</p> <p>With a type 2 connection (or with the corresponding attribute set to SQL_COORDINATED_TRANS), the application can connect to any number of database servers, and is able to process distributed units of work. All connections within the application must have the same connect type.</p> |
| <b>MULTICONTTEXT</b> | <p>This keyword is an IBM extension, which can take the value 1 (true) or 0 (false) to specify whether a context must be created for each DB2 ODBC connection handle. A context defines the scope of a connection, and is the DB2 ODBC equivalent of a DB2 thread. This applies to all DB2 connections within the application. The use of MULTICONTTEXT=1 requires the use of RRSF and OS/390 Version 2 Release 5 or higher.</p>  |

Next we describe the impact of these keywords. A summary is given in Table 4-2.

- ▶ When MULTICONTTEXT is false and the connection type is 1, each DB2 ODBC connection represents one transaction, as required by ODBC. But unlike ODBC, only one physical connection is active at any given time. If the current connection is done on the data source X and the program submits a statement on a data source Y, then all work done on X will be committed and the connection on X will be physically closed. It will be re-established at the next submission of a statement on X. Consequently, DB2 ODBC cannot maintain 2 cursors concurrently open at 2 data sources.
- ▶ When MULTICONTTEXT is false and the connection type is 2, the connections are not independent, and the ODBC connection model does not apply. All connections belong to the same coordinated transaction, which is therefore a distributed unit of work. This is an improvement to ODBC, which does not support distributed transactions.
- ▶ When MULTICONTTEXT is true, the application can have multiple physical connections. If the connection type is 1, the ODBC connection model can be fully implemented: indeed, DB2 ODBC creates a context (and a DB2 thread) for each connection. We can therefore have multiple physical connections, each being totally independent.



- ▶ Enabling multiple contexts for connection type 2 is not allowed.
- ▶ Note that for coordinated transactions, the default commit mode is manual, unlike concurrent transactions. This default can be changed by invoking the `SQLSetConnectAttr()` function.
- ▶ Note also that when using DB2 ODBC on z/OS' UNIX System Services, the application can have multiple independent connections under multiple threads. To benefit from this possibility, multiple contexts must be enabled, and the DB2 ODBC calls must be *threadsafe* (a routine is threadsafe when it can be called from multiple threads without unwanted interaction between the threads). See chapter 6 of *DB2 Universal Database for OS/390 and z/OS Version 7 ODBC Guide and Reference*, SC26-9941 for information about multithreaded applications.

Table 4-2 Impact of *MULTICONTTEXT* and *CONNECTTYPE* in the z/OS environment

MULTI-CONTEXT	CONNECT-TYPE	Can the application have several physical connections at a time (with outstanding transaction on each)?	Can the application have independent ODBC transactions?	default commit mode
0	2	Y	N	Manual
0	1	N	Y	Automatic
1 <sup>a</sup>	1	Y	Y	Automatic

a. if multicontext =1, DB2 ODBC ignores value of 2 for connecttype and rejects any attempt to set this value by a function.

#### 4.6.5 Connections supported by DB2 ODBC on UNIX or NT

The way connections are managed depends on the value of three keywords or attributes:

##### **CONNECTTYPE**

This keyword is equivalent to the *CONNECTTYPE* keyword on z/OS or OS/390 platforms.

##### **DISABLEMULTITHREAD**

This keyword is an IBM extension, equivalent to the *MULTICONTTEXT* keyword on z/OS or OS/390 platforms. When *DISABLEMULTITHREAD* is true, there is only one context for all connections made by the application. When *DISABLEMULTITHREAD* is false, a context is allocated for each connection handle. The value of the keyword applies for all connections asked by the application.

##### **SYNCPOINT**

This option is an IBM extension, available as a keyword or an attribute of the connection handle. Specify "1" for single phase commit, or "2" for 2-phase commit. All connections within the application must use the same commit protocol.

By default, *CONNECTTYPE* = 1 and *DISABLEMULTITHREAD* = false, which means that each connection is independent and runs its own unit of work.

The value of *SYNCPOINT* is ignored if *CONNECTTYPE* = 1. If *CONNECTTYPE* = 2 and *SYNCPOINT* = 1, the transaction must not update more than one database. Any attempt to update a second database is rejected.

When using `CONNECTTYPE = 2` and `SYNCPOINT = 2`, distributed transactions with updates on multiple databases can be implemented. More precisely, the following values must be set to enable multisite updates in one transaction:

- ▶ `CONNECTTYPE = 2`
- ▶ `DISABLEMULTITHREAD = 1`
- ▶ `SYNCPOINT = 2`

## 4.7 Performance considerations for ODBC applications

Several techniques are available to improve or control performance of DB2 ODBC dynamic SQL.

### 4.7.1 ODBC usage and programming hints

When developing or installing ODBC applications, consider the following:

- ▶ Use bindings instead of `SQLGetData()` which is very expensive from the performance point of view because it moves the data twice: once from the server to the driver, and again from the driver to the `SQLGetData()` area. Limit the usage of this function to the retrieval of large data in pieces.
- ▶ Increase the efficiency of transferring character data by placing the length argument and the value argument contiguously in memory (for example in a C data structure).
- ▶ Use optimized programming techniques such as arrays of parameters and arrays of retrieved columns whenever possible. Note that fetching arrays when using the DB2 CLI driver on NT or UNIX is very efficient, because of the strong relationship between the driver and DB2 Connect.
- ▶ Pay special attention to the following attributes:
  - `SQL_ATTR_AUTOCOMMIT`  
By default, the value of this connection attribute is on (except for coordinated transactions). Application packages could rely on this value, but if you are writing your own application, you should consider to turn off this attribute (and to manage commits in your programs), since you can save network flows. However, do not forget that not committing transactions can hold resources on the server.
  - `SQL_ATTR_CURSOR_HOLD`  
Set the value of this statement attribute to off (it is set to on by default), in order to save resources associated with statement handles. Be careful if you are using an application package, as it could rely on another value for this parameter.
  - `SQL_ATTR_NOSCAN`  
When this statement attribute is off (which is the default), DB2 ODBC scans the statement to search for escape clauses. Turn this parameter on, if your programs do not use escape clauses for DB2 specific extensions.
  - `SQL_ATTR_MAXCONN`  
This connection attribute can be used to limit the number of concurrent connections allowed for a DB2 ODBC application.
- ▶ Make also use of the following keywords or attributes, available with DB2 ODBC on NT or UNIX:
  - The `DEFERREDPREPARE` and `EARLYCLOSE` keywords, as well as the `SQL_ATTR_CLOSEOPEN` statement attribute can be used to save network traffic, as described in 14.3, “Minimizing message flows across the network” on page 231.

- STATICCAPFILE, STATICLOGFILE, STATICMODE and STATICPACKAGE are a set of keywords to allow the use of static SQL instead of dynamic SQL. DB2 ODBC is able to capture dynamic SQL into a file. A command can then be launched to bind the most frequently used statements, and the resulting static SQL plan can be used at run time when needed. More information about this feature (known as the “Static Profiling Feature”) can be found on the site:

<http://www.ibm.com/software/data/db2/udb/staticcli/>

For more information about these keywords, see chapter 4 of *DB2 Universal Database for OS/390 and z/OS Version 7 ODBC Guide and Reference*, SC26-9941 and *DB2 Universal Database Call Level Interface Guide and Reference* SC09-2950.

## 4.7.2 Improving access to the catalog

ODBC applications heavily access the DB2 catalog, to prepare statements, or to describe parameters and result sets. These accesses are even more critical in tools generating SQL statements interactively, as they submit catalog queries to provide the end user with information about available tables and columns. DB2 ODBC offers several possibilities to improve performance of catalog queries:

- The SYSSHEMA keyword can be used to specify the schema name of an alternative catalog. If this schema groups views, limiting the number of rows on the actual DB2 catalog, or if it groups a set of user-defined tables containing a subset of the catalog, the performance of catalog queries can be improved. This keyword should be used in conjunction with:
  - The TABLETYPE keyword, which defines the type of tables (such as ‘view’, ‘alias’, ‘synonym’, ‘table’, ‘system table’) the application is supposed to work with. This parameter can reduce the time to query catalog table information.
  - The SCHEMALIST keyword, which defines a list of schemas to reduce the number of tables presented to the end user.
  - The DBNAME keyword, which defines the name of the database on which the application runs, reduces the number of table involved, and reduce the time to query the catalog for information
- The CLISHEMA keyword can be used to specify the schema name of the DB2 ODBC shadow catalog. The DB2 ODBC catalog is a copy of a subset of the DB2 catalog, containing only columns needed by ODBC operations and containing pre-joined tables, specifically indexed for ODBC.

Several DB2 ODBC catalog can be defined, each for a particular user group. With DB2 ODBC on NT, CLISHEMA can also be specified as an attribute of the connection handle.

More information about the DB2 ODBC catalog can be found in “*ODBC Catalog Function performance*”, found at

<http://www.ibm.com/software/data/db2/os390/odbc/>

## 4.7.3 Using features of DB2 on z/OS

DB2 itself offers features to improve or control ODBC performance:

- **DB2 dynamic statement caching**  
All ODBC applications can benefit from the DB2 global dynamic statement cache. If the application runs on NT or UNIX, it can also benefit from the DB2 local statement cache, by using the KEEPDYNAMIC keyword (available in DB2 CLI V7.2, fixpack 3). When this keyword is set to YES, the driver is informed that DB2 uses the KEEPDYNAMIC bind

option, and avoids to send implicit SQLPrepare() calls, when the driver is asked for the execution of a previously prepared statement after a commit. Note that the driver sends the SQLPrepare() if he is explicitly asked to do so.

However, binding the ODBC packages with KEEP\_DYNAMIC(YES) can consume active threads (threads cannot become inactive when they touched a KEEP\_DYNAMIC package) and must be used selectively. It is recommended to bind ODBC packages with this option only for critical applications. Bind the ODBC packages in a specific collection and use the CURRENT\_PACKAGESET keyword to point to this name for those critical applications.

See Chapter 8, “Dynamic statement caching” on page 141 for information about the different types of dynamic statement caching and the usage of the KEEP\_DYNAMIC bind option.

► **Automatic close of cursor**

Under certain circumstances, DB2 for z/OS can automatically close cursors at the end of the result set (SQLCODE 100). In this case, it informs DB2 Connect, which can then avoid flowing a message to the server (to close the cursor) when the application explicitly asks for the close. Detailed information about the automatic close of cursors, and other DB2 performance features on z/OS can be found in *DB2 Universal Database for OS/390 and z/OS Administration Guide*, SC26-9931.

► **Stored procedures**

Stored procedures containing static SQL statements can improve performance. Using stored procedures within an ODBC program does not require any special technique. The CALL statement can be prepared and executed in ODBC applications. Parameters can be bound and result sets can be fetched with standard ODBC functions. Note that DB2 supports stored procedures called by an ODBC application, as well as procedures using ODBC.

More considerations about stored procedures can be found in chapter 6 of *DB2 Universal Database for OS/390 and z/OS ODBC Guide and Reference*, SC26-9941 and chapter 3 of *DB2 Universal Database Call Level Interface Guide and Reference*, SC09-2950. See also the Redbook *Cross-Platform DB2 Stored Procedures: Building and Debugging*, SG24-5485.

## Developing JDBC applications

In this chapter we introduce the JDBC environment and explain how to develop JDBC based applications. The following topics are covered:

- ▶ JDBC, an alternative to ODBC
- ▶ JDBC concepts and facilities
- ▶ The JDBC API
- ▶ JDBC programming samples
- ▶ Specific features of DB2 JDBC on z/OS
- ▶ SQLJ, an alternative to JDBC

## 5.1 Introduction

JDBC is the Java API to execute dynamic SQL statements against relational databases. Its goal is to make Java programs independent from vendor-specific DBMS implementations. It is based on the Call-Level Interface standard developed by the Open Group (formerly X/Open) standards group, and it is therefore very similar to the ODBC API.

We do not intend here to explain the full Java environment or give details on Java programming techniques. We want to give an overview of the JDBC concepts and browse through the similarities and differences with ODBC. We also introduce the JDBC API, and give some programming samples showing how Java programs can use dynamic SQL. Finally, we list some important JDBC features specific to DB2 on z/OS, and give a very short introduction to SQLJ, the Java interface to execute static SQL.

Note that JDBC is a component of the core Java API standard. Although it often seen as the acronym for “Java Database Connectivity”, it is actually a trademark of Sun Microsystems Inc.

## 5.2 JDBC, an alternative to ODBC

The JDBC API provides Java programs with the same kind of functions (and advantages) that C or C++ programs have with ODBC. It provides many similar functions like:

- ▶ Cursors must not be explicitly declared or opened.
- ▶ It is possible to do the following:
  - Pass SQL statements as non-interpreted string arguments of functions
  - Retrieve very large values in pieces
  - Send very large parameters in pieces
  - Manage multiple result sets
  - Query the DBMS catalog through standardized methods
  - Do bulk updates
- ▶ Support is provided for:
  - Stored procedure calls
  - Standard isolation levels (JDBC does not support the versioning isolation level, which is not part of the SQL92 standard)
  - Escape clauses
  - Connection pooling
  - Scrollable result sets (JDBC also support updatable result sets)

There are also a few differences between JDBC and ODBC. The main ones are the following:

- ▶ The supporting language is of course different (C or C++ for ODBC, Java for JDBC).
- ▶ ODBC provides support for asynchronous SQL. Because of the multithreading capabilities of the Java environment, JDBC does not have to explicitly support it. From the main thread, the developer can create separate threads for each asynchronous activity.
- ▶ Unlike ODBC:
  - JDBC does not support bulk operations to fetch retrieved rows into an array.
  - JDBC supports distributed transactions with two phase-commit.
  - JDBC supports advanced data types defined in SQL99, such as, BLOB and CLOB.
  - JDBC API does not deal with pointers or parameter and column bindings.

A complete list of features supported by JDBC can be found in the API specifications:

<http://java.sun.com/products/jdbc/>

The advantages and drawbacks of JDBC are also closely related to the underlying Java environment. The main drawback is the performance of Java, which can partly be addressed by SQLJ (see explanations about SQLJ in “SQLJ, an alternative to JDBC” on page 111).

On the other hand, when accessing databases from a Java application, programs, and developers can benefit from several advantages intrinsic to Java, such as these:

- ▶ A high level programming language (developers do not have to deal with low level functions such as memory management).
- ▶ The network awareness native to Java.
- ▶ The hardware and operating system independence native to Java. Combined with this independence, the portability offered by JDBC across DBMSs is a step forward in the “write-once, run-anywhere” philosophy of Java.
- ▶ Integration with Java e-business extensions such as Java Transaction Processing and the JavaBeans model for object component building.
- ▶ Object interoperability in the Java environment.
- ▶ Multi threaded nature of the Java environment.
- ▶ Access to Java services such as the *Java Naming and Directory Interface* (JNDI).

## 5.3 JDBC concepts and facilities

This section provides an overview of the environment, concepts, and facilities used by JDBC applications.

### 5.3.1 The driver manager, the drivers and the data sources

Figure 5-1 gives a general overview of the JDBC environments.

Like ODBC applications, JDBC applications can interface with a driver manager, which in turn interfaces with database specific drivers. On z/Os or OS/390, the driver is attached to the local DB2 via CAF or RRSAF. On other platforms, an additional layer to manage connections (for example, DB2 Connect, not represented in the figure) may or may not be necessary, depending on the driver type. The different driver types are described in 5.3.4, “JDBC driver types” on page 92.

Note that JDBC can be used to manipulate any kind of tabular data sources, even data not stored in a relational database (for example a segment in an IMS database).

Drivers must be supplied by the DBMS vendor, and must implement the API specifications. The driver is loaded by one of the following methods:

- ▶ When the driver manager initializes, it loads each driver listed by name in a specific property in the system properties file.
- ▶ Or the program itself explicitly loads a driver by invocation of a specific method.

The driver manager itself can support multiple database drivers simultaneously. Its role is to load or unload the drivers, and to connect Java applications to the correct driver.

The *JDBC properties file* contains keywords used by JDBC at run time. In the Java world, these keywords are called *properties*. In the case of DB2 on z/OS, the name of this file is provided by the environment variable DB2SQLJPROPERTIES.

Note that JDBC can be used in normal Java applications, as well as in a *servlet*, a *Java Server Page* (JSP), or an *applet*. In this latter case, the applet is downloaded in the HTML page with a JDBC client, and executes within the Java enabled browser. When a request must be sent to the DBMS, the JDBC client submits a command on a separate network connection to a JDBC applet server, which does calls to the DBMS and sends the results back to the client. Note that this implementation needs a so-called “Type 3 driver” that is not available for DB2 on z/OS. See 5.3.4, “JDBC driver types” on page 92 for more information about driver types.

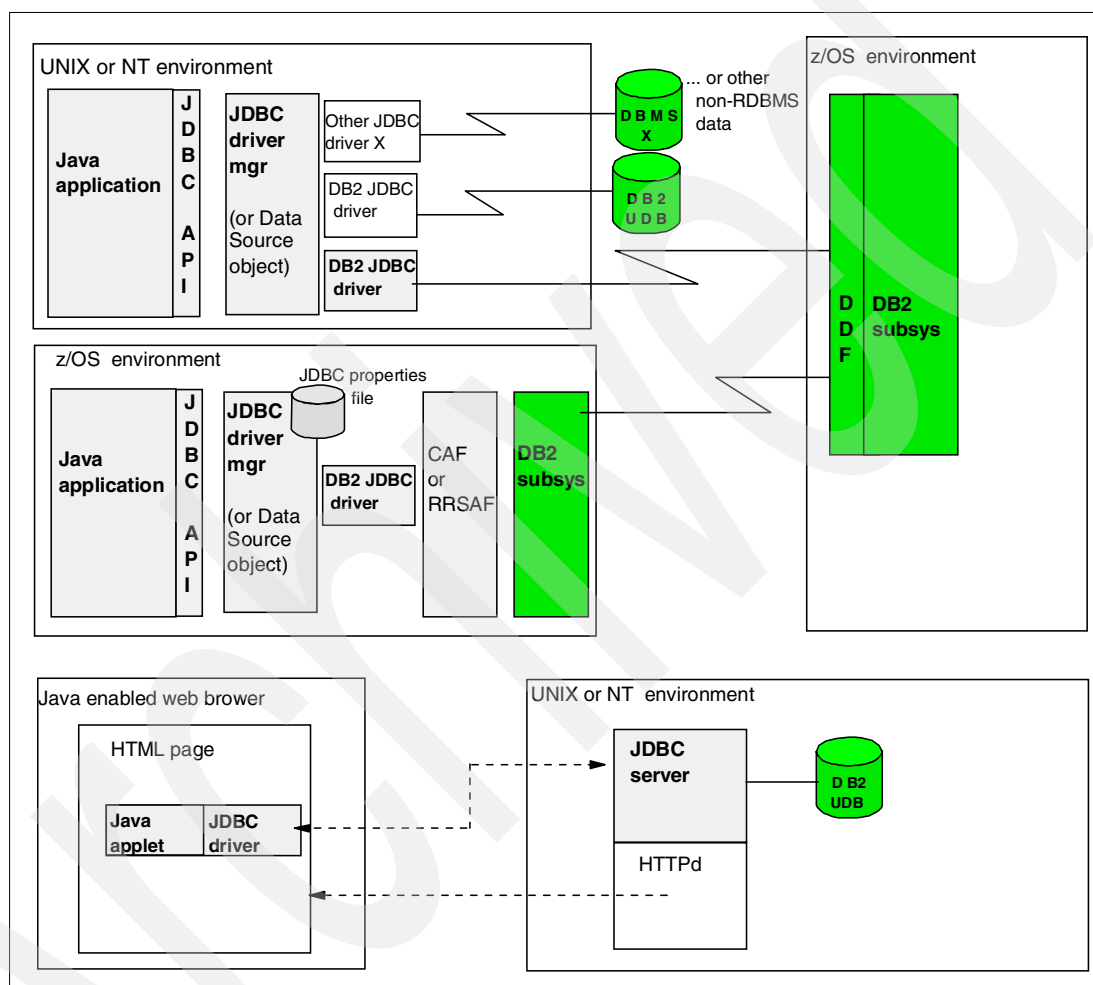


Figure 5-1 The JDBC environment

## 5.3.2 The JDBC connections

As explained here, there are two ways that a Java program itself can request a connection to a target source.

### Connection via the DriverManager class

In this case, the program gives a *Uniform Resource Locator* (URL) to the driver manager, to identify the target source it wants to connect to. This URL must have the following standardized structure: `jdbc:<subprotocol>:<subname>`.



The *subprotocol* allows to identify the type of connection under JDBC, and the *subname* identifies the target source. On OS/390 and z/OS, the subprotocol must have the value “db2os390” or “db2os390sqlj”. The subname must either:

- ▶ Identify a local DB2, by taking the value of its location name.
- ▶ Identify a remote DB2 by taking the value of a location name defined in SYSIBM.SYSLOCATIONS.
- ▶ Be unspecified. In this case, the local DB2 will be used.  
This format is a DB2 extension for OS/390 and z/OS.

When the driver manager receives an URL with a request to create a connection, it gives the URL to each driver, which must examine the *subprotocol* and check if he can support it. If many drivers are available to support the subprotocol, JDBC can choose one based on a specific Java property containing a list of drivers to be considered.

When this connection technique is used, the driver must be registered within the driver manager. The registration is done by the driver itself when it is loaded.

When using the DriverManager class, you must avoid hard-coding the URL in the program, since a change in the target source specification (for example, a change in the target name or the protocol) implies a change in the program.

### Connection via a DataSource object

JDBC provides a DataSource interface, which must be implemented in the driver. Using this implementation, it is possible to create a DataSource object and to set several properties representing a target source (for example, the location of the database server, the name of the database, and the identification of the network protocol) and determining the correct JDBC driver. Note that in the case of DB2 on z/OS:

- ▶ The name of the database is actually the location name of the local DB2 or a location name defined in the SYSIBM.SYSLOCATIONS table.
- ▶ The plan name of the JDBC driver must be specified (by default, DSNJDBC is used).

A DataSource object can be given a name in the *Java Naming and Directory Interface* (JNDI) and this name can be referred by java program to get the DataSource object. Once a program obtains the DataSource object, it can call a method to receive a connection to the data source.

When accessed via the DataSource API, a JDBC implementation should not automatically register itself with the DriverManager.

By using this connection technique, properties of the DataSource object can be changed (for example, to reflect a change in the supported protocol), without impacting the application. The programs are therefore made independent from a specific JDBC driver, or JDBC URL.

Chapter 1 of *DB2 Universal Database for OS/390 and z/OS Version 7 Programming Guide and Reference for Java*, SC26-9932 provides an example, showing how JNDI and the DataSource object can be used to obtain a connection.

The DataSource facility provides an alternative to the JDBC DriverManager, essentially duplicating all of the driver manager’s useful functionality. Although, both mechanisms may be used by the same application if desired, we encourage developers to regard the DriverManager as a “legacy feature” of the JDBC API. Applications should use the DataSource API whenever possible. The DriverManager, Driver, and DriverPropertyInfo interfaces may be deprecated in the future.

### 5.3.3 JDBC classes and interfaces

As explained in more detail in 5.4, “The JDBC API” on page 93, the JDBC API is a set of abstract Java classes and interfaces, which must be implemented by the JDBC driver. The API allows the programmer to create connection objects, each representing a connection to a database server. Once a connection object is created, statement objects can be obtained to execute SQL statements.

Each class has a constructor to instantiate objects, and a set of methods to manage them. JDBC defines many specialized methods, instead of a few general purpose methods. This is a design choice, to make the methods easier to understand.

These are the four main classes or interfaces:

- ▶ The DriverManager class works with the driver interface to manage the set of available drivers.
- ▶ The Connection interface allows to obtain a Connection object which can then be used to create a statement object. Methods can be used to obtain information about the connection (for example, the connection capabilities, supported grammar, database meta-information).
- ▶ The Statement interface allows to manipulate Statement objects, which are used as containers to execute SQL statements and retrieve the possible result sets.
- ▶ The ResultSet interface allows to manipulate a ResultSet object which contains the rows of a resulting table.

### 5.3.4 JDBC driver types

There are many possible driver implementations, which are categorized in four types, all illustrated in Figure 5-2.

#### **Type 1 driver - bridge driver**

A *Type 1 JDBC driver* implements the JDBC API as a mapping to another API. Type 1 drivers usually map JDBC to ODBC. The driver provider can easily implement this bridge, considering the similarities between the two APIs.

#### **Type 2 driver - native API driver**

A *Type 2 JDBC driver* is partly written in Java, and is implemented on top of a DBMS specific function library. JDBC calls are translated into calls to DBMS specific functions or modules. Type 2 drivers give better performance than Java-only drivers (Type 3 and Type 4 drivers described in the following sections).

#### **Type 3 driver- net driver**

A *Type 3 JDBC driver* is entirely written in Java, and communicates with a middleware server via a database-independent protocol. The middleware server communicates the request to the DBMS.

#### **Type 4 driver - native Java driver**

A *Type 4 JDBC driver* is entirely written in Java. It implements directly the DBMS protocol (such as for example, DRDA), and provides the application with a direct connection to the data source.

Type 1 drivers are less performant than the other types and are being used less and less frequently for that reason.

The portability of Type 1 and Type 2 drivers themselves can be limited, as they rely on a native library and are not completely written in Java. Type 3 and Type 4 drivers benefit from the Java portability and offer the possibility of automatic installations by download.

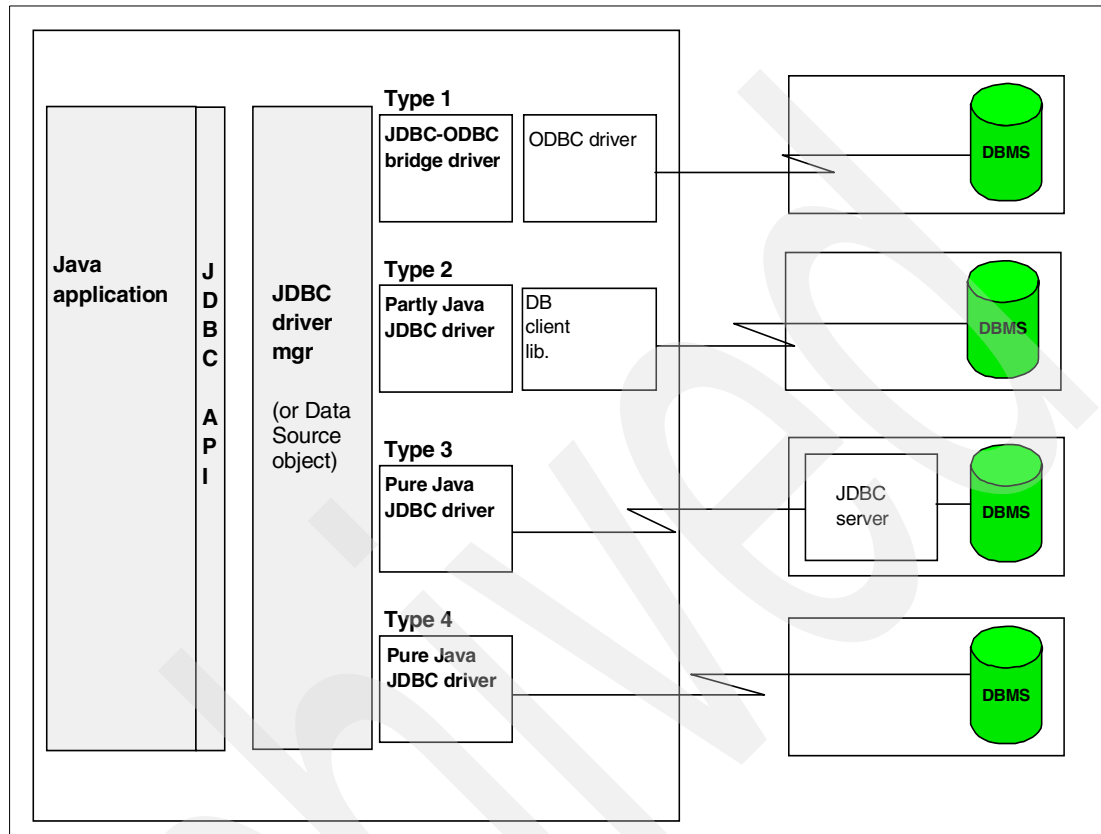


Figure 5-2 The JDBC driver types

## 5.4 The JDBC API

The JDBC API provides all capabilities of dynamic SQL, including the possibility to describe parameter markers (in JDBC version 3.0; at the time of writing, version 3.0 is not implemented by the DB2 JDBC driver) and to describe unknown retrieved columns. It can therefore be used to develop ad-hoc query environments.

Following the specifications of the API, JDBC drivers provide object classes and methods to manage database access. These classes allow to instantiate JAVA objects that can then be used to exploit SQL. The four main classes are briefly introduced in 5.3, "JDBC concepts and facilities" on page 89. In this section, we provide a more detailed overview of the available classes and methods.

The classification in the following sections is neither official nor exhaustive. It gives a summary of the main possibilities provided by the API, and is only used in support of the samples in section 5.5, "JDBC programming samples" on page 100.

## 5.4.1 Managing connection and statement objects

The main classes and methods described here are illustrated in Figure 5-3, and in the samples provided in 5.5.1, “Initializing driver and database connection” on page 100.

Before a connection can be provided, the associated driver must be loaded. If not loaded by the driver manager, it can be loaded by invocation of the `Class.forName()` method.

The connection itself is requested through the `DriverManager.getConnection()` or `DataSource.getConnection()` method, possibly with the specification of a userid and a password. The differences between a connection through the `DriverManager` or the `DataSource` class is explained in 5.3, “JDBC concepts and facilities” on page 89.

A transaction starts when the first SQL statement is executed. By default, new JDBC connections are in “auto-commit” mode, which means that each statement is a separate transaction. To change this, the auto-commit mode must be disabled by invocation of the `setAutoCommit()` method, with the argument “false”. The transaction is terminated via the `Connection.commit()` or `Connection.rollback()` method, and the connection object is closed by the `close()` method.

Other methods can be called on the connection object, for example to set the transaction isolation level or to set and release `SAVEPOINTS`.

Note that asking for a commit, a rollback, a `SAVEPOINT`, or enabling the auto-commit mode cannot be requested on connections participating in a distributed transaction. The boundaries of a distributed transaction are controlled by an external transaction manager (RRS when running on z/OS).

Once a connection object exists, a `Statement` object can be created. There are three types of `Statement` objects:

- ▶ The `Statement` object, created by the `Connection.createStatement()` method.
- ▶ The `PreparedStatement` object, created by the `Connection.prepareStatement()` method. The `PreparedStatement` class is a subclass of `Statement`, and has specific methods to deal with input parameters. Such an object can be prepared once and then executed several times.
- ▶ The `CallableStatement` object, created by the `Connection.prepareCall()` method. The `CallableStatement` class is a subclass of `PreparedStatement` and must be used to call a stored procedure. It provides specific methods to deal with output parameter values returned from the stored procedure

A `Statement`, `PreparedStatement`, or `CallableStatement` object is closed by the `close()` method called on the object. All open statement objects are implicitly closed when the connection that created them is closed.

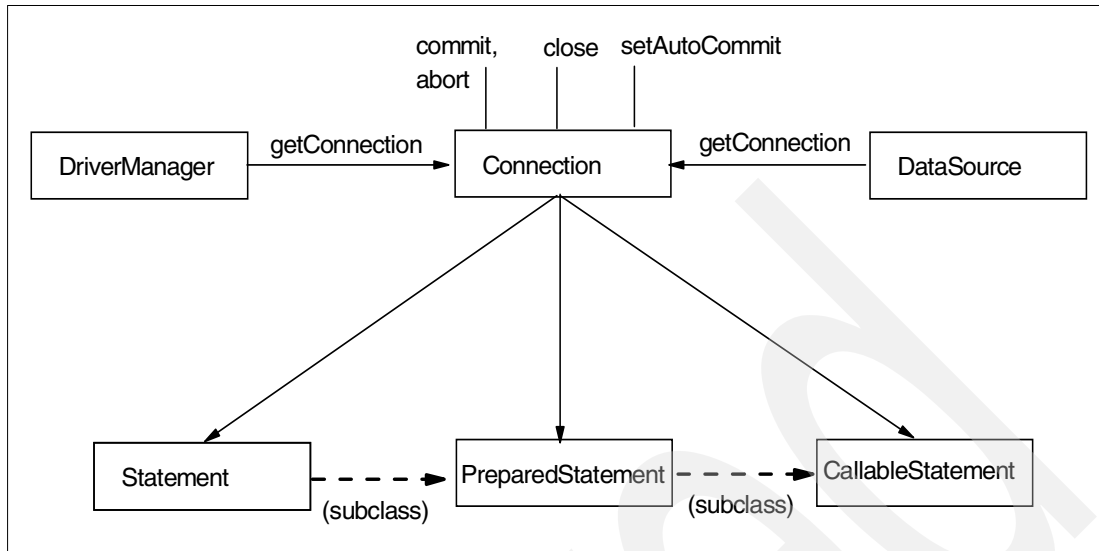


Figure 5-3 JDBC classes and methods to manage Connection and Statement objects

## 5.4.2 Preparing the execution

The main classes and methods described here are illustrated in Figure 5-4 on page 96, and in the sample provided in 5.5.2, “Preparing and executing SQL statements” on page 101.

A statement is actually prepared when it is created by invoking the `prepareStatement()` method or `prepareCall()` method. Both `PreparedStatement` and `CallableStatement` objects represent prepared statements. Once a statement is prepared:

- It can be executed several times with possibly different values for the parameters.
- The possible parameter markers (JDBC V3.0), and the possible result sets can be described.

The methods briefly described in the following sections apply to `PreparedStatement` objects. `CallableStatement` objects have similar methods which are not illustrated in this book.

### Describing parameters

The method `PreparedStatement.getParameterMetaData()` can be used to return a `ParameterMetaData` object describing the parameter markers. Different methods can then be called on this object to retrieve the characteristics of the parameters (for example, `getParameterType()` to know the type of the parameter). Note however that this is a new feature of the JDBC V3.0, not supported by the current DB2 JDBC driver.

### Assigning a value to a parameter

The `PreparedStatement` interface provides the so-called `setXXX()` methods to assign a value to a parameter. There is one `setXXX()` method for each Java data type. For example, `setInt()` must be used to assign an integer value to a parameter. The JDBC driver must map it to the corresponding JDBC type (a JDBC type is one of the SQL types defined in `Java.sql.Types`). The `setObject()` method allows explicit conversions and can be used instead of `setXXX()`. The mapping used by `setXXX()` and the conversions authorized by `setObject()` are both described in tables in the JDBC API specification, which can be found at:

<http://java.sun.com/products/jdbc/>

The mapping table for `setXXX()` is also reproduced in Table 5-1 on page 102.

The setXXX methods have at least two arguments: one is the position of the parameter and the other is the value to be assigned. A special method, setNULL must be used to assign a NULL value to a parameter.

When setXXX is used, the previously assigned value is replaced by the new one, but the ClearParameters() method can be explicitly called to clear parameter values.

## Describing columns

The method PreparedStatement.GetMetaData() produces a ResultSetMetaData object which contains a record for each column to be retrieved. Specific methods applied on this object allow to know the number and characteristics of the columns (for example, getColumnType() to know the type of a column).

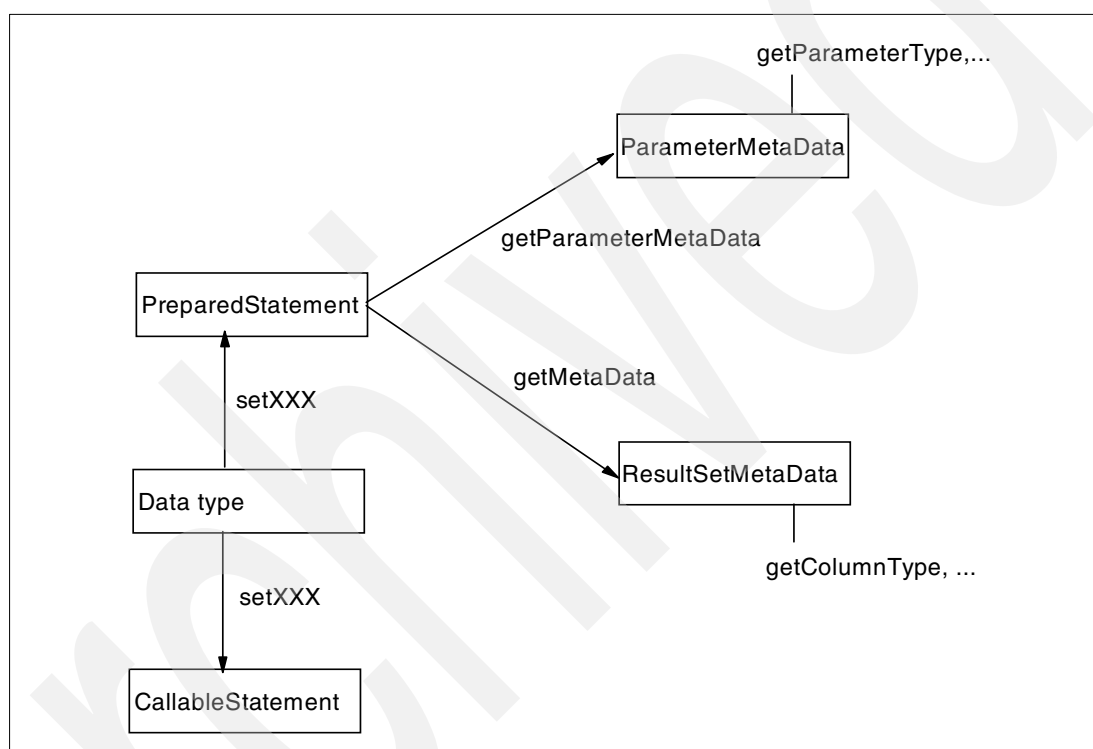


Figure 5-4 JDBC classes and methods to prepare the execution

## 5.4.3 Executing statements

The main classes and methods described here are illustrated in Figure 5-5, and in the sample provided in 5.5.2, “Preparing and executing SQL statements” on page 101.

The execution of an SQL statement is requested through the invocation of one of the following methods, on a Statement, PreparedStatement or CallableStatement object:

- ▶ The `executeQuery()` method, which must be used to execute a statement that is expected to return one set of data (for example, a SELECT statement). This method returns a `ResultSet` object representing the retrieved data.
- ▶ The `executeUpdate()` method, which must be used to execute a statement that updates the state of the database (for example, a DELETE, INSERT or UPDATE statement, or a DDL statement). This method returns the number of rows affected by the change, or 0 for DDL statements.

- The `execute()` method, which is used when the type of the SQL statement is not known or when it returns multiple result sets and/or multiple result counts. This method returns true if the first result is a `ResultSet` and false if it is an update count. Others methods must then be used to retrieve the `ResultSet` object or the update count.

Of course, the execution of an unprepared statement implicitly asks for a prepare before the execution.

Note that for a `Statement` object, the SQL statement string is passed when asking for the execution, and for a `PreparedStatement` or a `CallableStatement` object, it is passed at the creation of the object.

JDBC supports the “batch update” operation which consist in associating a set of update statements to a statement object via the `addBatch()` method, and then execute the whole set in one call by `executeBatch()`. This facility also allows to associate a parameter in a `PreparedStatement` or `CallableStatement` with a list of values. Note that JDBC does not provide a support for “batch select”.

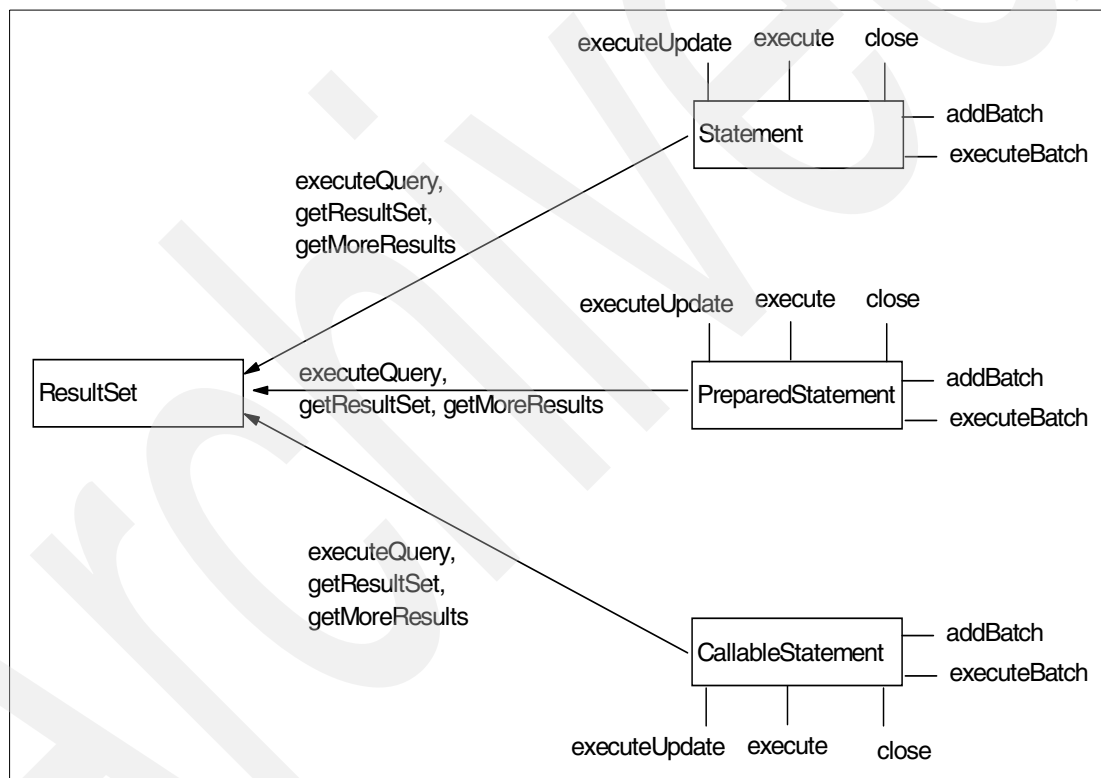


Figure 5-5 JDBC classes and methods to execute statements

#### 5.4.4 Managing execution results

The main classes and methods described here are illustrated in Figure 5-5, Figure 5-6, and in the sample provided in 5.5.2, “Preparing and executing SQL statements” on page 101.

A result set is characterized by:

- Its type, indicating whether the cursor is scrollable or not, and whether it is sensitive to changes done on the data source while it is open
- Its concurrency, indicating whether it can be updated with the `ResultSet` interface
- Its “holdability”, indicating whether it must be held after a commit

These characteristics are defined by passing parameters in the method used to create the statement object, or one of its subtypes.

By default, a result set is not scrollable and not updatable. The default “holdability” depends on the keyword DB2CURSORHOLD provided in JDBC properties file. A result set is closed by using the close() method.

Note that JDBC does not allow to return columns in an array.

## Describing and retrieving results

The getMetaData() method can be called on a ResultSet object to obtain the description of columns into a ResultSetMetaData object. Note that the possibility to describe result sets is available in JDBC 2.0, unlike the possibility to describe parameters, which needs JDBC 3.0.

The only method applicable to move a cursor on a non-scrollable result set is next(), which fetches rows sequentially, one by one.

The columns of a row are accessed (by name or order in the result set) via the so-called getXXX() methods. There is one getXXX() method for each JDBC type. For example, getInt() must be used to fetch an SQL INTEGER value into a variable. The recommended method for each JDBC type is described in a table in the JDBC API specification, which can be found at

<http://java.sun.com/products/jdbc/>

The table is also reproduced in Table 5-2 on page 103. The wasNULL() method can be used to check whether the last column retrieved has a NULL value.

## Positioned updates and deletes

If the result set is updatable, positioned updates and deletes can be used, but like in ODBC programs, they must be used in another statement object. Since cursor names are managed by JDBC, methods must be provided to provide the program with the cursor name to be used in the positioned update or delete statement. The getCursorName() method on a statement object allows to get the cursor name, and the setCursorName() method can be used to set a cursor name.

## Retrieving multiple results

If the execute() is invoked on a statement object (or one of its subtype) because the result was not known, then three specific methods can be used to manage the results:

- ▶ If the execute() method returns true, then the first result is a ResultSet object. You can use getResultSet() on the statement object to obtain this object.
- ▶ If the execute() method returns false, then the first result is an update count. You can call getUpdateCount() on the statement object to obtain this count.
- ▶ The getMoreResults() method, when used on the statement object, allows to move to the next result. It returns true if the next object is a ResultSet, or false if it is an update count or if there are no more objects.



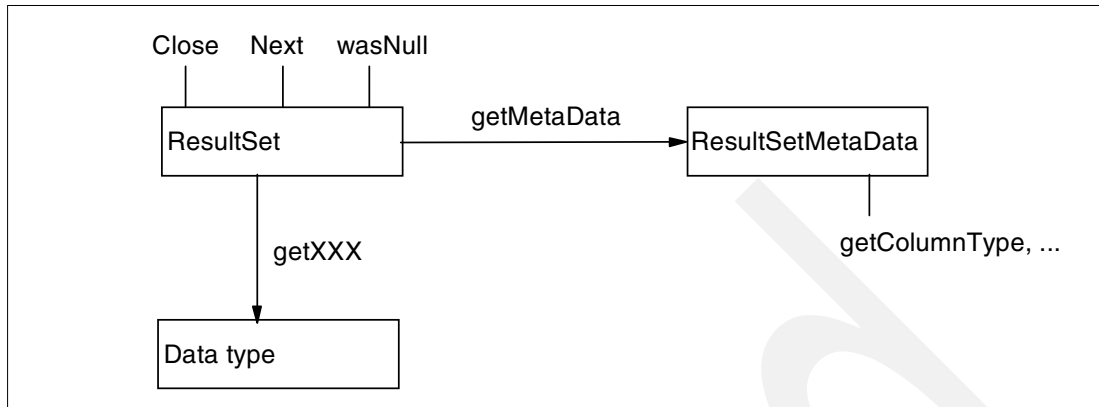


Figure 5-6 JDBC classes and methods to manage execution results

### 5.4.5 Retrieving diagnostics

The main classes and methods described here are illustrated in Figure 5-7.

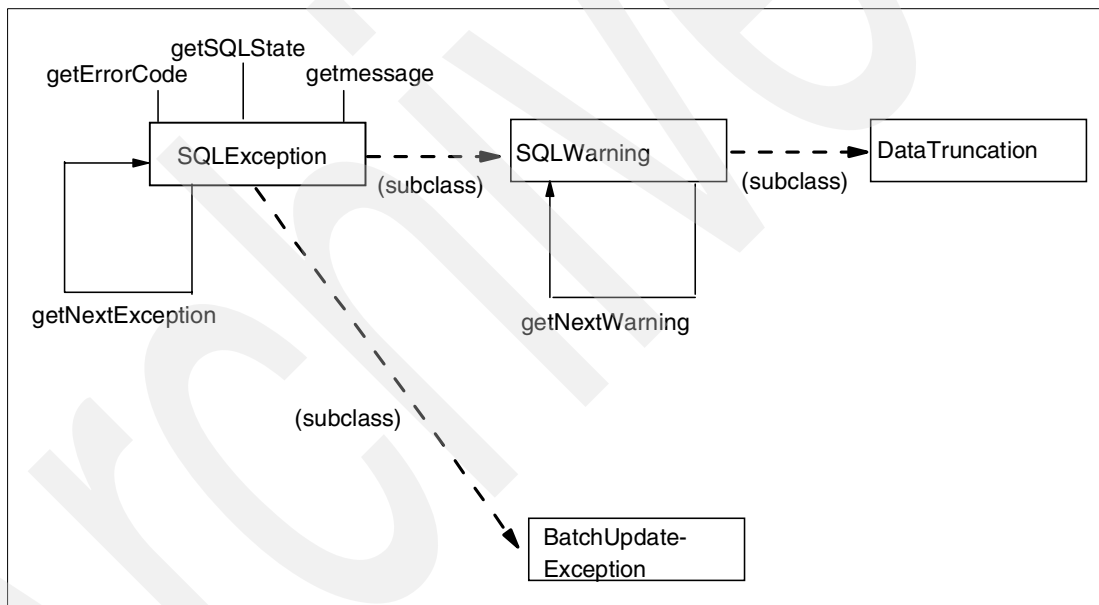


Figure 5-7 JDBC classes and methods to retrieve diagnostics

Different objects can be returned to the program to analyze errors and warnings:

- ▶ An `SQLException` object is thrown to the program when an error occurs during an interaction with a data source. This object contains information such as an error code, an `SQLState`, and textual description that can be retrieved by methods `getErrorCode()`, `getSQLState()`, and `getMessage()`. If there is a chain of errors, the `getNextException()` method allows to retrieve the next exception.
- ▶ An `SQLWarning` object can be created when methods are invoked on a `Connection`, `Statement` (and its subtypes), or `ResultSet` object. This object is not automatically obtained. It is retrieved by calling the method `getWarnings()` on the appropriate object. Chains of `SQLWarning` can be retrieved via the `getNextWarning()` method.
- ▶ A `DataTruncation` object, which provides information when a data is truncated. When the truncation happens on a write to a data source, the `DataTruncation` is automatically thrown.

- A `BatchUpdateException` object, which provides error information during execution of a batch of update statements.

### 5.4.6 Retrieving information about a data source

The `getMetaData()` method can be called on a connection object to create a `DatabaseMetaData` object. Methods can then be invoked on this object to get information about the features supported by the target DBMS, or the descriptions of database objects defined in the catalog.

## 5.5 JDBC programming samples

As already mentioned, JDBC provides a set of classes and methods to access a database. Almost all the methods within the JDBC classes map to one or more DB2 CLI APIs. Whereas DB2 CLI or ODBC APIs are procedural, JDBC classes are object-oriented, in this way better fitting in the JAVA programming language. The handling of the JDBC methods is less complicated than the handling of the complex ODBC calls.

The logical flow of a simple JDBC program is typically the following:

1. Load the driver
2. Connect to the database or DB2 for z/OS subsystem
3. Create statements
4. Execute statements
5. Retrieve the results
6. Close the statements
7. Close the connection

In the following sections we give some examples, how to use JDBC in a Java program to implement the steps described above. These examples are kept simple on purpose. They do not fully exploit the JDBC interface nor the Java language. For an advanced usage of JDBC and Java please refer to books and manuals listed in “Related publications” on page 255.

### 5.5.1 Initializing driver and database connection

Before you can use any JDBC class you must import the package for the JDBC interface:

```
import java.sql.*;
```

Then you must load the specific driver for your database by calling the method `Class.forName`. This method explicitly loads the *driver* class as shown in Example 5-1. The driver creates an instance of the class and then registers it with the `DriverManager` class when it is loaded.

The `DriverManager` class is a management layer of JDBC, working between the user and the drivers.

*Example 5-1 Load of the DriverManager*

---

```
static
{
    try
    {
        Class.forName ("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
    }
    catch (Exception e)
    {
        System.out.println ("\n Error loading DB2 Driver...\n");
        System.out.println (e);
        System.exit(1);
    }
}
```

---

When the driver has been loaded and registered with the *DriverManager* class, it is available for establishing a connection with a database using the *DriverManager.getConnection()* method.

The Connection interface supports the connection to a specific database or a DB2 for z/OS subsystem. In Example 5-2 we execute the program providing a separate user ID and password for DB2. If not provided, the connection uses the authorization ID of the executing user.

*Example 5-2 Connecting to a DB2 subsystem*

---

```
Connection con = null;
// URL is jdbc:db2:dbname
String url = "jdbc:db2os390sqlj:DB2G";

if (argv.length == 0)
{
    // connect with default id/password
    con = DriverManager.getConnection(url);
}
else if (argv.length == 2)
{
    String userid = argv[0];
    String passwd = argv[1];
    // connect with user-provided username and password
    con = DriverManager.getConnection(url, userid, passwd);
}
else
{
    throw new Exception("\nUsage: java Dynamic [username password]\n");
}
```

---

It is recommended for application programs to turn off the autocommit of JDBC in the program, as shown here:

```
con.setAutoCommit(false);
```

Please remember that connections can also be obtained through a DataSource object as explained in 5.3.2, “The JDBC connections” on page 90.

## 5.5.2 Preparing and executing SQL statements

In this section we consider the preparation and execution of SQL statements.

### Creating, preparing and executing statements

After initializing driver and database connection, you can now access the database. You need a *Statement* Object associated with an active connection to prepare and execute an SQL statement. A *Statement Object* is used to send SQL statements to the database. You can use three classes of *Statement Objects*:

- |                                 |  |
|---------------------------------|--|
| <b><i>Statement</i></b>         | To execute a simple SQL statement with no parameters                 |
| <b><i>PreparedStatement</i></b> | To execute a prepared SQL statement with or without input parameters |
| <b><i>CallableStatement</i></b> | To execute a call to a stored procedure                              |

Example 5-3 shows how to create a *Statement* object:

*Example 5-3 Creating a Statement object*

---

```
// Create the Statement
Statement stmt = con.createStatement();
System.out.println("**** JDBC Statement Created");
```

---

After creating a *Statement* object, you must supply it with the text of the SQL statement and execute it, as shown in Example 5-6 on page 103.

Alternatively, the *PreparedStatement* interface allows you to prepare an SQL statement explicitly before executing it. It supports any SQL statement containing parameter markers as shown in Example 5-4.

**Example 5-4 Preparing a statement**

```
// Perform dynamic SQL SELECT using JDBC
PreparedStatement pstmt1 = con.prepareStatement(
"SELECT NAME FROM SYSIBM.SYSTABLES " +
"WHERE NAME <> ? " +
"ORDER BY 1");
```

How to create and use a *CallableStatement* object is shown in Example 5-10 on page 105.

If your statement text contains parameter markers, you have to supply the actual values using a so called *setXXX* method, before you can execute the statement as shown here:

```
pstmt1.setString(1, "STAFF");
```

Table 5-1 shows the data type conversion between Java and JDBC SQL data types performed by the different *setXXX* methods.

**Table 5-1 Conversion table for setXXX methods**

	T I N Y I N T	S M A L L I N T	I N T E G E R	B I G I N T	R E A L	F L O A T	D O U B L E	D E C I M A L	N U M E R I C	B I T	C H A R	V A R C H A R	L O N G V A R C H A R	B I N A R Y	L O N G B I N A R Y	L O N G V A R B I N A R Y	D A T E	T I M E	T I M E S T A M P
String	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
java.math.BigDecimal	x	x	x	x	x	x	x	x	x	x	x	x	x						
Boolean	x	x	x	x	x	x	x	x	x	x	x	x	x						
Integer	x	x	x	x	x	x	x	x	x	x	x	x	x						
Long	x	x	x	x	x	x	x	x	x	x	x	x	x						
Float	x	x	x	x	x	x	x	x	x	x	x	x	x						
Double	x	x	x	x	x	x	x	x	x	x	x	x	x						
byte[]														x	x	x			
java.sql.Date											x	x	x				x		x
java.sql.Time											x	x	x					x	
java.sql.Timestamp											x	x	x				x	x	x

As mentioned before, JDBC version 2 does not support the SQL DESCRIBE INPUT statement. The program has to know the input parameter(s). In JDBC Version 3 you can use the class *ParameterMetaData*. In Example 5-5 we show, how to describe input parameters with JDBC version 3, which is not yet supported by DB2 for z/OS.

*Example 5-5 Using ParameterMetaData to describe input parameters*

```
ParameterMetaData P = pstmt1.getParameterMetaData();
int colType = P.getParameterType(1);
System.out.println("ParType = " + colType);
```

To execute a SELECT statement, you use the *executeQuery()* method, as shown in Example 5-6. The executed query returns a result set.

*Example 5-6 Executing a query*

```
// Execute a Query and generate a ResultSet instance
// The Query is a Select from SYSIBM.SYSTABLES
ResultSet rs = stmt.executeQuery("SELECT NAME FROM SYSIBM.SYSTABLES");
System.out.println("**** JDBC Result Set Created");
```

A statement which modifies data has to use the *executeUpdate()* method as shown here. The number of rows involved is returned after execution.

```
int modCount = pstmt2.executeUpdate();
```

## Using result sets

The *ResultSet* interface provides access to the results of an executed query. The so-called *getXXX* methods are used to retrieve the values in the result set according to their data types. The Table 5-2 shows the possible methods. "X" indicates that the method is recommended for the JDBC type.

*Table 5-2 Common getXXX methods*

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	LONGBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getBytes	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						

getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x		X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

**Note:** With PTF UQ59527 for APAR PQ51847 (version 7 only), LOB support was added to the JDBC 2.0 driver for DB2 for z/OS and OS/390. The driver now supports `get/setBlob`, `get/setClob`, `get/setObject` as defined in the JDBC 2.0 specifications. In addition, DB2 provides additional `getXXX` and `setXXX` methods that are not specified in JDBC 2.0, to help you manipulate LOBs. For more details, see the README file that accompanies the aforementioned PTF.

The `next()` method moves the cursor to the next row in the result set. The cursor is initially positioned before the first row of the result. This makes it is very easy to read the retrieved data as shown in Example 5-7. (The statement has been prepared before; see Example 5-4 on page 102.)

*Example 5-7 Retrieving the result of a query*

```
ResultSet rs = pstmt1.executeQuery();
System.out.print("\n");
while( rs.next() )
{
    String tableName = rs.getString("NAME");
    System.out.println("Table = " + tableName);
};
```

If you do not know the columns and their attributes in your result, you can describe the columns of the result set using the class `ResultSetMetaData` as shown in Example 5-8:

*Example 5-8 Using `ResultSetMetaData` to describe a query result*

```
ResultSetMetaData M = rs.getMetaData();
int colCount = M.getColumnCount();
int colType;
String colName;
for (int i = 1; i <= colCount; i++)
{
    colType = M.getColumnType(i);
    colName = M.getColumnName(i);
    // here follows your coding using the results
}
```

## Positioned updates or deletes

For positioned updates or deletes you must use a cursor and create new *Statement* objects for the update or delete statements.

In Example 5-9 we prepare and execute a SELECT statement with a parameter marker. We associate the *PreparedStatement* to a named cursor for positioned update. For the UPDATE statement we create a second *PreparedStatement* object. Retrieving the data of the result set in a *while* loop, we execute the prepared UPDATE statement.

*Example 5-9 Using a cursor for positioned updates*

---

```
PreparedStatement pstmt1 = con.prepareStatement(
    "SELECT NAME, DEPT FROM PAOLR5.STAFF WHERE JOB = ? FOR UPDATE OF JOB");

// set cursor name for the positioned update statement
pstmt1.setCursorName("c1");
pstmt1.setString(1, "MANGR");
ResultSet rs = pstmt1.executeQuery();

PreparedStatement pstmt2 = con.prepareStatement(
    "UPDATE PAOLR5.STAFF SET JOB = ? WHERE CURRENT OF c1");
pstmt2.setString(1, "CLERK");

System.out.print("\n");
while( rs.next() )
{
    String NAME = rs.getString("NAME");
    short DEPT = rs.getShort("DEPT");
    System.out.println(NAME + " in dept. " + DEPT + " will be demoted to Clerk");

    pstmt2.executeUpdate();
}
};
```

---

## Calling stored procedures

In JDBC you use the *CallableStatement* class to call stored procedures. Generally, this method is used to execute statements which return more than one result set, more than one update count, or a combination of both. In Example 5-10 we set up a call string with procedure name and parameter markers. Then we prepare the statement creating the *CallableStatement* object, and supply the input parameter. The output parameters must be registered with their data types. The call is executed with the method *execute()*.

*Example 5-10 Calling a stored procedure*

---

```
// prepare the CALL statement for EXAMPLE
String procName = "EXAMPLE";
String sql = "CALL " + procName + "(?, ?, ?)";
CallableStatement callStmt = con.prepareCall(sql);

// set the input parameter
callStmt.setString (1, job);

// register the output parameter
callStmt.registerOutParameter (2, Types.DOUBLE);
callStmt.registerOutParameter (3, Types.INTEGER);

// call the stored procedure
System.out.println ("\\nCall stored procedure named " + procName);
callStmt.execute();
```

```
// retrieve output parameters
double outParm1 = callStmt.getDouble(2);
int outParm2 = callStmt.getInt(3);
```

---

The returned parameter values of the called stored procedure must be retrieved using the *getXXX* methods (refer to Table 5-2 for more information).

### Getting error information

The standard way of error handling in JAVA is to throw exceptions. In order to handle error conditions in exception handling, you can receive SQLCODE, SQLSTATE, or the SQL message as shown in Example 5-11.

*Example 5-11 Receiving SQLCODE, SQLSTATE and SQL message*

---

```
String[] SQLSTATE,
int[] SQLCODE,
String[] SQLMessage
try
{
    // your JAVA program code
}
catch (SQLException e)
{
    SQLSTATE[0] = e.getSQLState();
    SQLCODE[0] = e.getErrorCode();
    SQLMessage[0] = e.getMessage();
    // your exception handling
}
```

---

## 5.5.3 Transactions and termination

In this section we discuss transactions and their termination.

### Closing and terminating

Result sets which you do not use any longer should be closed to free the objects as shown here. Otherwise the JVM garbage collection cannot reclaim the objects.

```
rs.close();
```

Closing the result sets is not enough to release the underlying cursor resource. Therefore, the prepared statements should be closed as well like this:

```
pstmt1.close();
```

Before terminating your program, close the database connection as shown in Example 5-12:

*Example 5-12 Closing a database connection*

---

```
// Close the connection
con.close();
System.out.println("**** JDBC Disconnect from DB2 for z/OS.");
```

---

### Transaction processing

To control the transactions in your program, you can issue commits as shown in Example 5-13.



---

*Example 5-13 Commit*

---

```
{ // Commit the transaction
  con.commit();
  System.out.println("Transaction committed.");
}
```

---

To undo changes after an error, you can use the *rollback* method as shown in Example 5-14:

---

*Example 5-14 Rollback*

---

```
{ // Rollback the transaction
  System.out.println("\nRollback the transaction...");
  con.rollback();
  System.out.println("Rollback done.");
}
```

---

## 5.5.4 Multi-threading sample

In Example 5-15 we show sample code for a multi-threading program. The main class loads the database specific driver, checks the passed parameters, and starts the *SQLExecution* class with different department numbers as parameters. It waits for the termination of both threads, printing “.” to inform the user about the waiting status.

---

*Example 5-15 Multi-threading sample*

---

```
import java.io.*;
import java.lang.*;
import java.sql.*;

class UseThrds2 {

    static {
        try {
            // register the driver with DriverManager
            Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main (String[] argv) {
        try {
            int numInt = 0;

            // Set the department number
            if (argv.length == 0)
            {
                numInt = 10;
            }
            else if (argv.length == 1)
            {
                String num = argv[0];
                Integer temp = Integer.valueOf (num);
                numInt      = temp.intValue();
            }
            else
            {
                throw new Exception("\nUsage: java UseThrds [departmentNumber]\n");
            }

            Department deptmnt = new Department (numInt);
```

```

// create and start the thread
SQLExecution sqlexec = new SQLExecution (deptmnt);
sqlexec.start();
// indicate that thread is running along with the program
System.out.print ("\nRunning thread...");

numInt = 20;
Department deptmnt2 = new Department (numInt);
// create and start the thread
SQLExecution sqlexec2 = new SQLExecution (deptmnt2);
sqlexec2.start();
// indicate that thread is running along with the program
System.out.print ("\nRunning thread2...");

while (sqlexec.isAlive()) {
    System.out.print (".");
    for (int delay = 1; delay <= 500000; delay = delay + 1) {}
}
// print all information received from thread and stored in Department class
deptmnt.printDeptInfo();

while (sqlexec2.isAlive()) {
    System.out.print (".");
    for (int delay = 1; delay <= 500000; delay = delay + 1) {}
}
// print all information received from thread and stored in Department class
deptmnt2.printDeptInfo();

} catch (Exception e) {e.printStackTrace(); }
}
}

//
// SQLExecution
// - thread (runs independently from main program)
// - queries the org table for information about the deptnum chosen
//

class SQLExecution extends Thread {
    private Department department;

    public SQLExecution (Department dept) {
        department = dept;
    }

    public void run() {
        Connection con = null;
        try {
            // connect to the DB2 subsystem DB2G
            con = DriverManager.getConnection("jdbc:db2os390sqlj:DB2G");
            con.setAutoCommit(false);
            String deptName = new String("");
            String DIVISION = new String("");
            String LOCATION = new String("");

            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery ("SELECT DEPTNAME, DIVISION, LOCATION " +
                "FROM PAOLOR5.ORG WHERE DEPTNUMB = " + department.getDeptNum());
            if (rs.next()) {
                deptName = rs.getString (1);
            }
        }
    }
}

```

```

        DIVISION = rs.getString (2);
        LOCATION = rs.getString (3);
        // pass in results of query to object Department
        department.setDeptInfo (deptName, DIVISION, LOCATION);
    } else { department.setDeptInfo ("NO SUCH DEPT NUMBER", "", ""); }

    // disconnect from the server
    rs.close();
    con.close();
} catch (Exception e) { e.printStackTrace(); }
}

//
// Department
// - this class allows interaction and passing of information
//   between the main program and the thread so that information
//   gathered in the thread can be accessed (printed) from the
//   main program
//

class Department {
    private int deptNum;
    private String deptName;
    private String DIVISION;
    private String LOCATION;

    public Department (int num) {
        deptNum = num;
    }

    public int getDeptNum() {
        return deptNum;
    }

    public void setDeptInfo (String name, String dvison, String loc) {
        deptName = new String (name);
        DIVISION = new String (dvison);
        LOCATION = new String (loc);
    }

    public void printDeptInfo() {
        System.out.print ("\n\nDEPARTMENT #      : ");
        String deptNumString = String.valueOf (deptNum);
        System.out.println (deptNumString );

        System.out.print ("DEPARTMENT NAME : ");
        System.out.println (deptName );

        System.out.print ("DIVISION      : ");
        System.out.println (DIVISION );

        System.out.print ("LOCATION      : ");
        System.out.println (LOCATION );
    }
}

```

---

The `SQLExecution` class connects to the DB2 subsystem, creates a *Statement* object, concatenates the passed input parameter to the statement string and executes the *Statement* object. It retrieves the result of the query, passes it using the class `Department` to the main program, and closes result set and connection.

The main program prints the returned results of the terminated threads using class `printDeptInfo` and terminates.

## 5.6 Specific features of DB2 JDBC on z/OS

More detailed information about the topics presented in this section can be found in *DB2 Universal Database for OS/390 and z/OS Version 7 Programming Guide and Reference for Java*, SC26-9932; the *CCF Connectors and Database Connections Using WebSphere Advanced Edition*, SG24-5514; and *DB2 for OS/390 and z/OS Powering the World's e-business Solutions*, SG24-6257.

### 5.6.1 Properties

A set of properties specific to DB2 on z/OS or OS/390 can be defined in the JDBC properties file, for example:

- ▶ `DB2SQLJPLANNAME`, which represents the name of the plan associated with a JDBC or SQLJ application. By default, this property has the value `DSNJDBC`.
- ▶ `DB2SQLJSSID`, which gives the name of the DB2 subsystem the application must connect to. The default value is the name that was specified during installation of the local DB2.
- ▶ `DB2SQLJMULTICONTEXT`, which specifies if multiple contexts are supported. The default value is `YES` (see topic 5.6.2, “Connections supported by DB2 JDBC” on page 110 for more information).
- ▶ `DB2CURSORHOLD`, specifies whether or not the cursor is kept open after a commit. The default is `YES`.
- ▶ A set of properties to control connection and traces.

Other runtime parameters can optionally be set, such as a *cursor properties file*, to control the cursor names used by JDBC, and their hold option. More information about setting the JDBC runtime environment is given in chapter 6 of *DB2 Universal Database for OS/390 and z/OS Version 7 Programming Guide and Reference for Java*, SC26-9932.

### 5.6.2 Connections supported by DB2 JDBC

The way connections are managed depends on the `DB2SQLJMULTICONTEXT` property, which can be `NO`, to disable multiple contexts, or `YES` to enable multiple contexts.

A context is equivalent to a DB2 thread. If the possibility of multiple contexts is not enabled, the connection objects share the same context, meaning that only one connection object can have an active transaction at any time. Before using another connection, the program must issue a `COMMIT` or `ROLLBACK` on the connection having the current active transaction. The application can create multiple Java threads, but a connection object cannot be shared between those threads.

If the possibility of multiple contexts is enabled, each connection object has its own context, and its own DB2 thread. The Java program can then have multiple concurrent connections, each with its own active transaction. The application can create multiple Java threads, and the connection objects can possibly be shared between objects. To support multiple contexts, z/OS must be at least at Version 2 Release 5, and the attachment to DB2 must be RRSAF.

## 5.7 SQLJ, an alternative to JDBC

The SQLJ ANSI standard defines a set of clauses to include static SQL in Java programs. It is implemented as extensions to the core Java classes. SQLJ and JDBC are complementary and can be used in the same program. The standard specifies three components:

- ▶ **The database language:** This defines how to code SQLJ statements and specifies which statements are supported.
- ▶ **The SQLJ translator:** The SQLJ clauses are transformed by the translator into calls to the runtime environment. After translation, the Java program can be compiled normally. The translator produces *profile* files which contain information about the database schema that must be accessed. The profiles are used at execution time by the SQL runtime environment. The profiles must be customized if the use of vendor-specific features or optimization is needed. With DB2, the profiles can be used to create DBRMs that must be bound into DB2 packages to have a true static SQLJ implementation.
- ▶ **The runtime environment:** This is a thin layer of pure JAVA code that communicates with the DBMS. SQLJ programs can access any DBMS for which a SQLJ runtime implementation exists. By default, the SQLJ runtime performs accesses using JDBC. A SQLJ program can therefore access any DBMS for which a JDBC driver exists. SQLJ also relies on JDBC for tasks like connecting to databases and handling SQL errors. Like JDBC, SQLJ on z/OS supports DB2 attachments to CAF or RRSAF.

### 5.7.1 Preparing SQLJ programs

Figure 5-8 gives a general overview of preparation of SQLJ programs. For more information about SQLJ, refer to the following publications:

- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Programming Guide and Reference for Java*, SC26-9932
- ▶ *Java Programming Guide for z/OS*, SG24-5619
- ▶ *SQLJ Support in DB2 Universal Database Version 7.1*, found at:  
<http://www.ibm.com/software/data/db2/java/sqlj/>
- ▶ *SQLJ Availability in DB2 for z/OS*, found at:  
<http://www.ibm.com/software/data/db2/os390/sqlj.html>
- ▶ General information, found at:  
<http://www.sqlj.org/>

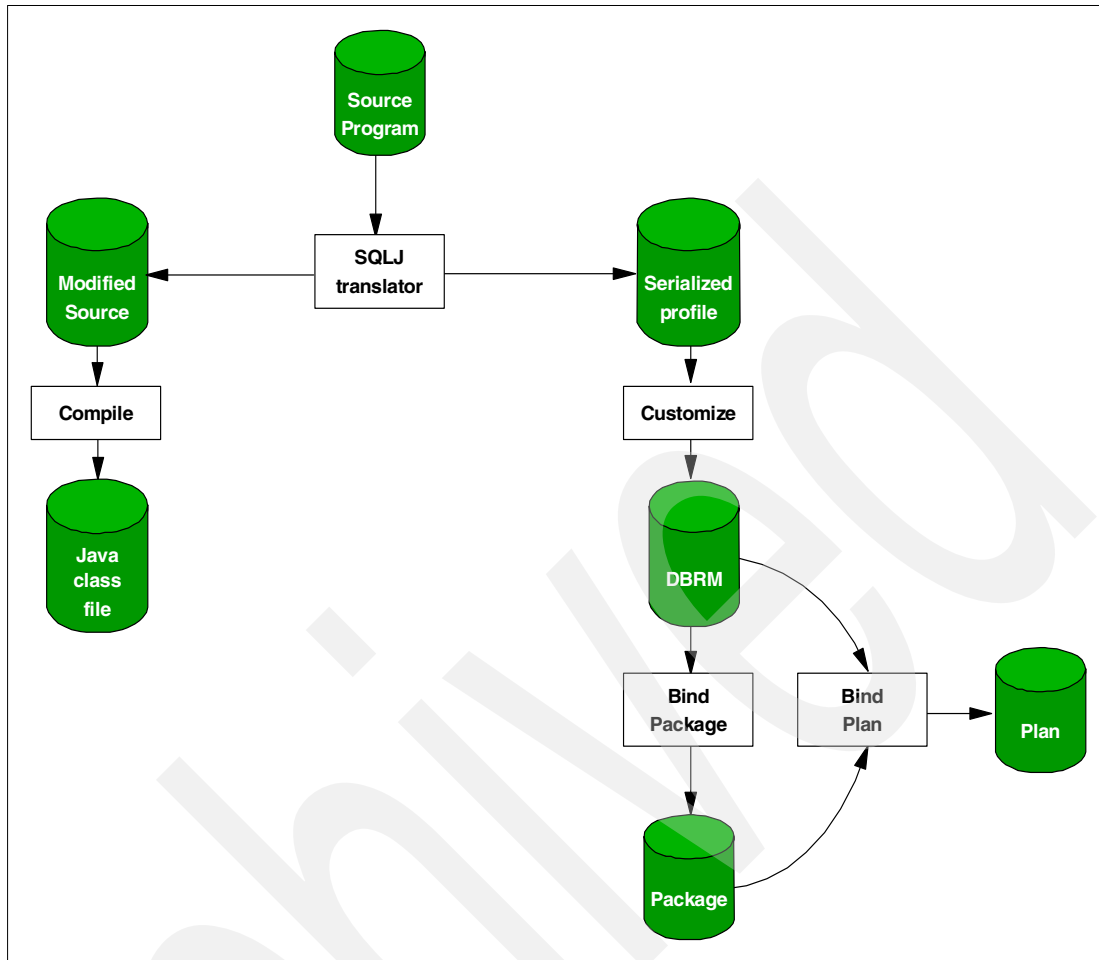


Figure 5-8 The SQLJ program preparation process

The advantages of SQLJ are the same as for static embedded SQL:

- ▶ Within DB2, better security can be provided since you only have to GRANT EXECUTE on the DB2 plans and/or packages, without any authorizations to end users on the actual application tables.
- ▶ The translator can perform checking at precompilation time (for example, type or schema checking and authorization checking).
- ▶ The customizing of profile files can improve performance. In the case of DB2, the use of packages allows to bind SQL statements before the execution of the application.

## 5.7.2 SQLJ and dynamic SQL

As mentioned earlier, the translator generates a serialized profile. In the DB2 case, the profile can be customized (using the *db2profc* command) to create standard DBRMs. Those you can bind, just as for any other program, into DB2 packages and/or plans, to have a true static SQLJ implementation.

However, if you don't go through these extra steps of customizing the profile and binding the DBRM, the SQLJ program can still be executed successfully. In this case, the calls to DB2 are executed as dynamic SQL statements using JDBC. Therefore, even though an SQLJ program is normally associated with static SQL in a DB2 for z/OS environment, you can still see dynamic SQL statements being executed if you have not customized the serialized profile, or if for some reason the DB2 package you created cannot be found.

On the other hand, if the SQL serialized profile has been customized for the DB2 application and is available on the run time path, there can still be dynamically prepared SQL statements with SQLJ. In some situations, positioned updates and deletes (UPDATE/DELETE WHERE CURRENT OF) are executed as dynamic SQL.

The SQLJ specification allows you to have the cursor OPEN and FETCH in one object and have the cursor UPDATE or DELETE in another object. An iterator can be passed as an argument to another method in another source file (another SQLJ profile and package). This presented a problem since DB2 requires that a positioned UPDATE/DELETE MUST be issued from the same package as the cursor (iterator). (In DB2, you cannot OPEN a cursor in PGMA and UPDATE it in PGMB.). Before PTF UQ59527 (APAR PQ51847), a dynamically prepared positioned UPDATE/DELETE was always used. This way the driver could be sure that the positioned UPDATE/DELETE was in the same package as the cursor.

To improve performance, with PTF UQ59527 applied, true static execution of a positioned UPDATE or DELETE can be used, when the positioned UPDATE/DELETE is issued from the same package as the cursor. To enable the feature, you must specify the “-staticPositioned=yes” option during the profile customization.

Guidelines for high performance in JDBC/SQLJ can be found in the Redpaper *DB2 for z/OS, Selected Performance Topics*.

Archived





## Developing REXX applications using dynamic SQL

In this chapter we describe how dynamic SQL is used in REXX applications. The REXX language interface for DB2 for z/OS provides a simple method for developing an application very quickly.

## 6.1 REXX interface, an alternative to embedded SQL

Compared with a host language such as COBOL using embedded SQL, the main advantage of using the REXX interface to DB2 is the substantial reduction in development time. Let us quickly review the steps to prepare a program containing embedded SQL. As shown in Figure 6-1, several steps are involved, leading to a longer development time.

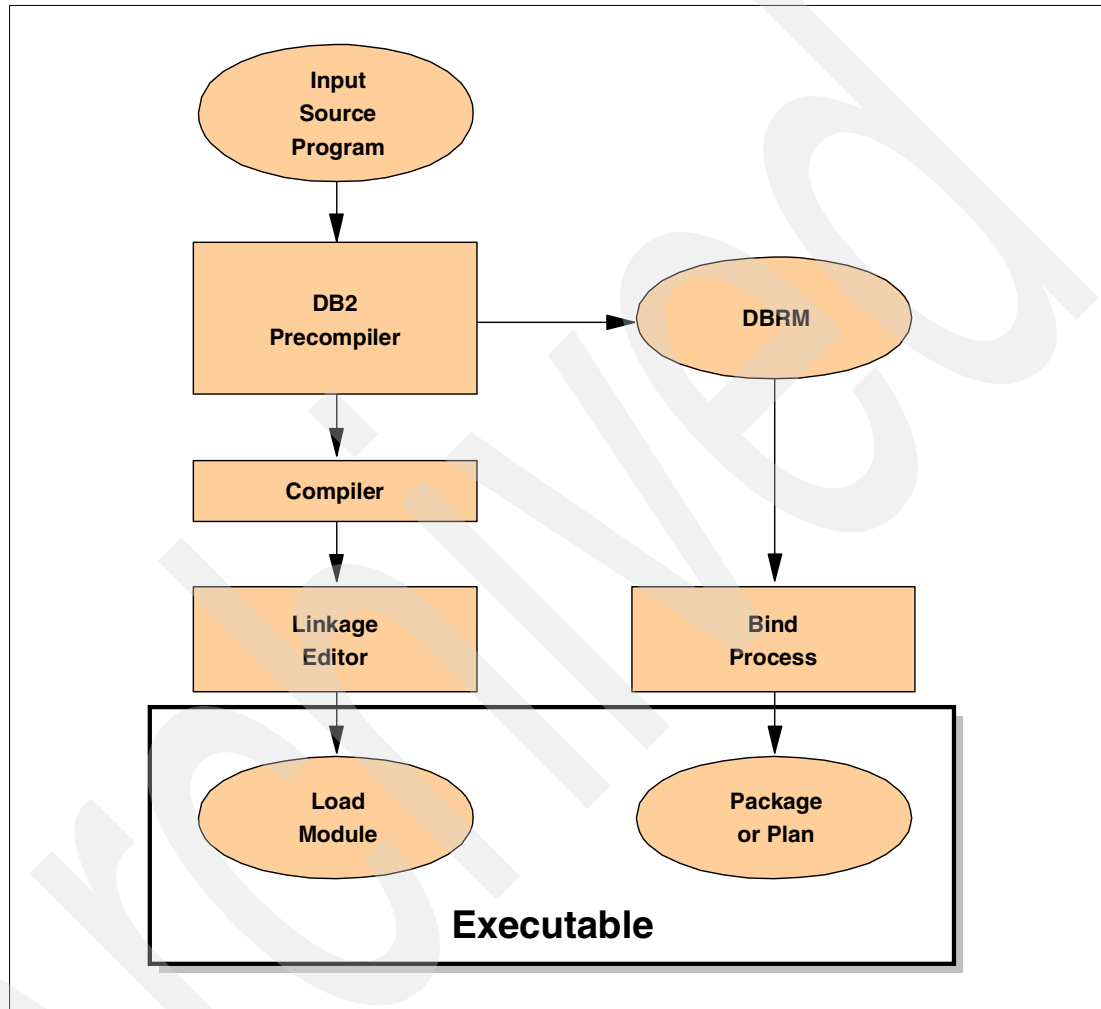


Figure 6-1 Program preparation for an embedded SQL program

Contrast this with the steps needed for a REXX program, as shown in Figure 6-2.

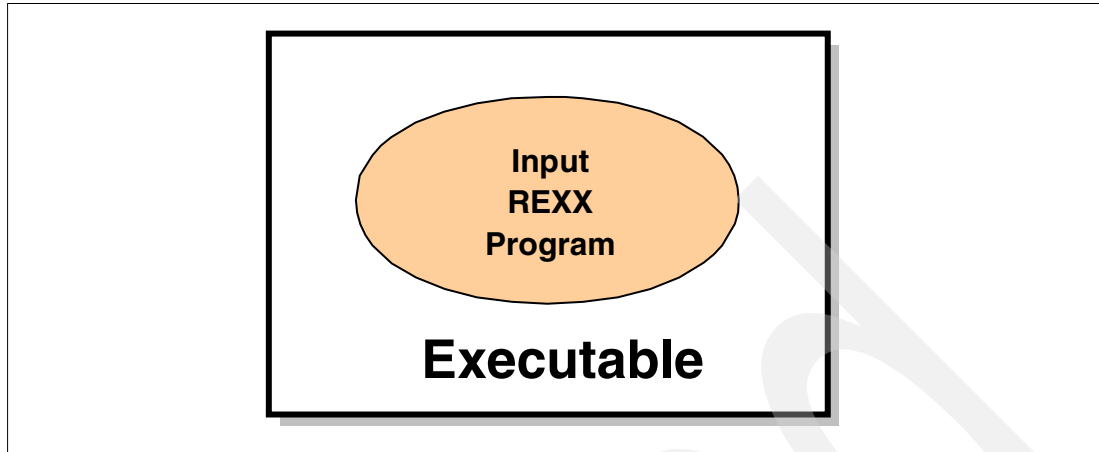


Figure 6-2 Program preparation for a REXX program

There is no pre-compile, compile, link, or bind needed — REXX is interpreted at run time.

Since several of the required components are pre-defined, a REXX program requires less code than a traditional program in a language such as COBOL. Combined with interactive debugging, this leads to a substantially shorter development time.

## 6.2 Connecting to DB2

Before issuing SQL calls, a REXX DB2 environment must be established by issuing the commands shown in Example 6-1.

Example 6-1 Setting up the REXX DB2 environment

---

```

ADDRESS TSO "SUBCOM DSNREXX"
IF RC THEN
DO
    S_RC = RXSUBCOMM('ADD','DSNREXX','DSNREXX')
END

```

---

There are two methods of connecting to DB2 which we now discuss:

► Via call attach:

This method can be used when executing in a TSO or batch environment. (It allows us to use ISPF edit macros when running under TSO.) This method of connecting cannot be used for stored procedures. To connect to a DB2 subsystem DB2G, the statements of Example 6-2 can be used.

Example 6-2 Connecting to DB2 via call attach

---

```

ADDRESS DSNREXX
SSID = 'DB2G'
"CONNECT SSID"
IF SQLCODE <> 0 THEN CALL DBERROR
To disconnect, we issue:
"DISCONNECT SSID"
IF SQLCODE <> 0 THEN CALL DBERROR

```

---

► Via RRS attach:

This method is required for stored procedures.

More choices exist when operating under a UNIX System Services (USS) environment. These include the ATTACH TO *ssid* and DETACH commands. These are not discussed in this book. See *DB2 Universal Database for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933 for details.

## 6.3 Pre-defined cursors and statements

The interface includes the following pre-defined elements:

► Cursors C1-C50:

These are defined using the WITH RETURN clause and correspond to statements S1 through S50 respectively. They do not have the WITH HOLD clause. They should be used for normal processing that does not require a cursor that must remain open across commits.

► Cursors C51-C100:

These are defined using the WITH RETURN and WITH HOLD clauses. As before, they correspond to statements S51 through S100 respectively. They should be used when cursor that remains open across commits is needed.

► Cursors C101-C200:

These are defined with the FOR RESULT SET clause and should be used for processing result sets created by a stored procedure by issuing ALLOCATE CURSOR statements. There are no statements associated with these cursors.

► Statements S1-S100:

These are associated with cursors C1 through C100 and may be used for this purpose.

**Important:** There is an implied relationship between a cursor and the corresponding statement. A cursor must be declared before its corresponding statement is used. This is true even when the statement contains a non-cursor operation such as UPDATE.

For example, to process an UPDATE via statement S1, a cursor C1 must be declared.

## 6.4 SQL statements not supported

The following SQL statements are not supported by the DSNREXX interface:

- BEGIN DECLARE SECTION
- END DECLARE SECTION
- INCLUDE
- SELECT INTO
- WHENEVER

## 6.5 Handling the SQLCA

Unlike programs that contain embedded SQL, that must contain an INCLUDE SQLCA statement, REXX programs inherit the elementary items contained in the SQLCA implicitly and these are populated by the REXX interface as follows:

<b>SQLCODE</b>	The primary SQL return code
<b>SQLERRMC</b>	Error and warning message tokens

<b>SQLERRP</b>	Product code, and if there is an error, the name of the module that returned the error
<b>SQLERRD.n</b>	Six variables containing diagnostic information (n=1 to 6)
<b>SQLWARN.n</b>	Eleven variables containing warning flags (n=0 to 10)
<b>SQLSTATE</b>	The alternate SQL return code

The use of REXX code to handle errors is discussed in Section 6.8, “Error handling” on page 128.

## 6.6 Handling the SQLDA

We discussed the structure, contents, and processing of the SQLDA in Chapter 3, “Developing embedded dynamic SQL applications” on page 19. For REXX programs, the method of using the SQLDA is similar, but there are differences in the details. Here, we focus only on these differences.

Just like the SQLCA, REXX programs cannot contain an INCLUDE SQLDA statement, but inherit the elementary items contained in the SQLDA implicitly. These are populated by the REXX interface as follows (*stem* refers to the variable name used for SQLDA in the PREPARE statement e.g. :MYSQLDA):

<b>stem.SQLD</b>	Number of variables contained in the SQLDA.
<b>stem.n.SQLNAME</b>	Name of the nth column in the result table.
<b>stem.n.SQLTYPE</b>	An integer representing the data type of the nth element.
<b>stem.n.SQLEN</b>	Length of data contained in stem.n.SQLDATA, except when the SQLTYPE is decimal.
<b>stem.n.SQLEN.SQLPRECISION</b>	Precision of the number if SQLTYPE is decimal.
<b>stem.n.SQLEN.SQLSCALE</b>	Scale of the number if SQLTYPE is decimal.
<b>stem.n.SQLCCSID</b>	The CCSID of the nth column of the result table.
<b>stem.n.SQLLOCATOR</b>	The result set locator value (for DESCRIBE PROCEDURE only).
<b>stem.n.SQLDATA</b>	Contains the input value supplied by the application or the output value fetched by the SQL. The format is determined by SQLTYPE, SQLEN, SQLEN.SQLPRECISION and SQLEN.SQLSCALE.
<b>stem.n.SQLIND</b>	Negative indicates the nth value is NULL. When used for setting values of an input variable, this variable is set to -1. <b>In addition, the corresponding SQLDATA must contain a valid value even though it is irrelevant. This is a special requirement for REXX only.</b>

**Note:** Unlike other languages like COBOL that contain the address of the host variable in SQLDATA and SQLIND, REXX contains the values themselves (not the addresses).

Also note that SQLN does not exist, nor does SQLDAID, since the allocation of the SQLDA is done automatically by the REXX interface.

## 6.7 Steps in developing REXX dynamic SQL applications

**Note:** A complete sample program in REXX can be downloaded from the Web. Instructions can be found in Appendix B, “Additional material” on page 251. We strongly encourage you to download this sample program before reading this and the next section.

We assume that a REXX environment has been established and a successful connection to DB2 has been made, as discussed in Section 6.2, “Connecting to DB2” on page 117. We now discuss in detail the specific steps a REXX application must take to handle the six primary variations of dynamic SQL discussed in Section 3.6, “Variations of embedded dynamic SQL” on page 24.

### 6.7.1 Non-SELECT without parameter markers

Handling non-SELECTs without parameter markers is the easiest type of statement to process. Figure 6-3 shows the steps involved.

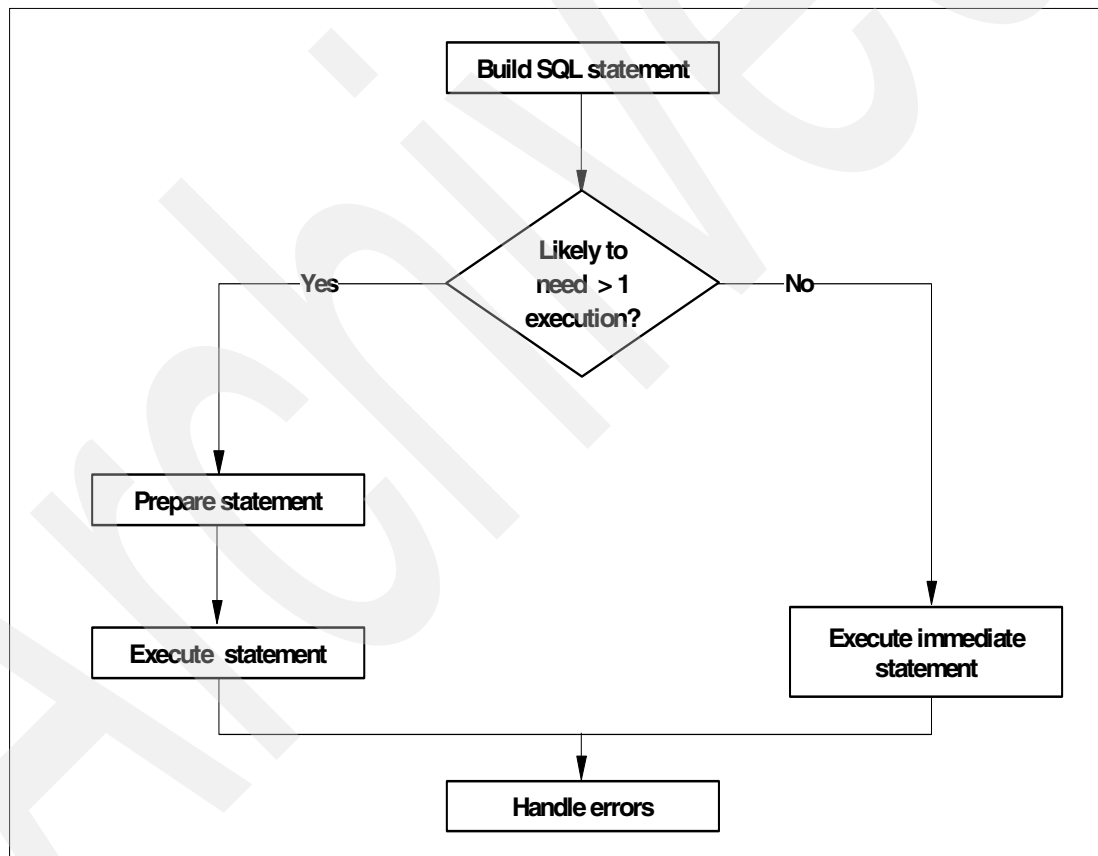


Figure 6-3 Steps for non-SELECT without parameter markers

#### 1. Build the SQL statement:

This step populates the contents of a host variable with the full SQL statement to be built and executed. This is dependent on the source of information. For example, the input may be obtained as a full or partial SQL statement from the terminal or a data set. A template statement that is further manipulated by the language may be used. The end result is a text variable containing the full SQL. In REXX, this step may look like this:

```
PARMSQL = 'UPDATE DSN8710.EMP SET SALARY = SALARY * 1.10 WHERE SAALRY < 50000'
```

2. Execute the statement:

As discussed in section Section 3.3, “The basics of using dynamic SQL” on page 21, this is accomplished by one of two methods - either by issuing an EXECUTE IMMEDIATE or by issuing a PREPARE followed by an EXECUTE. A REXX program would issue either:

```
EXECSQL PREPARE S1 FROM :PARMSQL
```

followed by:

```
EXECSQL EXECUTE S1
```

or issue:

```
EXESQL :PARMSQL
```

3. Handle any resulting errors:

Any resulting errors are handled by interrogating the contents of the SQLCA. A typical error-handling routine may be called and in a REXX program can look like this:

```
IF SQLCODE <> 0 THEN CALL DBERROR
```

### 6.7.2 Non-SELECT with parameter markers

This type of statement is also handled in a similar fashion. Since we have parameter markers, we need to associate the statement with a host variable that has been populated before the execution of the statement. Figure 6-4 shows the steps involved.

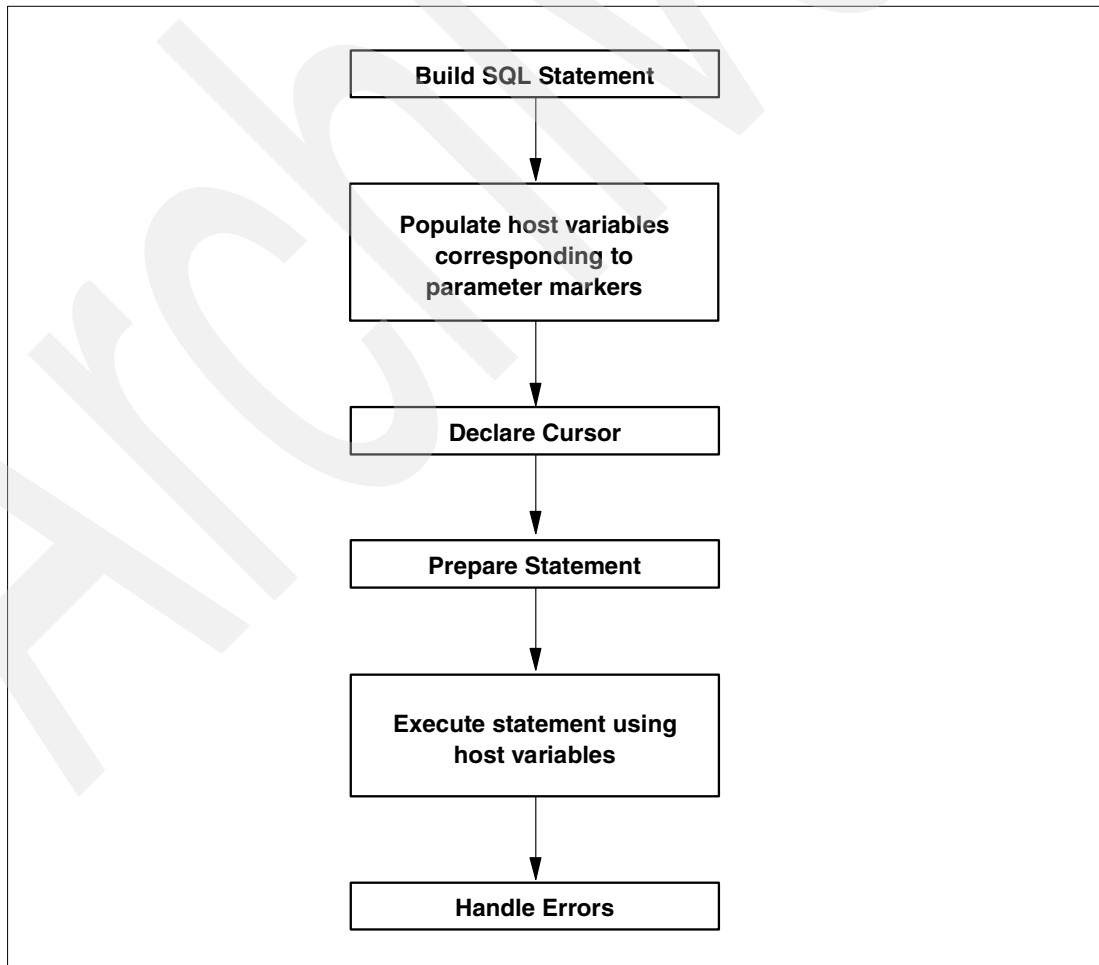


Figure 6-4 Steps for non-SELECT with parameter markers

These steps outline the sequence of actions required:

1. Build the SQL statement:

Populated as discussed before in Section 6.7.1, “Non-SELECT without parameter markers” on page 120. The only difference is that the statement is built with parameter markers. The string may contain, for example:

```
UPDATE      DSN8710.EMP
SET         SALARY = SALARY * 1.10
WHERE      EMPNO = ?
```

2. Populate host variables corresponding to parameter markers:

For each parameter marker specified in the statement, a corresponding host variable of the correct type must be populated. For example, for the statement above, a string variable must be declared and populated. In REXX, this is done as follows:

```
/*-- NOTE THE DBL/SINGLE/DBL COMBO - NEEDED FOR CHAR DATA THAT */
/*-- CONTAINS NUMERICS - REXX WOULD MAKES IT 100                */
HVEMPNO = ""000100""
```

3. Declare cursor:

***This step is needed even though we do not plan to use the cursor. In order to use statement S1, we must declare cursor C1.*** This is done as follows:

```
EXECSQL DECLARE C1 CURSOR FOR S1
```

4. Execute the statement:

This step is similar to the prepare and execute in Section 6.7.1, “Non-SELECT without parameter markers” on page 120, except for the need to associate each parameter with a corresponding host variable before execution (shown in step 2). Executing the statement is accomplished as follows:

```
EXECSQL PREPARE S1 FROM :PARMSQL
followed by
EXECSQL EXECUTE S1 USING :HVEMPNO
```

5. Handle any resulting errors:

This step is identical to the error-handling discussed in Section 6.7.1, “Non-SELECT without parameter markers” on page 120.

### 6.7.3 Fixed-list SELECT without parameter markers

Figure 6-5 shows the steps involved.



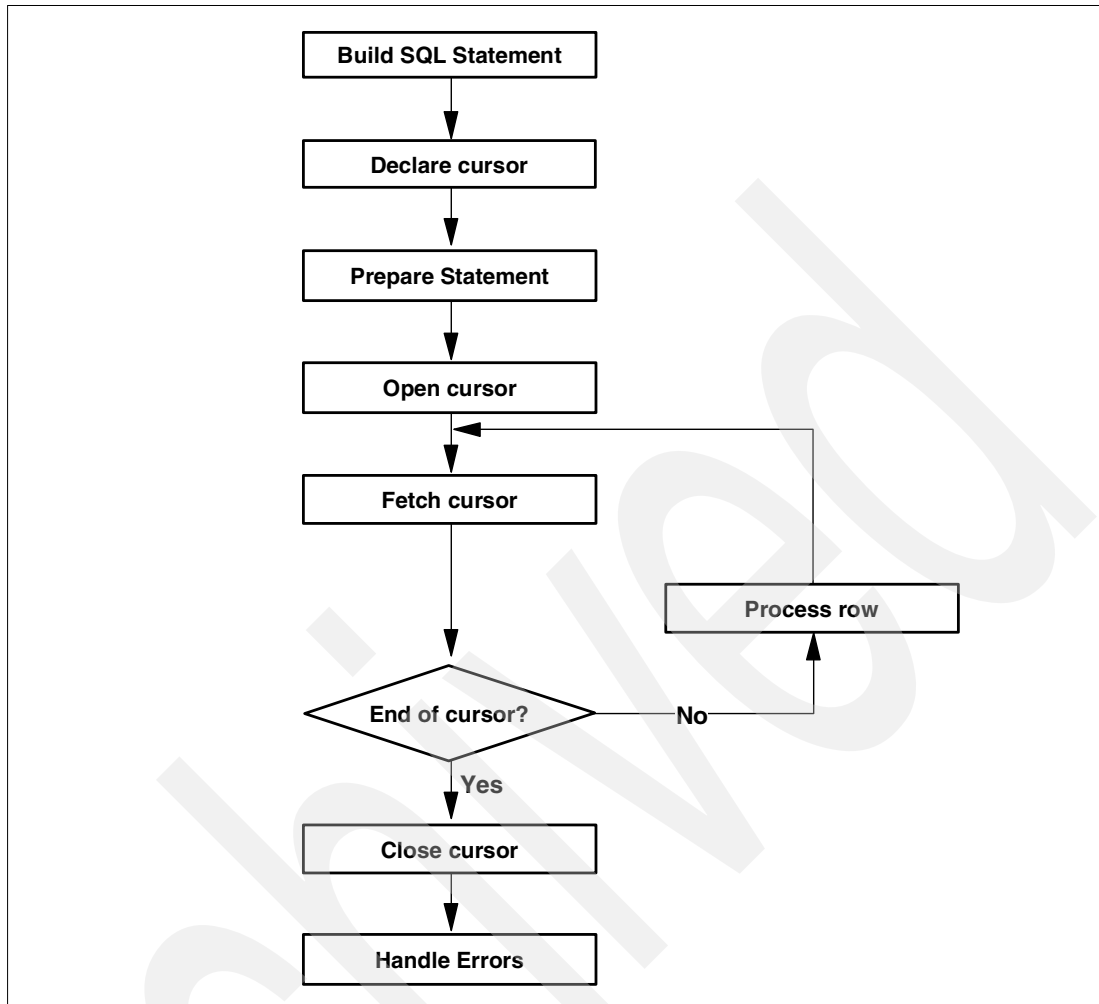


Figure 6-5 Steps for fixed-list *SELECT* without parameter markers

1. Build the SQL statement:

Populated as discussed before in Section 6.7.1, “Non-SELECT without parameter markers” on page 120. The statement may look like this:

```

SELECT      LASTNAME, SALARY
FROM        DSN8710.EMP
WHERE       EMPNO = '000100'
  
```

2. Declare a cursor:

```
EXECSQL DECLARE C1 CURSOR FOR S1
```

3. Prepare the statement:

As before, this can be done as follows:

```
EXECSQL PREPARE S1 FROM :PARMSQL
```

4. Open the cursor:

From this point on, the processing is identical to how a cursor is processed in a static SQL program. This can be done as follows:

```
EXECSQL OPEN C1
```

5. Fetch rows from cursor (looping until end of cursor):

Since this a fixed-list SELECT, the number and type of columns returned is known in advance and the FETCH statement simply obtains their values in compatible host variables, just like with static SQL. The statement may look like this:

```
EXECSQL FETCH C1 INTO :LASTNAME, :SALARY
```

6. Close the cursor:

This is the same as for static SQL. This step looks like this:

```
EXECSQL CLOSE C1
```

7. Handle any resulting errors:

This step is identical to the error-handling discussed in Section 6.7.1, “Non-SELECT without parameter markers” on page 120.

#### 6.7.4 Fixed-list SELECT with parameter markers

Most of the processing here is identical to that of the previous section, except for step 5 (new step) and step 6 (modified to include the USING clause on the OPEN). All steps are included and shown here for completeness. Figure 6-6 shows the steps involved.

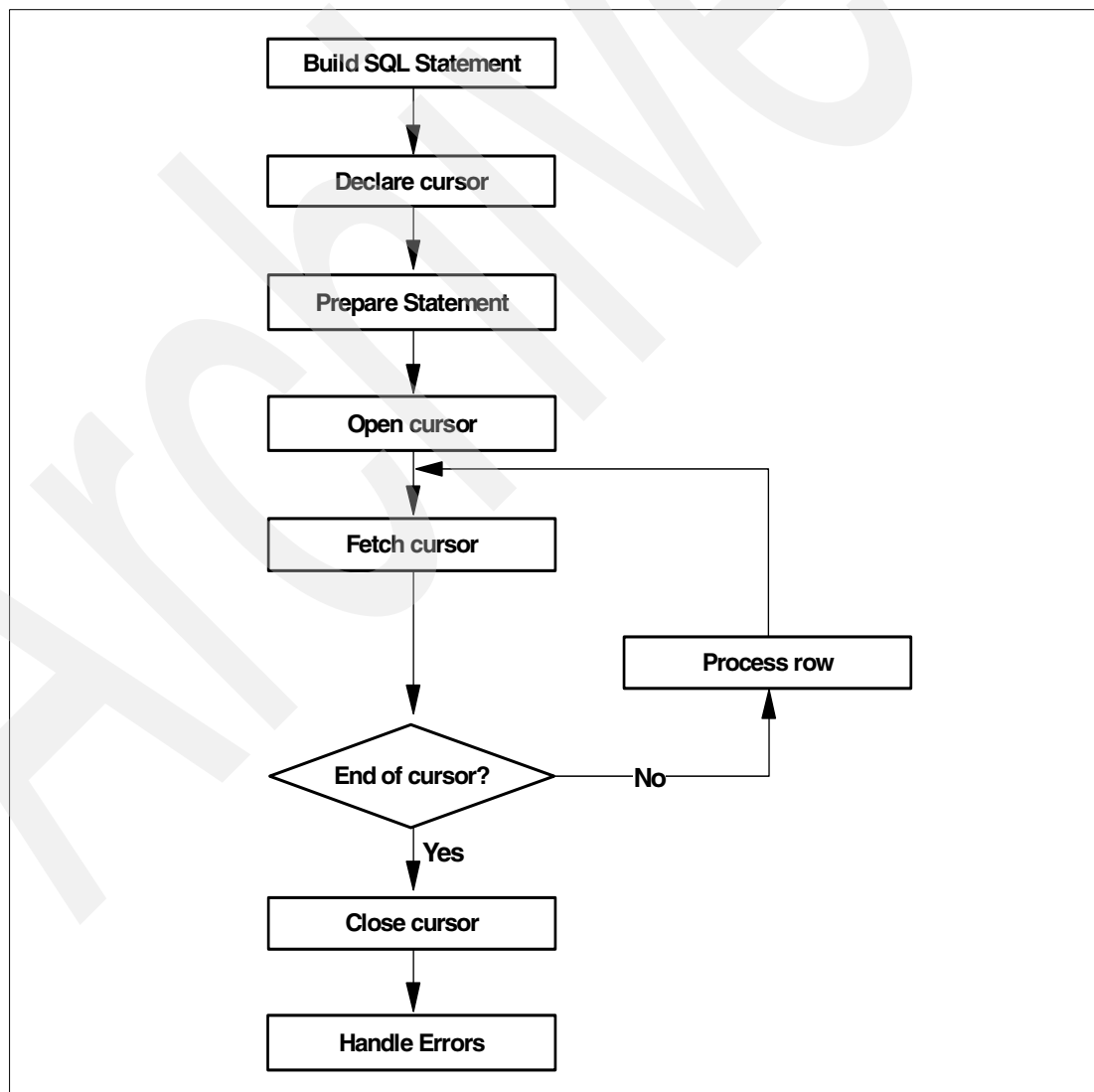


Figure 6-6 Steps for fixed-list SELECT with parameter markers

1. Build the SQL statement:

This is the same as in the previous section. The statement may look like this:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE ? AND SALARY > ?
```

2. Declare a cursor:

This is the same as in the previous section.

3. Prepare the statement:

This is the same as in the previous section.

4. Populate host variables corresponding to parameter markers:

For each parameter marker specified in the statement, a corresponding host variable of the correct type must be populated. For example, for the statement above we issue:

```
HVLASTNAME = "'S%'"
HVSALARY = 5000
```

5. Open the cursor:

The program could contain an OPEN statement like this:

```
EXECSQL OPEN C1 USING :HVLASTNAME, :HVSALARY
```

6. Fetch rows from cursor (looping until end of cursor):

This is the same as in the previous section.

7. Close the cursor:

This is the same as in the previous section.

8. Handle any resulting errors:

This is the same as in the previous section.

### 6.7.5 Varying-list SELECT without parameter markers

Figure 6-7 shows the steps involved.

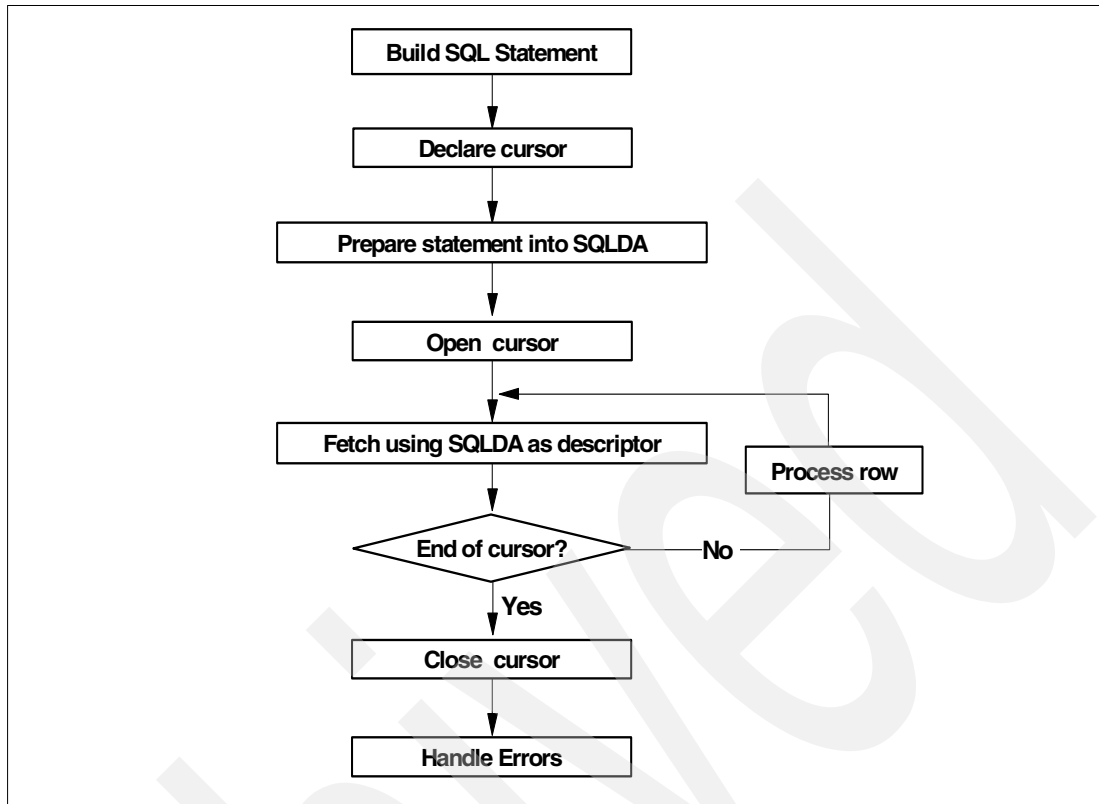


Figure 6-7 Steps for varying-list SELECT without parameter markers

1. Build the SQL statement:

Populated as discussed before in section Section 6.7.1, “Non-SELECT without parameter markers” on page 120.

2. Declare a cursor. We issue:

```
EXECSQL DECLARE C1 CURSOR FOR S1
```

3. Prepare the statement. The program issues:

```
EXECSQL PREPARE S1 INTO :OUTSQLDA FROM :PARMSQL
```

4. Open the cursor:

From this point on, the processing is identical to how a cursor is processed in a static SQL program. In a REXX program it looks like this:

```
EXECSQL OPEN C1
```

5. Fetch rows form the cursor (looping until end of cursor):

Unlike previous cases, the list of host variables where the values need to be fetched is known only via the SQLDA. For this reason, a different form of the FETCH statement must be used. Remember that in a REXX program the SQLDA contains the real data, not just pointers to the data like for example in COBOL. The program would contain a statement like this:

```
EXECSQL FETCH C1 USING DESCRIPTOR :OUTSQLDA
```

6. Close the cursor:

This is the same as for static SQL. This step would look like this in a COBOL program:

```
EXECSQL CLOSE C1
```

7. Handle any resulting errors:

This step is identical to the error-handling discussed in section Section 6.7.1, “Non-SELECT without parameter markers” on page 120.

### 6.7.6 Varying-list SELECT with parameter markers

This variation required the most complex type of processing. Figure 6-8 shows the steps involved.

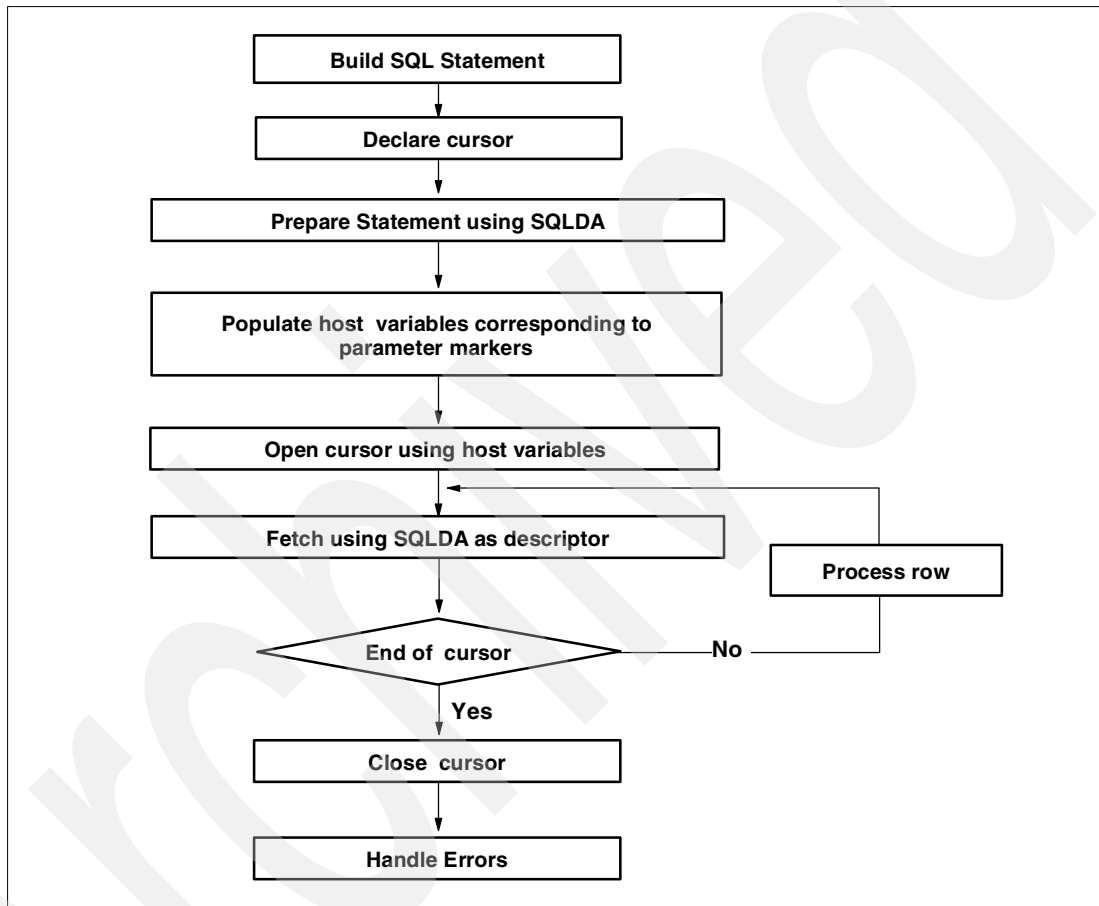


Figure 6-8 Steps for varying-list SELECT with parameter markers

1. Build the SQL statement:

Populated as discussed in the previous section.

2. Declare a cursor:

This is the same as in the previous section.

3. Prepare the statement:

This is the same as in the in the previous section.

4. Populate host variables corresponding to parameter markers:

For example, for the SQL statement shown in the next step, the program issues:

```
HVLASTNAME = ""S%""  
HVSALARY = 25000
```

5. Open the cursor:

This is similar to the previous section, but a USING clause must point to the host variables that correspond to the parameter markers in the correct sequence. A program that processes an SQL statement like this:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE ? AND SALARY > ?
```

contains an OPEN statement like this:

```
EXECSQL OPEN C1 USING :HVLASTNAME, :HVSALARY
```

6. Fetch rows from the cursor (looping until end of cursor):

This is the same as in the previous section.

7. Close the cursor:

This is the same as in the previous section.

8. Handle any resulting errors:

This is the same as in the previous section.

## 6.8 Error handling

In addition to normal REXX error handling logic any unexpected SQL errors should be handled by displaying the contents of SQLCA, as shown in Example 6-3.

*Example 6-3 Error handling in a REXX program*

```
"EXECSQL FETCH C1 INTO :HVLASTNAME, :HVSALARY"
IF      SQLCODE = 100 THEN ENDOFC1 = Y
ELSE IF SQLCODE <> 0 THEN CALL DBERROR
...
```

```
DBERROR:
SAY 'SQLSTATE= ' SQLSTATE
SAY 'SQLWARN = ' SQLWARN.0
SAY 'SQLWARN.1 = ' SQLWARN.1
SAY 'SQLWARN.2 = ' SQLWARN.2
SAY 'SQLWARN.3 = ' SQLWARN.3
SAY 'SQLWARN.4 = ' SQLWARN.4
SAY 'SQLWARN.5 = ' SQLWARN.5
SAY 'SQLWARN.6 = ' SQLWARN.6
SAY 'SQLWARN.7 = ' SQLWARN.7
SAY 'SQLWARN.8 = ' SQLWARN.8
SAY 'SQLWARN.9 = ' SQLWARN.9
SAY 'SQLWARN.10 = ' SQLWARN.10
SAY 'SQLERRD.1 = ' SQLERRD.1
SAY 'SQLERRD.2 = ' SQLERRD.2
SAY 'SQLERRD.3 = ' SQLERRD.3
SAY 'SQLERRD.4 = ' SQLERRD.4
SAY 'SQLERRD.5 = ' SQLERRD.5
SAY 'SQLERRD.6 = ' SQLERRD.6
SAY 'SQLERRP = ' SQLERRP
SAY 'SQLERRMC = ' SQLERRMC
SAY 'SQLCODE = ' SQLCODE
```

## 6.9 Choosing the correct isolation level

The REXX DB2 interface supports all four isolation levels, namely: repeatable read (RR), read stability (RS), cursor stability (CS) and uncommitted read (UR). There are two methods of changing the isolation level when executing an SQL statement:

- Using the WITH isolation level clause:

Here, we simply add the isolation level to the SQL statement text itself. For example, to issue the following select:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE ?
AND         SALARY > ?
```

With an isolation level of uncommitted read, we add:

```
WITH UR
```

- Using the CURRENT PACKAGESET option:

When you install DB2 REXX Language Support, you bind four packages for accessing DB2, each with a different isolation level in four different collection ids. They are:

Collection id	Isolation level
DSNREXRR	Repeatable read (RR)
DSNREXRS	Read stability (RS)
DSNREXCS	Cursor stability (CS)
DSNREXUR	Uncommitted read (UR)

You choose the desired level by choosing the proper collection id. For example, to use the isolation level of uncommitted read (UR), we issue:

```
"EXECSQL SET CURRENT PACKAGESET = 'DSNREXUR' "
```

## 6.10 Summary

The REXX DB2 interface provides a simple and easy-to-use interface for quick development and rapid prototyping of simple applications. It can be used to develop tools to enhance the productivity of the systems programmer or the DBA.

**Attention:** No formal performance studies have been conducted, but REXX programs should not be used for frequently executed mission-critical applications due to their overhead. Compared to using dynamic SQL in other language such as COBOL, the elapsed time for fetching hundreds of rows is significantly higher (by orders of magnitude). For this reasons, the REXX DB2 interface should be used only for low-volume applications whose performance is not critical.

Archived



## Preparing dynamic SQL applications

In this chapter we discuss various BIND options for programs containing dynamic SQL that impact performance, caching, concurrency, and security of these applications. These options include:

- ▶ REOPT(VARS)
- ▶ KEEP\_DYNAMIC
- ▶ ACQUIRE and RELEASE
- ▶ DYNAMICRULES
- ▶ DEFER(PREPARE)

For each option, we explain the choices and their impact. We also provide recommendations as to when each choice is appropriate. In some cases, the choices are interrelated.

## 7.1 REOPT(VARS) option

This section deals with the use of REOPT(VARS) option for dynamic SQL statements. Use of this option for static SQL as an alternative to using dynamic SQL is discussed in 2.6, “Choosing between REOPT(VARS) and dynamic SQL” on page 16.

### What are the choices and what do they mean?

A plan or package may be bound with NOREOPT(VARS), the default, or with REOPT(VARS). When REOPT(VARS) is specified, DB2 re-optimizes the access path at run time for all statements in the plan or package containing host variables, parameter markers or special registers. Since the actual values are known, the access path created can be substantially better than what would be created using default filter factors, especially when non-inform distribution or correlation of data columns exists. Note that REOPT(VARS) has no effect when no parameter markers are used in the dynamic SQL statements. In that case, all necessary information required to determine the best possible access path is available in the statement string.

REOPT(VARS) automatically implies DEFER(PREPARE), which means that the actual prepare is not carried out till the program issues an EXECUTE or OPEN CURSOR statement.

**Important:** Applications must be aware of the fact that, because of the implicit DEFER(PREPARE), any errors that would normally be returned after the PREPARE are now returned after the EXECUTE or OPEN if REOPT(VARS) is used.

For dynamic SQL, since it must be prepared (at run time) before it is executed in any case, it would appear that preparing only after the value of host variables, parameter markers and special registers are known would be preferred. While this is true in general, there are a few negative impact that you must be aware of. The first is potential for a double-prepare and its impact on dynamic statement caching. These are discussed below.

### Avoiding double-prepare for non-SELECT statements

When a program issues a DESCRIBE followed by an EXECUTE statement, DB2 prepares the statement twice - once for DESCRIBE without considering the values of the variables that correspond to the parameter markers and a second time with these values taken into account. If such statements are executed frequently, they can cause a degradation of performance. In this case, they should be isolated into a separate package which is bound with NOREOPT(VARS) option.

### Avoiding double-prepare for SELECT statements

A similar situation exists for SELECT statements. When a program issues a DESCRIBE followed by an OPEN CURSOR statement, DB2 prepares the statement twice - the values of variables that correspond to parameter markers being considered only during the OPEN CURSOR. In this case, simply reversing the order (OPEN CURSOR followed by DESCRIBE) will eliminate the double prepare. Note that the PREPARE INTO automatically issues a DESCRIBE and must also be avoided prior to opening the cursor.

### Impact on dynamic statement caching

Statement in plans or packages bound with REOPT(VARS) are not eligible for dynamic statement caching. The implications of this are that the benefits gained by using REOPT(VARS) (a potentially better access path) must be weighed against the loss of the dynamic statement caching capability.

### **Impact on predictive governing**

For packages and plans bound with the REOPT(VARS) BIND option, the warning SQLCODE +495 cannot be reported on the prepare of a dynamic SQL statement with host variables that is reoptimized.

### **Recommendations**

When dynamic statement caching is not active, use REOPT(VARS) for all dynamic SQL statements which have host variables, parameter markers, or special registers, making sure that techniques to avoid the double-prepares discussed earlier, are implemented. When dynamic statement caching is in effect, use REOPT(VARS) sparingly and only for those statements that are likely to benefit. Isolate such statements into separate packages if required.

## **7.2 KEEPYNAMIC option**

In this section we discuss considerations regarding the KEEPYNAMIC option.

### **What do the various choices mean?**

Chapter 8, “Dynamic statement caching” on page 141 discusses dynamic statement caching in detail. Briefly, there are two forms of dynamic statement caching - local and global. The local caching occurs at the thread level and is activated by binding an application with KEEPYNAMIC(YES). It allows an application to hold dynamic statements past a commit point. (When using KEEPYNAMIC(NO), the default, a dynamic SQL statement has to be re-prepared after a commit).

The prepared form of the statement may or may not be available when the statement is executed again (depending the amount of prepares in the systems and the MAXKEEPD DSNZPARM, that determines the number of statements (at the subsystem level) that can be kept across a commit). If available, an EXECUTE request by the application re-uses the statement in the local cache. If it is not available, it is implicitly prepared by DB2 before running the EXECUTE statement. In that case, DB2 will first look in the global cache if the statement is cached there or DB2 might have to do a full prepare.

Remember that the application must be coded in a special way in order to be able to take advantage of the local cache. The statement must be prepared only once. If you execute a new PREPARE for a locally cached statement, the information in the local cache is destroyed and the statement will be re-prepared.

### **Recommendations**

Use this only for applications likely to benefit from local statement caching after the impact has been evaluated.

## **7.3 RELEASE option**

In this section we discuss considerations regarding the RELEASE BIND option.

### **What do the various choices mean?**

The RELEASE BIND option generally applies only to static SQL statements, since dynamic SQL statements lock objects when accessed and release locks at the next commit point, except for the situation discussed below.

If RELEASE(DEALLOCATE) is used, KEEPYNAMIC(YES) is used and global dynamic statement caching for the subsystem is active (DSNZPARM CACHEDYN=YES), then locks are held until deallocation or the first commit point that follows the removal of the statement from the local cache. The statement is removed from the local cache for one of the following reasons:

- ▶ The application issues a PREPARE with the same statement identifier.
- ▶ The statement is removed from the cache because it has not been used sufficiently recent.
- ▶ 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.
- ▶ A ROLLBACK was issued.

### Recommendations

The choice should be based on concurrency requirements taking into account the additional duration of locks that may be possible when local and global statement caching is active and you use RELEASE(DEALLOCATE). More information about locking when using dynamic SQL can be found in Chapter 10, “Locking and concurrency” on page 161.

## 7.4 DYNAMICRULES option

In this section we discuss considerations regarding the DYNAMICRULES option.

### What do the various choices mean?

The DYNAMICRULES option along with the run time environment of a package (whether package is run stand-alone or under the control of a stored procedure or user-defined function) determines the authorization id used to check authorization, the qualifier for unqualified objects, the source of application programming options for SQL syntax and whether or not the SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP and RENAME statements.

Table 7-1 below shows how DYNAMICRULES and the run time environment affect the dynamic SQL statement behavior.

Table 7-1 How runtime behavior is determined

DYNAMICRULES value	Stand-alone program environment	Stored procedure or user-defined function 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

Table 7-2 shows the implications of each dynamic SQL statement behavior.

Table 7-2 What the runtime behavior means

Dynamic SQL attribute	Bind behavior	Run behavior	Define behavior	Invoke behavior
Authorization id	Plan or package owner	Current SQLID	Owner of user-defined function or stored procedure	Authorization id of invoker
Default qualifier for unqualified objects	Bind OWNER or QUALIFIER value	Current SQLID	Owner of user-defined function or stored procedure	Authorization id of invoker
CURRENT SQLID	Not applicable	Applies	Not applicable	Not applicable
Source for application programming options	Determined by DSNHDECP parameter DYNRULS	Install panel DSNTIPF	Determined by DSNHDECP parameter DYNRULS	Determined by DSNHDECP parameter DYNRULS
Can execute GRANT, REVOKE, ALTERM DROP, RENAME?	No	Yes	No	No

## Recommendations

Use the value appropriate for your environment. In a z/OS server-only environment. DYNAMICRULE(BIND) makes embedded dynamic SQL behave similar to embedded static SQL and is probably the best option for most users if the users are not allowed to use “free form SQL”. In a distributed environment, binding multiple packages using different levels of authorization may provide the best granularity. For a more detailed discussion, see Chapter 13, “Security aspects of dynamic SQL” on page 221.

## 7.5 DEFER(PREPARE) option

By using the DEFER(PREPARE) option, you indicate to DB2 that the actual PREPARE of the statement can be delayed until an EXECUTE, OPEN cursor or DESCRIBE for the dynamic statement is issued. This option is discussed in Chapter 14, “Remote dynamic SQL” on page 229.

## 7.6 Specifying access path hints for dynamic SQL

Optimization hints are not typical to dynamic SQL. The reason to discuss the matter here is that the way to specify optimization hints is different for static compared to dynamic SQL.

For static SQL, the hint is controlled via the BIND parameter OPTHINT. For example:

```
BIND/REBIND PACKAGE (...)
  OPTHINT('GOODPATH')
  EXPLAIN(YES)
  ...
```

For dynamic SQL, the hint is provided via the special register CURRENT OPTIMIZATION HINT. For example:

```
SET CURRENT OPTIMIZATION HINT = 'GOODPATH';
```

In the absence of a value for CURRENT OPTIMIZATION HINT, DB2 uses the OPTHINT value used during the BIND/REBIND of the package or plan that contains the embedded dynamic SQL.

If a hint is provided for dynamic SQL, the application should issue the following:

1. SET CURRENT OPTIMIZATION HINT = ...
2. PREPARE the dynamic SQL statement
3. Examine the SQLCODE and take appropriate action if needed
  - a. SQLCODE = +394 means the hint was used
  - b. SQLCODE = +395 means the hint was not used.  
In this case, EXPLAIN the dynamic SQL statement

The situations when it may be appropriate to influence the access path of an SQL statement are similar for dynamic and static SQL. They are discussed in chapter 31 of *DB2 Universal Database for OS/390 and z/OS Administration Guide*, SC26-9931.

When searching the PLAN\_TABLE for qualifying rows, the statement being prepared must match up on the following columns: QUERYNO, APPLNAME, PROGNAME, VERSION, COLLID and OPTHINT. The query number (QUERYNO) is normally the same as the statement number, unless the QUERYNO clause was used in the SQL statement.

When using dynamic SQL, using the QUERYNO clause has some advantages. Applications, such as DSNTEP2 or QMF, use the same PREPARE statement for each dynamic statement. Therefore the same query number (statement number) is used to identify each dynamic statement. By assigning a QUERYNO to each statement that uses optimization hints, you can eliminate ambiguity as to which rows in the PLAN\_TABLE are associated with each query. For more information on how to use the QUERYNO clause, see *DB2 UDB for OS/390 Version 6 Performance Topics*, SG24-5351.

## 7.7 Specifying the degree of parallelism for dynamic SQL

Query parallelism, although probably more frequently used in a dynamic SQL environment, especially for ad-hoc warehouse type of queries, is not typical to dynamic SQL. The reason to discuss the matter here is that the way to specify the parallel degree is different for static compared to dynamic SQL.

For static SQL, the degree of parallelism permitted is controlled via the BIND parameter DEGREE. For example:

```
BIND/REBIND PACKAGE (...)  
DEGREE(ANY)  
EXPLAIN(YES)  
...
```

For dynamic SQL, the DEGREE is provided via the special register CURRENT DEGREE. For example:

```
SET CURRENT DEGREE = 'ANY';
```

In the absence of a value for CURRENT DEGREE, DB2 uses the DEGREE value used during the BIND/REBIND of the package or plan that contains the embedded dynamic SQL.

## 7.8 Obtaining access path information for dynamic SQL

While it is possible to capture the SQL text via the dynamic statement cache, or starting a DB2 trace, obtaining such information *before* its execution is generally difficult. For static SQL, the statement text is available in the SYSPACKSTMT or SYSSTMT catalog tables and the access path information is available in the associated PLAN\_TABLE(s) after binding the plan or package with the EXPLAIN(YES) option.

If a complete quality assurance exercise is to be carried out for an application that uses dynamic SQL, this must be planned carefully and *all possible SQL statements generated and explained before* a production implementation. Depending on the source and number of variations of the SQL statements, this can be a monumental task.

The “easiest” way to accomplish this is probably using DB2 traces to capture the SQL statements and their corresponding access path information (mini-plan). More information on how to obtain this information can be found in 12.5, “Using the performance database of DB2 PM” on page 199.

To obtain access path information for individual dynamic SQL statements use:

```
EXPLAIN PLAN
SET QUERYNO = nnn FOR
sql statement;
```

For more information on how to obtain and exploit EXPLAIN information, see Chapter 23 of *DB2 Universal Database for OS/390 and z/OS Version 7 Administration Guide*, SC26-9931.

Archived



# Managing dynamic SQL

In this part of the book, we discuss how to manage and control an environment using dynamic SQL. The following topics are covered:

- ▶ Dynamic statement caching
- ▶ Administering and controlling the execution of dynamic SQL:
  - System controls for dynamic SQL
  - Locking and concurrency
  - Governing dynamic SQL
- ▶ Monitoring, tracing, and explaining dynamic SQL
- ▶ Security aspects of dynamic SQL
- ▶ Remote dynamic SQL

Archived



## Dynamic statement caching

In this chapter we describe the different types of dynamic statement caching:

- ▶ Local dynamic statement cache
- ▶ Global dynamic statement cache
- ▶ Full caching

## 8.1 Overview

With every release of DB2, the optimizer has become more and more sophisticated. The price you have to pay to be able to get the best possible access path is that the cost of preparing an SQL statement increases. Applications using dynamic SQL pay this cost at execution time and might have to pay this cost more than once. (An application must prepare a dynamic SQL statement again after a commit, even if that dynamic SQL statement had been prepared before.)

In DB2 for OS/390 Version 5 a feature called *dynamic statement caching* was introduced. The concept is simple. Whenever DB2 prepares an SQL statement, it creates a control structure that is used when the statement is executed. When dynamic statement caching is in effect, DB2 stores the control structure associated with a prepared dynamic SQL statement in a storage pool. If that same statement is executed again, DB2 can reuse the cached control structure, avoiding the expense of re-preparing the statement.

The following SQL statements can be cached:

- ▶ SELECT
- ▶ UPDATE
- ▶ INSERT
- ▶ DELETE

Both local and distributed SQL statements are eligible for caching. Prepared, dynamic statements using DB2 private protocol access are eligible as well. However, note that static statements which use DB2 private protocol access, even though they are prepared dynamically at the remote site, are not eligible for caching. Refer to Chapter 14.1, “Protocols supported by remote SQL” on page 230 for detailed information.

Statements in plans or packages bound with REOPT(VARS) are not cached in the global cache. The bind options REOPT(VARS) and KEEP DYNAMIC(YES) are not compatible.

In a data sharing environment prepared statements cannot be shared among the members. As each member has its own EDM pool. A cached statement of one member is not available to an application that runs on another DB2 member.

There are different levels of statement caching, which are explained in the following sections:

- ▶ Local dynamic statement caching
- ▶ Global dynamic statement caching
- ▶ Full caching

## 8.2 The different levels of statement caching

Let us start explaining the different levels of caching with an example of two programs in a subsystem without caching.

### 8.2.1 No caching

Program A prepares a dynamic SQL statement S, executes the prepared statement twice, and terminates.

Program B starts after program A had terminated, prepares exactly the same statement S as A did, executes the prepared statement, issues a commit, tries to execute S again, receives an error SQLCODE -514 or -518 (SQLSTATE 26501 or 07003), has to prepare the same statement S again, executes the prepared statement and terminates.

Each time a prepare has been issued by the programs A and B, via the SQL PREPARE statement, DB2 prepared the statement from scratch. After the commit of program B, the prepared statement is invalidated, so program B had to repeat the prepare of statement S. Figure 8-1 helps to visualize this behavior.

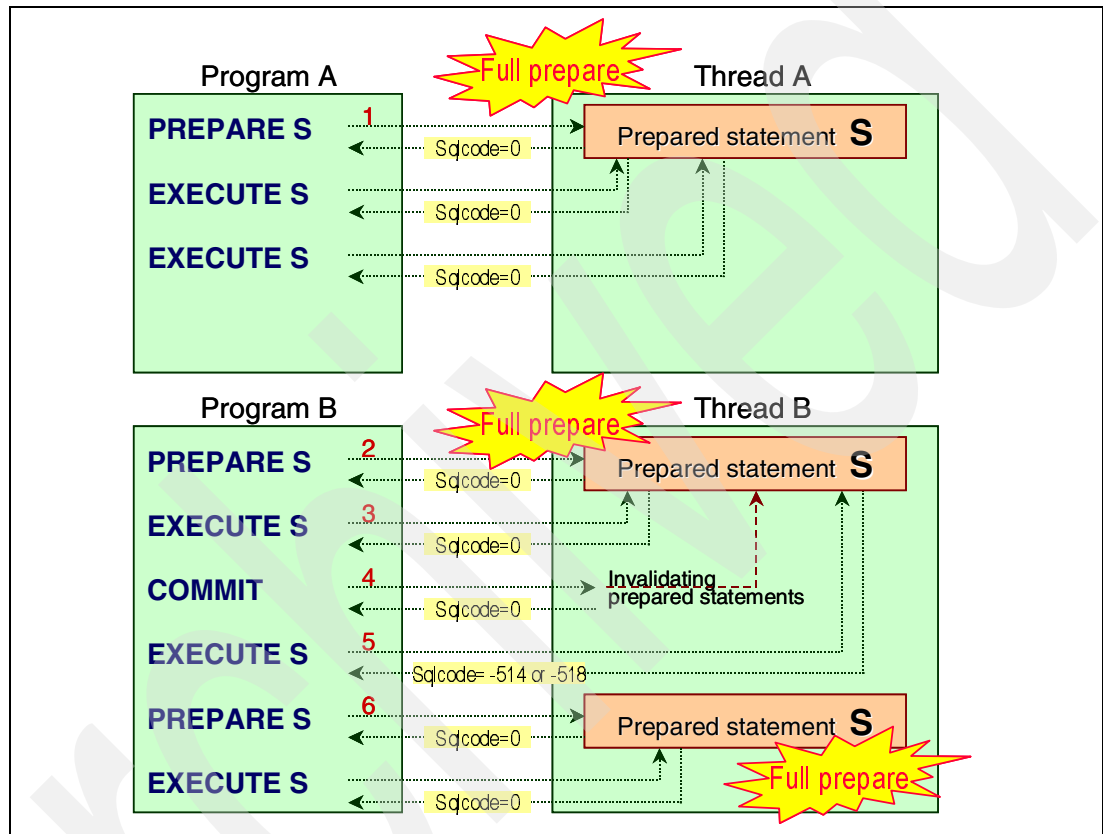


Figure 8-1 Dynamic SQL without statement caching

### 8.2.2 Local dynamic SQL cache only

A local dynamic statement cache is allocated in the storage of each thread in the DBM1 address space. You can control the usage of this cache by using `KEEPDYNAMIC` the bind option.

Bound with `KEEPDYNAMIC(YES)`, an application can issue a `PREPARE` for a statement once and omit subsequent `PREPARE`s for this statement, even after a commit has been issued.

To understand how the `KEEPDYNAMIC` bind option works, it is important to differentiate between the executable form of a dynamic SQL statement (the *prepared statement*) and the character string form of the statement (the *statement text*).

Let us take a look at our two example programs, shown in Figure 8-2.

Program A prepares a dynamic SQL statement S, executes the prepared statement twice, and terminates.

Program B starts after program A has terminated, prepares the same statement S as A did, executes the prepared statement, issues a commit, executes S again (causing an internal (implicit) prepare) and terminates.

Each time an SQL PREPARE is been issued by the programs (or DB2 for the implicit prepare), a complete prepare is executed. (This process is also called full prepare. See “Different prepare types” on page 148 for more information). After the COMMIT of program B, the prepared statement is invalidated (since the cursor was not open and not defined with hold), but the statement text has been preserved in the local statement cache of the thread (because it was bound with KEEP DYNAMIC(YES)). Therefore program B does not have to repeat the prepare of statement S explicitly; it can immediately issue the EXECUTE again. Under the covers, DB2 will execute a complete prepare operation, using the saved statement string. This operation is called an *implicit prepare*.

Be aware that application program B has to be able to handle the fact that the implicit prepare may fail and an error is returned. Any error that normally occur at prepare time can now be returned on the OPEN, EXECUTE, or DESCRIBE statement issued by the application.

The prepared statement and the statement text are held in the thread’s local storage within the DBM1 address space (outside the EDM pool). But only the statement text is kept across commits when you only use local caching.

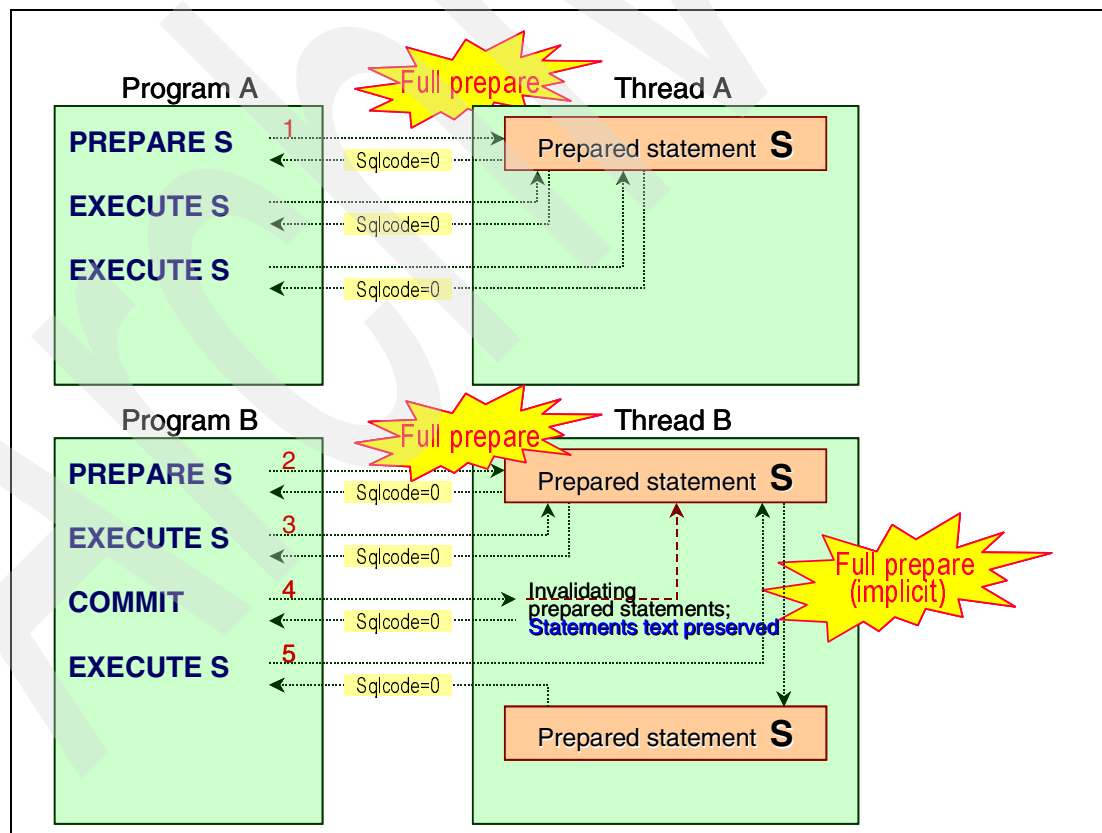


Figure 8-2 Local dynamic statement caching

The local instance of the prepared SQL statement (the prepared statement), is kept in DBM1 storage until one of the following occurs:

- ▶ The application process ends.
- ▶ The application commits and there is no open cursor defined WITH HOLD for the statement. (Since we are using only local caching, just the statement string is kept across commits.)
- ▶ A rollback operation occurs.
- ▶ The application issues an explicit PREPARE statement with the same statement name.

If the application issues a PREPARE for the same SQL statement name which is kept in the cache, the kept statement is discarded and DB2 prepares the new statement.

In a distributed environment, if the requester does not issue a PREPARE after a COMMIT, the package at the DB2 for OS/390 and z/OS server must be bound with KEEP\_DYNAMIC(YES). If both requester and server are DB2 for OS/390 and 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 may have 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 for OS/390 and 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.

When a distributed thread has touched any package which is bound with KEEP\_DYNAMIC(YES), the thread cannot become inactive. You find more information about this subject in Chapter 14.2, “Reusing database access threads” on page 230.

This level of caching, used without other caching possibilities, is of minor value, because the performance improvement is limited. The only advantage is that you can avoid coding a PREPARE statement after a COMMIT, since DB2 keeps the statement string around. This is of course most beneficial in a distributed environment where you can save a trip across the wire this way. On the other hand, by using the DEFER(PREPARE) bind option, you can obtain similar network message savings.

### 8.2.3 Global dynamic statement cache only

The global dynamic statement cache is normally allocated in the EDM pool within the DBM1 address space. You can activate this cache by setting CACHEDYN=YES in DSNZPARM. The global statement cache can also reside in a data space instead of the EDM pool when a non-zero value is specified for the EDMDSPAC DSNZPARM value. (See 9.2.2, “Controlling global cache storage (EDM pool)” on page 156 for more details.)

When global dynamic statement caching is active, the skeleton copy of a prepared SQL statement (SKDS) is held in the global dynamic statement cache inside the EDM pool. Only one skeleton copy of the same statement (matching text) is held. The skeleton copy can be used by user threads to create user copies. An LRU algorithm is used for replacement. If an application issues a PREPARE or an EXECUTE IMMEDIATE (and the statement has not been executed before in the same commit scope), and the skeleton copy of the statement is found in the global statement cache, it can be copied from the global cache into the thread’s storage. This is called a *short prepare*.

**Note:** Without local caching (KEEPDYNAMIC(YES)) active, the application cannot issue EXECUTEs directly after a commit. The statement returns an SQLCODE -514 or -518, SQLSTATE 26501 or 07003.

Let us take a look at our example again. The global cache case is shown in Figure 8-3.

Program A prepares a dynamic SQL statement S, executes the prepared statement twice, and terminates.

Program B starts after program A has terminated, prepares the same statement S as A did, executes the prepared statement and issues a COMMIT. Then program B tries to execute S again. The program receives an error SQLCODE -514 or -518 (SQLSTATE 26501 or 07003) and has to prepare the same statement S again. Then it executes the prepared statement and terminates.

The first time a prepare for statement S is issued by the program A, a complete prepare operation is performed. The SKDS of S is then stored in the global statement cache. When program B executes the prepare of S for the first time, the SKDS is found in the global statement cache and is copied to the local storage of B's thread (short prepare). After the COMMIT of program B, the prepared statement is invalidated in B's local storage, but the SKDS is preserved in the global statement cache in the EDM pool. Because the statement string nor the prepared statement is kept after the commit, program B has to repeat the prepare of statement S explicitly. This causes another copy operation of the SKDS in the global cache to the local storage of the thread of application B (short prepare).

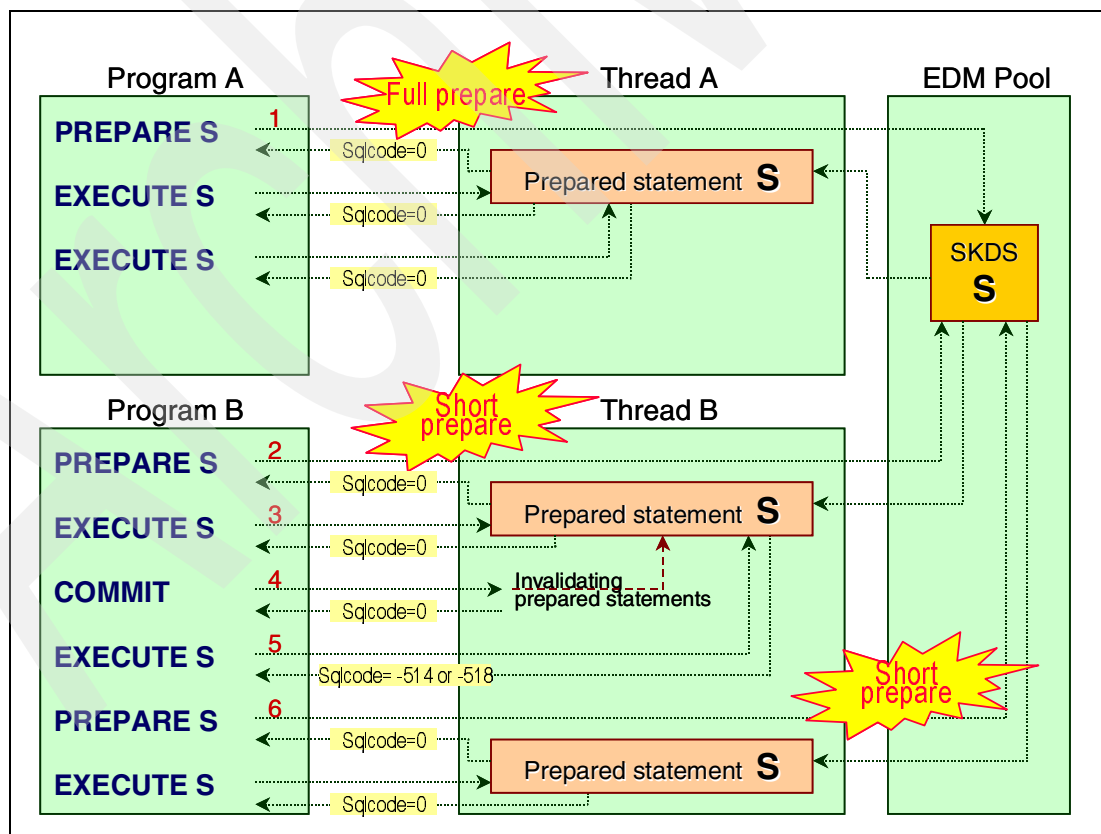


Figure 8-3 Global dynamic statement caching

This level of statement caching has important performance advantages.



## 8.2.4 Full caching

Full caching is a combination of local caching (KEEPDYNAMIC(YES)), a MAXKEEPD DSNZPARM value > 0, and global caching (CACHEDYN=YES). It is possible to avoid prepares, because a commit does not invalidate prepared statements in the local cache.

Let us look again at our example when full caching is active, shown in Figure 8-4.

Program A prepares a dynamic SQL statement S, executes the prepared statement twice, and terminates.

Program B starts after program A has terminated, prepares the same statement S as A did, executes the prepared statement, issues a commit, executes S again, and terminates.

The first time a prepare for statement S is issued by the program A, a complete prepare is done (full prepare). The SKDS of S is stored in the global cache. When program B executes the prepare of S the first time, the SKDS is found in the global statement cache and is copied to the local statement cache of B's thread (short prepare). The COMMIT of program B has no effect on the prepared statement. When full caching is active, both the statement string which is also kept for local caching only, and the prepared statement are kept in the thread's local storage after a commit. Therefore, program B does not have to repeat the prepare of statement S explicitly, and it was not necessary to do the prepare the statement implicitly since the full executable statement is now kept in the thread's local storage. This case is called *prepare avoidance*.

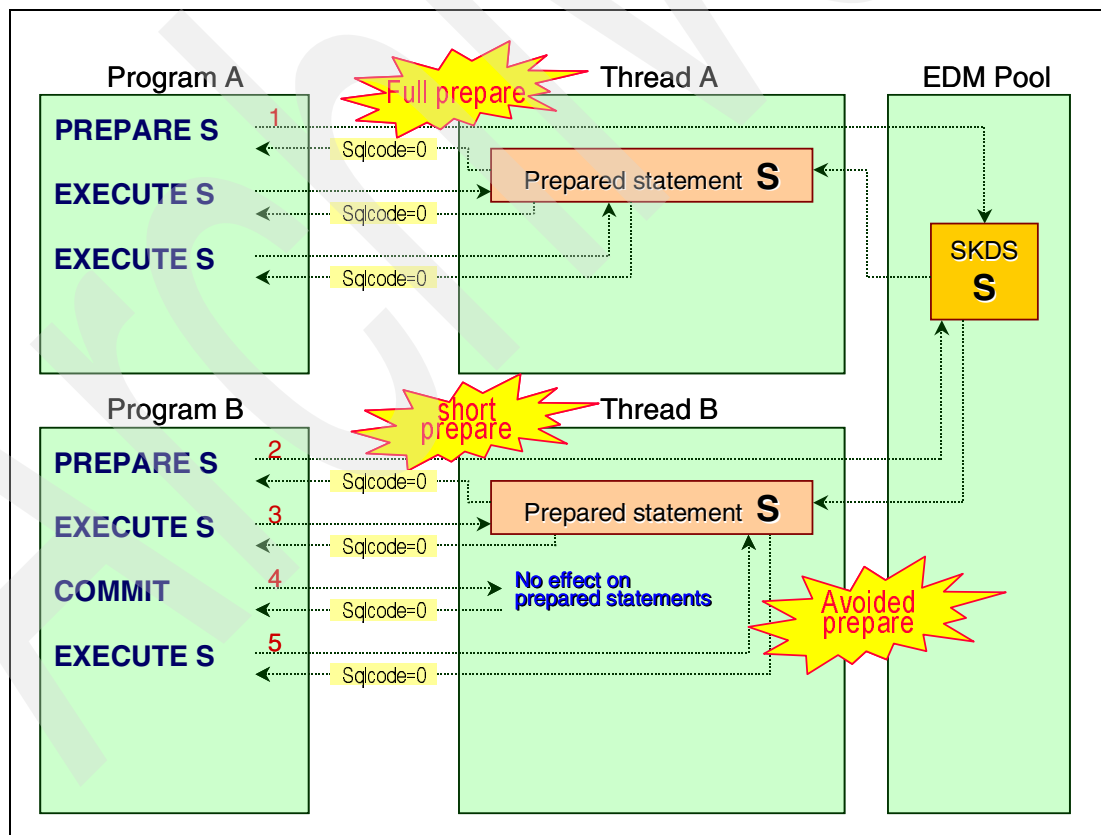


Figure 8-4 Full statement caching

Using full caching the maximum size of the local cache across all user threads is controlled by the MAXKEEPD DSNZPARM. A FIFO algorithm is used for replacement of statements in the local cache.

## Different prepare types

When using statement caching, four different types of prepare operations can take place:

- Full prepare:** A full prepare occurs when the skeleton copy of the prepared SQL statement does not exist in the global dynamic SQL cache (or the global cache is not active). It can be caused explicitly by a PREPARE or an EXECUTE IMMEDIATE statement or implicitly by an EXECUTE when using KEEP DYNAMIC(YES).
- Short prepare:** A short prepare occurs, if the skeleton copy of the prepared SQL statement in the global dynamic SQL cache can be copied into the local storage.
- Avoided prepare** A prepare can only be avoided when using full caching. Since in this case, the full prepared statement is kept across commits, issuing a new EXECUTE statement (without a prepare after a commit) does not need to prepare anything. The full executable statement is still in the thread's local storage (assuming it was not removed from the local thread storage because MAXKEEPD was exceeded) and can be executed as such.
- Implicit prepare** This is the case when an application, that uses KEEP DYNAMIC(YES), issues a new EXECUTE after a commit was performed and a prepare cannot be avoided (the previous case). DB2 will issue the prepare (implicitly) on behalf of the application. (The application must not explicitly code the prepare after a commit in this case.)  
Implicit prepares can result in a full or short prepare:

- In full caching mode, when a statement has been removed from the local cache because MAXKEEPD was exceeded, but still exists in the global cache, the statement is copied from the global cache. This is a short prepare. (If MAXKEEPD has not been exceeded and the statement is still in the local cache the prepare is avoided.)
- In full caching mode, and the statement is no longer in the global cache either, a full prepare is done.
- In local caching only mode, a full prepare has to be done.

Whether a full or short prepare is needed in full caching mode depends on the size of the global cache. The bigger the size, the more likely we can do a short prepare.

Comparing the relative cost of the different types of prepare:

- ▶ If a full prepare costs 100
- ▶ A short prepare costs 1
- ▶ And an avoided prepare costs nothing

As mentioned before, an implicit prepare can either be a full prepare (most likely) or a short prepare.

## Dynamic statement reuse

In order to get good performance from statement caching, it is important to achieve a high degree of matching statements. The dynamic statement cache is only used when:

- ▶ The statement given to DB2 is 100% the same (of the same length: trailing blanks are **not** removed!). However, some applications such as DSNTEP2 will take the SQL statement provided (in the SYSIN DD-card) and eliminate all the unnecessary blanks, before passing the statement to DB2, this way providing a better chance for a hit in the statement cache. As a result of this, in cases where you want to force DB2 to determine a new access path

for a statement instead of using a cached version, adding a few blanks to the statement is not enough to trigger a complete prepare process.

- ▶ At prepare the same ATTRIBUTES have been used. The values in the ATTRIBUTES clause are concatenated to the statement string before comparing the strings.
- ▶ The statement does not come from private protocol source (deferred embedded dynamic SQL). Statements can come from any other source (only SELECT, UPDATE, INSERT, DELETE).
- ▶ The authorization ID that was used to prepare the statement that and put it into the cache must be the same as the authorization id that is used to retrieve the statement from the cache for reuse.
  - 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.
- ▶ Plan or package are bound with the same parameters (CURRENTDATA, DYNAMICRULES, ISOLATION, SQLRULES, and QUALIFIER).
- ▶ At prepare time the special registers CURRENT DEGREE, CURRENT RULES, CURRENT PRECISION, CURRENT PATH and CURRENT OPTIMIZATION HINT are the same.
- ▶ Declared cursor characteristics: HOLD, STATIC, SENSITIVE, INSENSITIVE, SCROLLABLE should match.
- ▶ Parser options must be the same. APOST or QUOTE delimiter, PERIOD or COMMA decimal delimiter, date and time format DECIMAL options. This comes into play when DYNAMICRULES(BIND) and DYNAMICRULES(RUN) programs are run in the same system.
- ▶ Was parallelism disabled by RLF? If parallelism was disabled by RLF for the thread that put the statement in the cache, but not for the thread that is searching the cache, then there is no match.

For example, avoiding dynamic statement preparation requires that the statement in question is an exact character-for-character duplicate of a previously prepared statement. Therefore, to obtain a better statement reuse, it is recommended to replace constants (literals) used in the SQL statements by parameter markers. On the other hand the usage of parameter markers can be a disadvantage for the optimizer when searching the best access paths since the actual data values are not know at prepare time.

## Dynamic statement invalidation

In DB2 version 7 cached statements get invalidated or removed from the *global cache*, when:

- ▶ No free pages are available in the EDM pool. The SKDSs are discarded on an LRU basis. (Local copies are not affected.)
- ▶ DROP, ALTER, REVOKE executed for anything the plan is depending on.
- ▶ RUNSTATS executed for objects, the statement is depending on. RUNSTATS is the most efficient way to intentionally invalidate cached statements. Use the UPDATE NONE clause to invalidate statements without updating the catalog.

In DB2 version 7 cached statements (executable form of the statements) get invalidated or removed from the *local cache*, when:

- ▶ MAXKEEPD is exceeded. Locally cached statements are discarded on a FIFO basis. This threshold is not strictly enforced. The total number of locally cached statements fluctuates around this number.
- ▶ The statement ID is reused by a prepare operation.
- ▶ The thread deallocates.
- ▶ DROP, ALTER, REVOKE executed for anything the plan is depending on.
- ▶ ROLLBACK and re-signon executed.
- ▶ RUNSTATS executed for objects, the statement is depending on. RUNSTATS is the most efficient way to intentionally invalidate cached statements. Use the UPDATE NONE clause to invalidate statements without updating the catalog.

In DB2 version 7 cached statements (statement strings) get invalidated in local storage when:

- ▶ The statement ID is reused by a prepare operation.
- ▶ The thread deallocates.
- ▶ ROLLBACK and re-signon executed.

**Attention:** The cached statement is **not** invalidated, when you change statistical columns in the catalog via UPDATE statements.

Therefore, if you use software manage or migrate catalog statistics from one subsystem to another, or software which collects statistical information about DB2 objects outside of DB2 and updates the DB2 catalog afterwards, you will not invalidate dependent statements in the global cache.

**Tip:** In DB2 version 7, a CREATE INDEX does not invalidate cached statements which reference the table on which the index was created in a non-data sharing environment. This might change in a future release of DB2. In a data sharing environment however, a CREATE INDEX invalidates all related cached statements.

## DB2 BIND options and statement caching

There are a few BIND options that behave different when dynamic statement cache is active:

- ▶ REOPT(VARS) applications can run in the CACHEDYN=YES systems, but their statements are not cached. But REOPT(VARS) is incompatible with KEEP\_DYNAMIC(YES) and produces a bind error.
- ▶ On systems where statement caching is not or only partially used, the RELEASE(DEALLOCATE) bind option does not apply to dynamic SQL statements. The locks acquired by dynamic SQL statements are released at commit. If full caching is used, RELEASE(DEALLOCATE) applies for dynamic SQL as well.

## Statement caching in a distributed environment

If the requester does not issue a PREPARE after a COMMIT in a distributed environment, the package at the DB2 for z/OS server must be bound with KEEP\_DYNAMIC(YES). If both requester and server are DB2 for OS/390 and 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 may have performance implications for DRDA clients that use WITH HOLD cursors:

- ▶ If KEEP\_DYNAMIC(NO) is specified, a separate network message is required when the DRDA client issues the SQL CLOSE for the WITH HOLD cursor.
- ▶ If KEEP\_DYNAMIC(YES) is specified, the DB2 for z/OS server automatically closes the cursor when SQLCODE +100 is detected. This 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 can also reduce the duration of locks that are associated with the held cursor.

When a distributed thread has touched *any* package that is bound with KEEP\_DYNAMIC(YES), the thread cannot become inactive. You find more information about this subject in Chapter 14.2, “Reusing database access threads” on page 230.

## 8.2.5 Dynamic statement cache summary

In Figure 8-5 you find an overview of the advantages and disadvantages of the different levels of statement caching. We explained these facts earlier in this chapter in more detail.

	CACHEDYN No	CACHEDYN Yes
KEEP_DYNAMIC No	<ul style="list-style-type: none"> <li>– no skeletons cached in EDMP</li> <li>– only full prepares</li> <li>– no prepared statements kept across commits <sup>1</sup></li> <li>– no statement strings kept across commits <sup>3</sup></li> </ul>	<ul style="list-style-type: none"> <li>✓ skeletons cached in EDMP</li> <li>✓ only first prepare full; otherwise short prepares <sup>2</sup></li> <li>– no prepared statements kept across commits <sup>1</sup></li> <li>– no statement strings kept across commits <sup>3</sup></li> </ul>
KEEP_DYNAMIC Yes	<ul style="list-style-type: none"> <li>– no skeletons cached in EDMP</li> <li>– only full prepares</li> <li>– no prepared statements kept across commits <sup>1</sup></li> <li>✓ statement strings kept across commits - implicit prepares</li> </ul>	<ul style="list-style-type: none"> <li>✓ skeletons cached in EDMP</li> <li>✓ only first prepare full, otherwise short prepares <sup>2</sup></li> <li>✓ prepared statements kept across commits - avoided prepares <sup>4</sup></li> <li>✓ statement strings kept across commits - implicit prepares</li> </ul>

<sup>1</sup> unless withheld opened cursor  
<sup>2</sup> unless invalidated or LRU-out  
<sup>3</sup> SQL codes -514 and -518 expected  
<sup>4</sup> subject to MAXKEEPD > 0

Figure 8-5 Comparison of local, global, and full caching

You can exploit statement caching the best, if you use parameter markers in your SQL statements, and if you reuse the threads in your applications. ERP packages of various software companies benefit most from statement caching. For the average home-grown application program it is not so easy to fulfill the constraints for statement reuse and to benefit efficiently from the global statement cache.

From the discussion above, it might seem that dynamic statement caching is only goodness. While this is true to a large extent, there is always a price to pay. In the case of dynamic statement caching, especially when using full caching, the price to pay is storage.

Using a large number of active threads, that use full caching for a lot of statements, can introduce storage problems in the DBM1 address space. The system parameters that can help you control dynamic statement cache storage use can be found in 9.2, "Controlling storage usage of statement caching" on page 154.



## System controls for dynamic SQL

In this chapter we discuss the parameters controlling the following:

- ▶ Local thread storage
- ▶ EDM pool
- ▶ Other system parameters related to dynamic SQL

## 9.1 Introduction

When using dynamic statement caching extensively, the storage of the DBM1 address space may become a bottleneck, because local threads and the EDM pool share storage in the DBM1 address space. In this chapter we focus mainly on controlling storage usage.

In the last section of this chapter we discuss additional installation parameters (DSNZPARMs) related to dynamic SQL.

## 9.2 Controlling storage usage of statement caching

In this section we discuss how to control local and global cache storage usage.

### 9.2.1 Controlling local cache storage

Other than controlling the number of prepare statements executed and kept “in a prepared state” by the application itself, the amount of local storage used by locally cached dynamic statements can be controlled by means of the following DSNZPARMs.

#### **MAXKEEPD**

Locally cached statements can consume lots of DBM1 virtual storage (10 KB for a simple statement, per thread). MAXKEEPD limits the number of locally cached executables (prepared statements). It applies for full caching only. At each commit, threads check if MAXKEEPD has been reached and, if yes, they throw away some of their locally cached statements in the thread’s pool (FIFO method). The MAXKEEPD threshold is not strictly enforced, the total number of locally cached statements fluctuate around it (since the value is only evaluated at commit time. Therefore, if you have a large number of prepares occur on a system where applications are not well-behaved and do not commit frequently, MAXKEEPD can be exceeded before the “cleanup” occurs.)

For MAXKEEPD a range from 0 to 65535 statements are allowed. The default is 5000.

Using 0 for MAXKEEPD disables keeping statement executables across commit. Setting MAXKEEPD to 0 has no effect on the statement text.

Note that MAXKEEPD=0 is *not* the same as turning off the local cache (KEEPDYNAMIC(NO)). A plan or package using KEEPDYNAMIC(YES) running on a system with MAXKEEPD=0 can still benefit from the implicit prepare.

A suggested tuning approach is to incrementally reduce the size of the local dynamic statement cache by reducing MAXKEEPD, and increase the global statement cache in the EDM pool to compensate as necessary to make sure that the global cache hit ratio does not deteriorate. To pick an initial value for MAXKEEPD you have to analyze your virtual storage usage of the DBM1 address space. The recommendation is to start with MAXKEEPD set to 10000. (Assuming you use fairly simple SQL, a locally cached statement uses about 10KB of memory in DBM1. For 10000 statements, that is 100 MB. So you have to make sure that DBM1 can take 100MB of additional storage before you set MAXKEEPD to 10000). Then adjust it downwards based on monitoring virtual storage saved and local hit ratio. (More information on local cache hit ratio can be found in 12.4.3, “Monitoring dynamic SQL statements” on page 193.

The value for MAXKEEPD can be specified on the DSNTIPE installation panel as “MAX KEPT DYN STMTS”. See *DB2 Universal Database for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936 for more details. MAXKEEPD cannot be changed without stopping and starting the DB2 subsystem.



## CONTSTOR

Another parameter to control the thread's local storage usage is the CONTSTOR DSNZPARM.

If a DB2 subsystem uses many very long-running threads, the storage used in the DBM1 address space can gradually grow to a size approaching the MVS 2GB limit per address space. For example, some client/server applications establish a large number of connections to DB2 and never terminate them. These DB2 threads, assuming connection pooling is not used, can each accumulate large amounts of thread storage over their lifetime, especially if these threads use dynamic SQL in a full caching environment.

A primary storage pool is created for each of these threads. Storage needed to prepare and execute SQL is acquired from this pool. Over time, the amount of storage needed by the thread for all active SQL statements grows and shrinks. Blocks of storage which are no longer needed, are marked free, but they are not freed back to the operating system. They remain part of that storage pool and are available to satisfy later requests from the pool.

If the CONTSTOR parameter in DSNZPARM is specified as YES, DB2 will examine each thread when it reaches a commit point. DB2 will then look at two criteria to decide whether to contract the thread's storage pool. If one of them is true, storage pool contraction is initiated.

- ▶ The number of commits has exceeded 50 (parameter &SPRMCTH) since the last time the pool was contracted
- ▶ The total size of the thread's storage pool exceeds 2 MB (parameter &SPRMSTH)

(Under the guidance of IBM Service, the contraction threshold values, 50 commits, and 2MB of storage, can be changed, if necessary, to achieve an optimal balance between storage reduction and performance.)

Note that CONTSTOR applies to the thread's local storage usage in the DBM1 address space. It has no effect on the EDM pool usages of SKDSs.

The usage of CONTSTOR can be specified on the DSNTIPE installation panel as "CONTRACT THREAD STG". See *DB2 Universal Database for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936 for more details. CONTSTOR is an online DSNZPARM.

## MINSTOR

Long-running threads in DB2 can consume excessive amounts of storage in the DBM1 address space. This storage can be fragmented because of DB2's internal storage management. Some relief is available by enabling periodic contraction of those storage pools to free unused blocks of storage back to the operating system (See, "CONTSTOR" on page 155). However, CONTSTOR does not address the problem of fragmentation within the blocks of storage. Often, much of this accumulated storage is actually unused, due to storage fragmentation, and the lack of a garbage collection. Therefore, long-running threads can still have a high percentage of their allocated working storage unused because of the intra-block fragmentation. DB2 algorithms for allocating and freeing storage within these storage blocks are optimized for performance, not for the most efficient use of storage, resulting in fragmentation over time.

The APAR PQ37894 introduced an enhancement which allows a subsystem to manage DB2 thread working storage with objective of minimizing the amount of used storage. A new subsystem parameter (DSNZPARM) named MINSTOR is added to control whether DB2 storage minimizing algorithms should be used. The default for the parameter is NO. The use of this parameter is good for reducing storage usage but can come with a potential performance penalty. If you are not constraint on virtual storage usage in the DBM1 address space, then there is *no* need to set this DSNZPARM to YES.

The usage of MINSTOR can be specified on the DSNTIPE installation panel as “MANAGE THREAD STORAGE”. See *DB2 Universal Database for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936 for more details. MINSTOR is an online DSNZPARM.

## 9.2.2 Controlling global cache storage (EDM pool)

In this section we discuss how to control global cache storage usage.

### Sizing the global statement cache

Global dynamic statement caching has considerable effect on the size of your EDM pool. If you use global dynamic statement caching extensively, a large EDM pool is required in order to achieve a satisfactory global cache hit ratio. Some sites which make heavy use of dynamic statement caching have multi hundred-megabyte EDM pools. Of course, the EDM pool is not the only occupant of the extended private portion of the DB2 database services (DBM1) address space.

The EDM pool contains:

- ▶ CTs, PTs, and DBDs in use
- ▶ SKCTs for the most frequently used applications
- ▶ SKPTs for the most frequently used applications
- ▶ DBDs referred by these applications
- ▶ Cache blocks for plans that have caches
- ▶ Skeletons (SKDS) of the most frequently used dynamic SQL statements

The size of the EDM pool is calculated during the installation process (DSNTINST CLIST). When calculating the size of the global statement cache, you must consider two types of applications issuing dynamic SQL statements:

- ▶ Applications with repeatedly used statements. These application, like most ERP packages, can benefit most from dynamic statement caching.
- ▶ Ad-hoc applications with seldom repeated statements. These applications like QMF do not really benefit from dynamic statement caching.

You can calculate the needed size of the global statement cache as follows:

$$\text{cachesize} = (\text{maxdynpl} * (\text{dynssize} * \text{dynpno}) + (\text{dynssize} * \text{adhno}))$$

In this calculation:

<b>maxdynpl</b>	The expected maximum number of unique plans containing embedded dynamic SQL statements.
<b>dynpno</b>	The expected number of different dynamic SQL statements executed by plans containing dynamic SQL statements which are likely to be used repeatedly.
<b>adhno</b>	The expected maximum number of dynamic SQL statements which are likely to be used only once.
<b>dynssize</b>	The average size of prepared statement executed by those plans. calculate 1.8kB for each single table statement and add 0.2kB for each additional table.

## Using data space for global statement caching

Your site may have little room for EDM pool enlargement due to the combined size of the DB2 buffer pools, RID pool, sort pool, and compression dictionaries (64kB per open data set defined with COMPRESS YES). In that case, if you are using DB2 for OS/390 version 6 or above, DB2 allows you put the part of the EDM pool which is used for dynamic statement caching into a data space.

If you use a data space for the part of the EDM pool used for global dynamic statement caching, the statement skeletons (SKDSs) are kept in the data space whereas other control blocks are kept in the EDM pool, as shown in Figure 9-1.

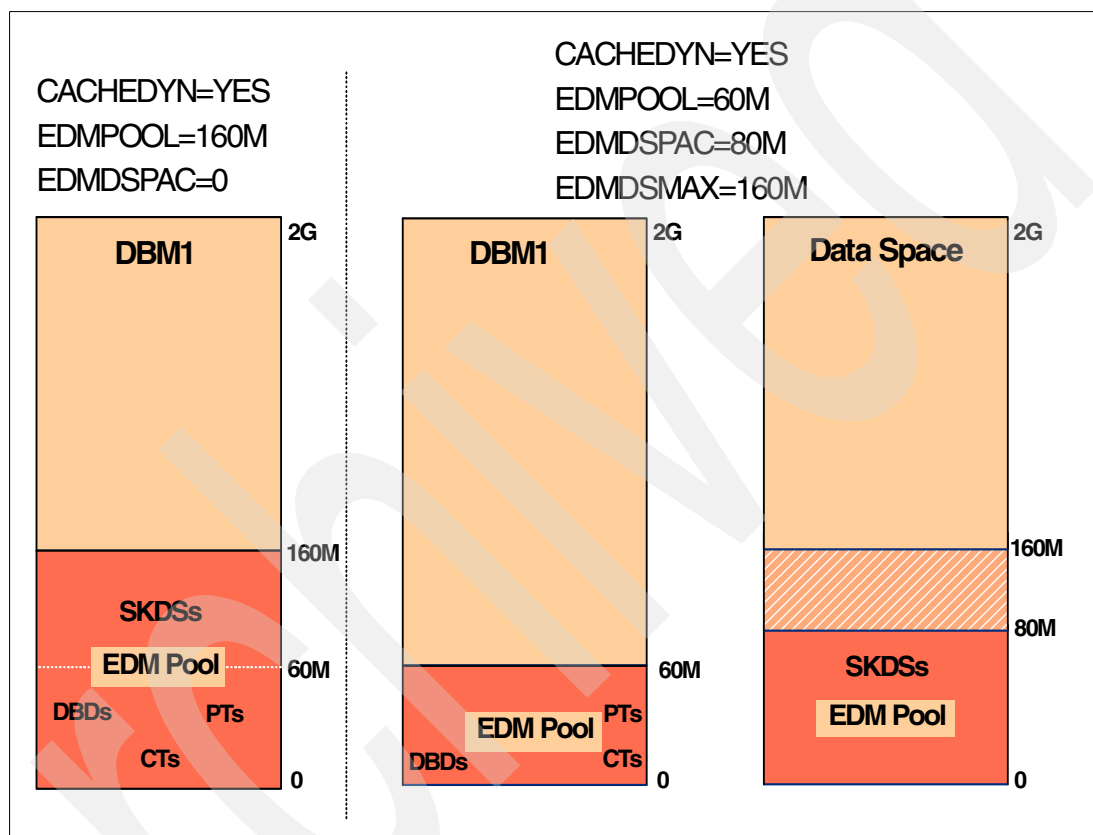


Figure 9-1 Using data space for global dynamic statement caching

If you use a data space for statement caching (DSNZPARM EDMSPAC > 0), the size of the data space can be changed online up to the maximum size you specified in DSNZPARM EDMDSMAX. These values are calculated by the DB2 installation CLIST (DSNTINST) but can be overwritten on DSNTIPC installation panel as “EDMPOOL DATA SPACE SIZE” for EDMSPAC and “EDMPOOL DATA SPACE MAX” for EDMDSMAX. See *DB2 Universal Database for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936 for more details.

A data space for global dynamic SQL caching should be used only, if you are close to the 2 GB addressability limit for DBM1 address space. Make sure that the data space is backed by real storage in order not increase the system’s paging activity significantly.

Performance measurements showed that the DB2 CPU cost for allowing the dynamic statement cache to use a data space is 1% for the measured workload, when backed by real storage. For more information see *DB2 UDB for OS/390 Version 6 Performance Topics*, SG24-5351.

The Version 6 APAR PQ25914 (PTF UQ29514) allows data spaces and virtual buffer pools to be backed by real storage above 2GB (also called 64 bit real addressing).

Putting the dynamic statement cache (DSC) in a data space stops it from clobbering other users ((SK)CTs, (SK)PTs, and DBDs) of the EDM Pool. DSC is more aggressive at grabbing EDM pool space than other users are. It allocates as much space as is available to store SKDSs, potentially pushing out other EDM pool “inhabitants”. By giving the DSC its own space this can be avoided.

## Controlling the DBD size

Another aspect of controlling the EDM pool size is the optimization of the size of the DBDs. This is especially important for some ESP packages that only use a few databases with a large number of tables in each database.

The DBD size directly depends on the number of tables in the database. A reasonable minimum size of a DBD is 12KB, but for database with 50 and more tables and an average of 200 columns per table the DBD can easily grow over 150 KB. Isolating regularly used tables into their own table spaces and databases downsizes the used DBDs and ensures in this way an optimal utilization of the EDM pool.

Since Version 6, a DBD no longer needs to be stored in one contiguous piece of storage in the EDM pool. DBDs can now be broken into chunks of maximum 32KB. However, this should not be an excuse for not running the REORG and MODIFY utility on a regular basis to reclaim space, of OBDs representing dropped objects, in the DBD. For more information see *DB2 UDB for OS/390 Version 6 Performance Topics*, SG24-5351.

## EDMBFIT

Because requests for EDM storage are variable in length, DB2 may need to free multiple objects to satisfy one large request (for example to chunk a DBD). If the multiple pieces of the same object types are in use and not able to be freed, DB2 fragments the EDM pool and reduces the largest contiguous storage areas available for a new request.

DSNZPARM EDMBFIT (APAR PQ31969) introduces a new free chain search algorithm for large EDM pools (> 40 MB). You can choose between two methods:

- ▶ First fit for best performance
- ▶ Best fit for optimal storage utilization

EDMBFIT helps to reduce the fragmentation of storage by grouping together those pieces which belong to the same plan, package, or DBD, and those pieces with like duration and types.

EDMBFIT is most helpful when commonly used objects are exceptionally large, for example, large DBDs or a packages with many statement sections. If all of the object pieces are small, the benefit is less.

EDMBFIT can affect performance when pool definitions are very large (> 400 MB). The overhead of the algorithm can result in longer latch wait time for very short running transactions which are “EDM pool bound” and use just a few sections of a plan or package.

In general, using EDMBFIT is good to tune EDM storage-constrained systems. EDMBFIT does not affect how DB2 reclaims storage, but it does affect where DB2 allocates storage.

The usage of EDMBFIT can be specified on the DSNTIP4 installation panel as “LARGE EDM BETTER FIT”. See *DB2 Universal Database for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936 for more details.

## 9.3 Other DSNZPARMs

In this section we mention other DSNZPARMs related to dynamic SQL. The first group of DSNZPARMs are parameters related to the *Resource Limit Facility* (RLF). Secondly, we explain the default DSNZPARM for the DYNAMICRULES option.

### 9.3.1 Resource Limit Facility (RLF)

At installation time you can specify defaults to control the *Resource Limit Facility* (RLF or *governor*). These parameters (DSNZPARMs) are:

<b>RLF</b>	Defines whether the resource limit facility is automatically started when DB2 is started.
<b>RLFAUTH</b>	Specifies the default authorization ID for the resource limit facility.
<b>RLFTBL</b>	Defines the default <i>resource limit specification table</i> (RLST) suffix. This suffix is used when the resource limit facility is automatically started or when it is started without specifying a suffix.
<b>RLFERR</b>	Specifies the default action for local users, if the RLST is not accessible for the governor, or if the governor cannot find a row in the RLST which applies to the authorization ID, the plan or package name, or the remote system name. The possible values are: <b>NOLIMIT</b> Allows all dynamic SQL statements to run without limit. <b>NORUN</b> Terminates all dynamic SQL statements. <b>nn</b> Allows all dynamic SQL statements to use up to nn service units (SU), where nn is between 1 and 5000000.
<b>RLFERRD</b>	Specifies a default action for remote threads, if the RLST is not accessible for the governor, or if the governor cannot find a row in the RLST which applies to the authorization ID, the plan or package name, or the remote system name. The possible values are similar to those for RLFERR.

### 9.3.2 DYNAMICRULES

Via the DYNRULS DSNZPARM, you can specify whether DB2 should use the DSNHDECP application programming defaults (installation defaults) or the options that were specified at precompile time. This only applies for dynamic SQL statements, using DYNAMICRULES bind, define, or invoke behavior. (More information about bind, define and invoke behavior can be found in 7.4, "DYNAMICRULES option" on page 134.

When the DYNRULS DSNZPARM is set to YES, the DSNHDECP application programming defaults are used. The following defaults are affected:

- ▶ DECIMAL POINT
- ▶ STRING DELIMITER
- ▶ SQL STRING DELIMITER
- ▶ MIXED DATA
- ▶ DECIMAL ARITHMETIC

When DSNZPARM DYNRULS is set to NO, the values of the precompiler options are used.

For DB2 Version 7 the default is YES. In earlier versions the default was NO.

The usage of DYNRULS can be specified on the DSNTIPF installation panel as "USE FOR DYNAMICRULES". See *DB2 Universal Database for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936 for more details.

Archived

## Locking and concurrency

Concurrency is the ability to have more than one application process accessing the same data at essentially the same time. Concurrency must be controlled to prevent lost updates and undesirable effects, such as access to uncommitted data or unrepeatable reads. In DB2, locks are used to prevent those situations (unless they are specifically allowed by the user or the program). In DB2, there are many different statements, parameters, and default behaviors that control locks and concurrency.

In this chapter we focus on those aspects of concurrency that are directly related to dynamic SQL. For more information about concurrency and locking in general, refer to *DB2 Universal Database for OS/390 and z/OS Version 7 Administration Guide*, SC26-9931.

## 10.1 DBD locks

Without statement caching, dynamic DML SQL statements acquire an S-lock on the database descriptor (DBD), when the statement is prepared. This prevents DDL from executing in the database which has been referenced by the dynamic DML statement. The lock is held until the thread reaches a commit point.

When using global statement caching (CACHEDYN=YES), no DBD lock is acquired for *short prepares* (retrieving a statement from the cache). APAR PQ24634 extends this behavior to *full prepares* (inserts into the cache) for SELECT, UPDATE, or DELETE statements (DML), if statement caching use has not been prohibited for another reason. Those other reasons include the use of the REOPT(VARS) bind option, or if DDL was performed by the application process in the same unit of work.

## 10.2 Ambiguous cursors

In this section we describe the various types of cursors and the meaning of an “ambiguous” cursor.

### 10.2.1 What is an ambiguous cursor?

DB2 differentiates between three types of cursors:

- ▶ Read-only cursors
- ▶ Updatable cursors
- ▶ Ambiguous cursors

For *ambiguous cursors*, DB2 cannot decide whether these cursors are read-only or may be updatable.

A cursor is considered *ambiguous*, when DB2 cannot decide, if these cursors are read-only or may be updatable. A cursor is ambiguous if:

- ▶ It is not defined with the clauses:
  - FOR FETCH ONLY or FOR READ ONLY. If these clauses are present, it is a *read-only* cursor.
  - FOR UPDATE OF. This makes the cursor *updatable*.
- ▶ It is not defined on a read-only result table (which would make it a *read-only* cursor).
- ▶ It is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement. (This would make the cursor *updatable*.)
- ▶ It is in a plan or package that contains the SQL statements PREPARE or EXECUTE IMMEDIATE:
  - The presence of any dynamic SQL causes an ambiguous cursors, because DB2 cannot detect what follows in the program.
  - A cursor can appear to be read-only, but a dynamic SQL statement could modify data through the cursor. Therefore the cursor is ambiguous.



## 10.2.2 Using the bind option CURRENTDATA

The CURRENTDATA option determines the behavior of ambiguous cursors. It has different effects, depending on whether the access is local or remote:

### Local access

For *local* access, the CURRENTDATA tells whether the data your cursor is positioned upon must remain identical to (or “current with”) the data in the local base table. This effect only applies to read-only or ambiguous cursors in plans or packages bound with CS isolation.

CURRENTDATA(YES) means that the data the cursor is positioned on, cannot be changed while the cursor is positioned on it. If the cursor is positioned on data in a local base table, then the data returned with the cursor is current with the contents of that table.

CURRENTDATA(YES) can guarantee that the cursor and the base table are current, CURRENTDATA(NO) does not.

For parallelism on read-only queries, the CURRENTDATA option has no effect. If a cursor uses query parallelism, data is not necessarily current with the contents of the table.

If you are using parallelism but want to maintain currency with the data, you have the following options:

- ▶ Disable parallelism (Use SET CURRENT 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).

If you use CURRENTDATA(NO) to indicate that an ambiguous cursor is read-only, you will get an error, when it is actually targeted by dynamic SQL for modification (sqlcode -510, SQLSTATE 42828).

### Remote access

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 (CURRENTDATA(NO)) or off (CURRENTDATA (YES)). Turning on block fetch offers better performance, but it means the cursor is not current with the base table at the remote site.

Read-only cursors or UR isolation level always use block fetch.

When 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.

### Lock avoidance

With CURRENTDATA(NO), you have a better opportunity to avoid locks. DB2 can try to recognize if a row or a page only 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.

To take the best advantage of this method of avoiding locks, all applications that are accessing data concurrently should issue COMMITs frequently.

Table 10-1 shows the lock avoidance factors. “Returned data” in the third column means data which satisfies the predicate. “Rejected data” in the fourth column means data which does not satisfy the predicate.

Table 10-1 Lock avoidance factors

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
		Updatable		
		Ambiguous		
	NO	Read-only	Yes	
		Updatable	No	
		Ambiguous	Yes	
RS	N/A	Read-only	No	Yes
		Updatable		
		Ambiguous		
RR	N/A	Read-only	No	No
		Updatable		
		Ambiguous		

## 10.3 Bind option RELEASE(DEALLOCATE)

The bind options ACQUIRE and RELEASE determine, when DB2 acquires and releases locks a table space, partition or a table which your application uses. These locks are sometimes called *gross* locks. The options apply only to static SQL statements, which are bound before your program executes. They do not affect dynamic SQL statements. Their locks are released at commit.

But there is one exception. On subsystems where *full caching* is used, RELEASE(DEALLOCATE) also applies for dynamic SQL. When full caching is active, DB2 retains dynamically 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 gross locks are held until deallocation, or until the commit after the prepared statement is freed from memory. Refer to Chapter 8.2.4, “Full caching” on page 147, to determine when a statement is removed from the cache.

When using RELEASE(DEALLOCATE) in a full caching environment, DB2 will still try to demote the gross lock at commit time if possible. If an S-lock, SIX-lock or X-lock is held, DB2 tries to demote it to an IS-lock or IX-lock if possible.

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)

Demotion cannot occur if any of the following conditions exist:

- ▶ The table space has LOCKSIZE TABLESPACE
- ▶ The segmented table has LOCKSIZE TABLE
- ▶ The gross lock was acquired as a result of ISOLATION(RR)
- ▶ The gross lock was acquired because the table space was started for ACCESS(RO)
- ▶ The lock was held for static SQL

Furthermore, for partitioned table spaces using selective partition locking, lock demotion can only occur at the table space level, not the partition level. Lock demotion is tracked only at the table space and table level.

Remember that dynamic DML SQL acquires an S-lock on the DBD, when it is prepared, except when global statement caching is used. Refer to Section 10.1, “DBD locks” on page 162 for more information.

With RELEASE(COMMIT) the use count of the DBDs referenced by the application are decreased by one. DBDs are “stealable” if their use count is zero.

With RELEASE(DEALLOCATE) the use count is only decreased when the thread terminates. Therefore, it takes significantly longer before the use count of a DBD reaches zero and becomes “stealable”.

Otherwise a DBD is only “stealable” if for all statements which referenced the DBD the statement name is reused by statements that do not use this DBD. If DBDs are kept longer, they may eat the EDM pool. Refer to 9.2.2, “Controlling global cache storage (EDM pool)” on page 156 for more information.

With RELEASE(DEALLOCATE) only if for all the statements that referenced the DBD the statement name is reused by statements that do not use that DBD.

Rarely committing transactions are not bad only for concurrency; they squeeze out SKDSs (most volatile object) from the EDM pool. This applies to all types of transactions, not only updating ones. Therefore, even though it may sound strange, ensure that read-only transactions also regularly commit.

A little-known effect of RELEASE(DEALLOCATE) that applies to all packages, whether they contain static or dynamic SQL, regardless of the KEEP\_DYNAMIC option, is that the package itself is kept by the thread at commit. That is, the copy of the PT (package table) in the EDM pool, owned by this thread, is not released. That means that on the first SQL in the next unit of work, the thread has to go back to EDM to get a new copy. This is normally not a big deal, since the corresponding SKPT will almost always be in the EDM pool already. It can however become a performance problem for application using ODBC or JDBC that specify autocommit=yes, committing after every statement.

Archived



## Governing dynamic SQL

In this chapter we explain how to control dynamic SQL statements by:

- ▶ Predictive governing
- ▶ Reactive governing
- ▶ Combining predictive and reactive governing

## 11.1 Introduction

Controlling the resource usage of the individual applications is an important issue to guarantee good overall performance. With OS/390 and z/OS you have several facilities inside or outside DB2 to control resource usage. For example:

- ▶ Time limit on job (through JES or JCL parameters) to limit resources for each job or step
- ▶ Time limit for TSO logon to limit resources for TSO sessions
- ▶ IMS and CICS controls to limit resources for IMS and CICS transactions
- ▶ MVS *Workload Manager* (WLM) definitions to prioritize resources
- ▶ The *Resource Limit Facility* (RLF) for predictive and reactive governing of dynamic SQL statements in DB2
- ▶ The Governor for reactive controlling in the *Query Management Facility* (QMF)
- ▶ The ASUTIME column of SYSIBM.SYSROUTINES catalog table to limit resources for a stored procedure
- ▶ RMF and DB2 accounting information to evaluate resource usage

In this Redbook we focus on the possibilities to control the resource usage of dynamic DML SQL statements inside the DB2 engine via the *Resource Limit Facility* (RLF), also known as the DB2 *Governor*. This by no means implies that the other knobs to control resource usage are inadequate to control dynamic SQL resource usage. RLF only provides an extra knob specially geared towards dynamic SQL. For more information on the non-DB2 knobs, please refer to Section 5.5.1 of *DB2 Universal Database for OS/390 and z/OS Version 7 Administration Guide*, SC26-9931.

RLF allows you to limit the resources used by running dynamic SQL statements (reactive governing), and to prohibit the execution of dynamic SQL statements which cost estimates exceed your thresholds at prepare time (predictive governing).

### The definition of resource limit specification tables (RLSTs)

You define your limits and thresholds in *Resource Limit Specification Tables* (RLSTs), which must be named `authid.DSNRLSTnn` (with *nn* any alphanumeric characters, which can be specified in a START RLIMIT command). To create such a table use the format shown in Example 11-1:

*Example 11-1 DDL to create an RLST*

---

```
CREATE TABLE authid.DSNRLSTnn
  (AUTHID   CHAR(8)      NOT NULL WITH DEFAULT,
   PLANNAME CHAR(8)      NOT NULL WITH DEFAULT,
   ASUTIME  INTEGER,
   -----3-column format -----
   LUNAME   CHAR(8)      NOT NULL WITH DEFAULT,
   -----4-column format -----
   RLFFUNC  CHAR(1)      NOT NULL WITH DEFAULT,
   RLFBIND  CHAR(1)      NOT NULL WITH DEFAULT,
   RLFCOLLN CHAR(18)     NOT NULL WITH DEFAULT,
   RLFPKG   CHAR(8)      NOT NULL WITH DEFAULT),
   -----8-column format -----
   RLFASUERR INTEGER,
   RLFASUWARN INTEGER,
   RLF_CATEGORY_B CHAR(1) NOT NULL WITH DEFAULT)
   -----11-column format -----
  IN DSNRLST.DSNRLSnn;
```

---

It is recommended to create these tables in their own database.

The column RLFFUNC determines which function is performed for a particular row. The values for RLFFUNC and the limiting columns ASUTIME, RLFASUERR, RLFASUWARN, or RLF\_CATEGORY\_B are explained in the following sections of this chapter.

AUTHID, PLANNAME, LUNAME, RLFCOLLN, and RLFPKG are the identifying columns, which help you to specify the sources of the dynamic SQL statements which you want to control.

Not all columns in a particular row are populated. The columns which you must populate depend on which function (RLFFUNC) is performed by that row and how you want to qualify values. For example, you can qualify broadly by leaving the AUTHID column blank. This means that the row applies to all authorization IDs. You can also qualify very narrowly by specifying a different row for each authorization ID for which the function applies.

A detailed description of the RLST columns can be found in chapter 5 of *DB2 Universal Database for OS/390 and z/OS Version 7 Administration Guide*, SC26-9931.

To create an index for the 11-column format table, use the SQL from Example 11-2:

*Example 11-2 Creating the RLST index*

---

```
CREATE UNIQUE INDEX authid.DSNARLnn
ON authid.DSNRLSTnn
(RLFFUNC, AUTHID DESC, PLANNAME DESC,
RLFCOLLN DESC, RLFPKG DESC, LUNAME DESC)
CLUSTER CLOSE NO;
```

---

It must be a descending index. The nn in the index name (DSNARLnn) must match the nn in the table name (DSNRLSTnn), otherwise you cannot issue the -START RLIMIT command successfully.

You activate the definition in a table with the command:

```
START RLIMIT ID=nn
```

In this command, *nn* refers to the individual characters in the table name DSNRLSTnn.

## How to handle processor service units

You specify your limits or thresholds in *processor service units*. The processing time for a particular SQL statement varies according to the processor on which it is executed, but the service units required remains roughly constant. Service units are independent of processor changes.

You can calculate the service unit (SU) time as follows:

$$\text{SU time} = \text{processor time} * \text{service units per second value}$$

The value of service units per second depends on the processor model. You can find it for your processor model in *z/OS V1R1.0 MVS Initialization and Tuning Guide*, SA22-7591.

The RLST values do not need to be modified when processors are changed. But the processing time allowed for your dynamic SQL statements changes with a change of a processor model, if you keep your RLST values constant.

For example, you define an RLST value of 58000 service units, and your processor has a value of 11600 service units per second. This allows a processing time of:

$$58000 \text{ service units} / 11600 \text{ service units per second} = 5 \text{ sec}$$

With a more powerful processor with a value of 14500 service units per second, you allow only a processing time of:

$58000 \text{ service units} / 14500 \text{ service units per second} = 4 \text{ sec}$

If you want to keep the allowed processing time constant, in order to exploit the more powerful processor, you must change your RLST threshold to:

$14500 \text{ service units per second} * 5 \text{ sec} = 72500 \text{ service units}$

For detailed information about how to calculate service units, see “Calculating service units” in *DB2 Universal Database for OS/390 and z/OS Administration Guide*, SC26-9931.

## 11.2 Predictive governing

Using the predictive governor, DB2 can warn or prevent the execution of a query, if it determines that more CPU is needed to execute the statement than a user-defined limit.

### RLST definitions for predictive governing

To use predictive governing you must supply the following columns in your RLST table:

**RLFFUNC** Specifies how the row is used. The values that are used for predictive governing are:

- '6' The row predictively governs dynamic SELECT, INSERT, UPDATE, or DELETE statements by plan name.
- '7' The row predictively governs dynamic SELECT, INSERT, UPDATE, or DELETE statements by package or collection name.

**RLFASUERR** Used only for statements that are in cost category A. It contains the error threshold number of system resource manager processor service units allowed for a single dynamic SELECT, INSERT, UPDATE, or DELETE statement. If the predicted processor cost is greater than the error threshold, an SQLCODE -495 (SQLSTATE 57051) is returned to the application.

Other possible values and their effects are:

Null No error threshold

0 (zero) or negative value

All dynamic SELECT, INSERT, UPDATE, or DELETE statements receive SQLCODE -495 (SQLSTATE 57051).

**RLFASUWARN** Used only for statements that are in cost category A. It contains the warning threshold number of processor service units allowed for a single dynamic SELECT, INSERT, UPDATE, or DELETE statement. If the predicted processor cost is greater than the warning threshold, an SQLCODE +495 (SQLSTATE 01616) is returned to the application.

Other possible values and their effects are:

Null No warning threshold

0 (zero) or a negative value

All dynamic SELECT, INSERT, UPDATE, or DELETE statements receive SQLCODE +495 (SQLSTATE 01616).

Of course, the value for RLFASUWARN must be less than that for RLFASUERR. If this value is higher, the warning is never reported.



**RLF\_CATEGORY\_B** Tells the governor the default action to take when the cost estimate for a given statement falls into cost category B, which means that the predicted cost is indeterminate and probably too low. The definable values are:

blank	By default, prepare and execute the SQL statement.
Y	Prepare and execute the SQL statement.
N	Do not prepare or execute the SQL statement and return SQLCODE -495 (SQLSTATE 57051) to the application.
W	Complete the prepare, return SQLCODE +495 (SQLSTATE 01616), and allow the application to decide whether to execute the SQL statement or not.

Cost categories are DB2's way of differentiating estimates for which adequate information is available (category A) from those for which it is not (category B). DB2 puts a statement's estimate into cost category B when any of the following conditions exist:

- ▶ The statement has user-defined functions (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 the 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.

Everything that doesn't fall into category B belongs to category A. Because of the uncertainty involved, category B statements are also good candidates for reactive governing.

Figure 11-1 shows the relationship between the three columns, which control predictive governing.

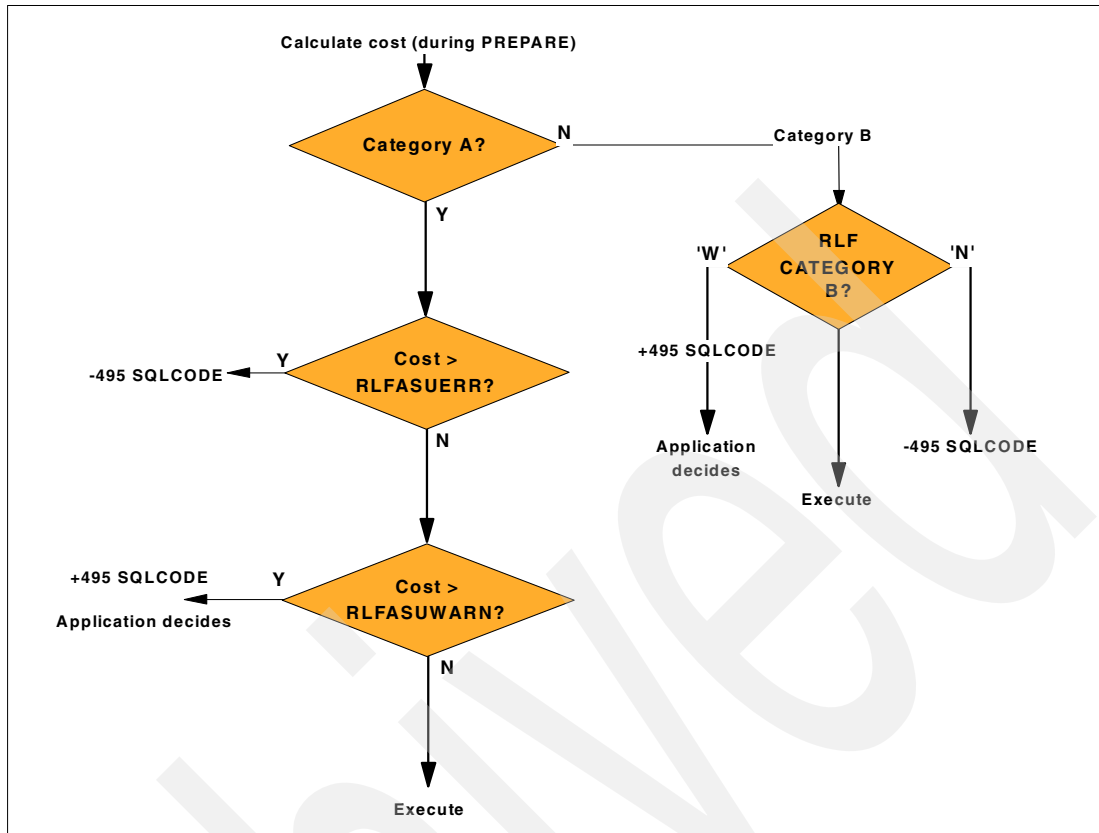


Figure 11-1 Logical flow of predictive governing

When predictive governing is in effect, the PREPARE of a dynamic SQL statement with an estimated cost that exceeds your RLFASUWARN defined threshold will return a warning code, giving you or your application the opportunity not to proceed and run the statement, or to ignore the warning and to continue running the statement.

If the estimated cost exceeds your defined RLFASUERR threshold, the prepare will return an error code, and there is no possibility to continue running this statement.

If you use predictive governing, your applications must check for the SQLCODEs +495 and -495 (SQLSTATEs 01616 and 57051) which predictive governing can generate after the execution of a prepare.

There are special considerations for this warning in combination with deferred prepares:

If the application requester uses deferred prepare, the presence of parameter markers determines when the application receives the SQLCODE +495 (SQLSTATE 01616). When parameter markers are present, DB2 cannot do PREPARE, OPEN, and FETCH processing in one message. If SQLCODE +495 (SQLSTATE 01616) is returned, no OPEN or FETCH processing occurs until your application requests it.

- ▶ If there are parameter markers, the warning code is returned on the OPEN.
- ▶ If there are no parameter markers, the warning code 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, you would ideally like to have the option to choose beforehand whether you want the OPEN and FETCH of the data to occur. For downlevel requesters, you do not have this option.

### Using predictive governing and downlevel DRDA requesters:

If SQLCODE +495 (SQLSTATEs 01616) 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 (SQLSTATEs 01616) is returned to the requester and OPEN processing does not occur.

If your application does defer prepare processing, the application receives the warning at its usual time (OPEN or PREPARE). If you have parameter markers with deferred prepare, you receive the warning at OPEN time as you normally do. However, an additional message is exchanged.

**Recommendation:** Do not use deferred prepare for applications which use parameter markers and which are predictively governed at the server side.

For applications bound with REOPT(VARS), DB2 does *not* return a warning SQLCODE at PREPARE, EXECUTE or OPEN time for a cost category B dynamic SQL statement and the action 'W' (warning) is specified in the RLST. The statement continues to execute. DB2 returns an SQLCODE -495 (SQLSTATE 57051) if the action is set to 'N' (no execution) at EXECUTE or OPEN time for category B statements.

### Example of predictive governing

In the following example we assume we are working with a processor with a capacity of 11600 service units per second. If you want to allow a dynamic SQL statement of user BARTR4, a processing time of 5 seconds without any RLF activity, and a processing between 5 and 10 seconds as the responsibility of the user, when he uses QMF, you should specify a row in your RLST as shown in Example 11-3:

Example 11-3 A definition for predictive governing

AUTHID	PLANNAME	ASUTIME	LUNAME	RLFFUNC	RLFBIND	RLFCOLLN	RLFPKG	RLFASUERR	RLFASUWARN	RLFCATEGORY_B
BARTR4		-		7		Q		116000	58000	N

According to the processor model and your time preferences, the row in Example 11-3 specifies a warning threshold RLFASUWARN of 58000 service units and an error threshold of 116000 service units for all packages (RLFFUNC = 7) in Collection Q (used by QMF) (column RLFCOLLN) used by user BARTR4 (column AUTHID). This means that BARTR4 gets an SQLCODE:

- ▶ Zero (0) if the estimated cost of this dynamic SQL statement prepared by a package from collection Q is less than 58000 service units
- ▶ +495 (SQLSTATE 01616) if the estimated cost of his dynamic SQL statement prepared by a package from collection Q exceeds 58000 service units, but is less than 116000 service units.
- ▶ -495 (SQLSTATE 57051) if the estimated cost of his dynamic SQL statement prepared by a package from collection Q exceeds 116000 service units.

In both cases the estimated cost must be of category A. For all statements of cost category B, BARTR4 receives an SQLCODE -495 (SQLSTATE 57051), because the value 'N' in column RLF\_CATEGORY\_B prohibits any execution.

# 11.3 Reactive governing

To guarantee a good overall performance in your DB2 subsystem, you might want to provide an automatic termination for long-running dynamic SQL statements or just in case something unexpected happens. You can use the reactive governing of the DB2 Resource Limit Facility (RLF) for this purpose. For reactive governing you can specify a limit on the amount of service units that a dynamic SQL statement can consume. A statement exceeding the limit will fail with a -905 SQLCODE (SQLSTATE 57014). You can provide a general limit for all dynamic SQL statements, or you can specify different limits according to plan, package name, collection ID, authorization ID, or remote system name.

When an application program receives SQLCODE -905 (SQLSTATE 57014), the application must determine what to do next.

- ▶ If the failed statement involves a cursor, the cursor's position remains unchanged. The application can CLOSE the cursor or a PREPARE command can be issued. All other operations with the cursor do not run and return the same -905 SQL error code.
- ▶ If the failed SQL statement does not involve a cursor, then all changes that the statement made are backed out before the error code is returned to the application. The application can either issue another SQL statement, including a commit for all the work done so far, or terminate.

## RLST definitions for reactive governing

To use reactive governing you must supply the following columns in your RLST:

<b>RLFFUNC</b>	Specifies how the row is used. The values that have an effect for reactive governing are:	
blank	The row reactively governs dynamic SELECT, INSERT, UPDATE, or DELETE statements by plan name	
'1'	The row reactively governs bind operations	
'2'	The row reactively governs dynamic SELECT, INSERT, UPDATE, or DELETE statements by package or collection name	
'3'	The row disables query I/O parallelism and therefore does not really apply to reactive governing	
'4'	The row disables query CP parallelism and therefore does not really apply to reactive governing	
'5'	The row disables sysplex query parallelism and therefore does not really apply to reactive governing	
<b>ASUTIME</b>	Contains your limit for any single dynamic SELECT, INSERT, UPDATE, or DELETE statement Other possible values and their meanings are:	
	Null	No limit
	0 (zero) or a negative value	No dynamic SELECT, INSERT, UPDATE, or DELETE statements are permitted.

## Example of reactive governing

Continuing with Example 11-3 on page 173 we are still working with a processor with a capacity of 11600 service units per second. If you want to allow an executing dynamic SQL statement of user BARTR4 to use a maximum processing time of 11 sec. in QMF, you should define the following row for your RLST as shown in Example 11-4.

*Example 11-4 A definition for reactive governing*

AUTHID	PLANNAME	ASUTIME	LUNAME	RLFFUNC	RLFBIND	RLFCOLLN	RLFPKG	RLFASUERR	RLFASUWARN	RLF_CATEGORY_B
BARTR4		127600		2		Q		-	-	

According to your processor model and your execution time preference, the row in Example 11-4 sets the ASUTIME threshold to 127600 service units for all packages (RLFFUNC = 2) in Collection Q (column RLFCOLLN) used by user BARTR4 (column AUTHID). This means that BARTR4 gets an SQLCODE:

- ▶ 0 if the cost of his dynamic SQL statement executed by a package from collection Q is less than 127600 service units
- ▶ -905 (SQLSTATE 57014) if the cost of his dynamic SQL statement executed by a package from collection Q exceeds 127600 service units.

## 11.4 Combining predictive and reactive governing

You can combine predictive and reactive governing. Statements which passed the predictive governing with or without warning can be terminated by reactive governing, when they exceed your defined thresholds for reactive governing. Especially statements which have been put into cost category B by the DB2 optimizer, should be reactively governed, if they are executed. Figure 11-2 shows you the integrated control flow of predictive and reactive governing.

The reactive governor does not differentiate between cost category A and cost category B statements, because reactive governing controls the number of service units that have actually been used and not the estimated number of service units.

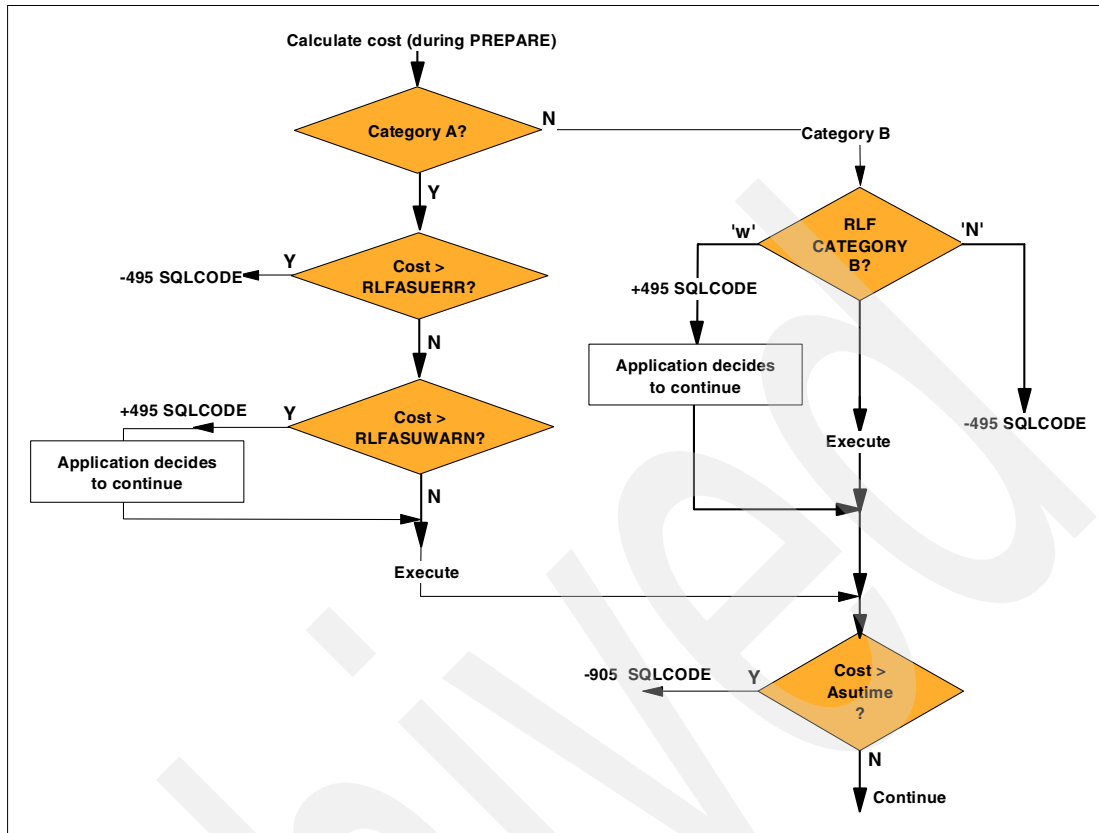


Figure 11-2 Logical flow of combined predictive and reactive governing

If you want to analyze the warning SQLCODE +495 (SQLSTATE 01616) in your application, in order to decide if your application should continue, you find the estimated service units, the cost category, and the warning threshold in the field SQLERRMC of the SQL communication area (SQLCA) shown in Example 11-5:

**Example 11-5** *SQLERRMC after SQLCODE +495*

```

Sample C program: DYNAMIC
PREP error: 495
Error Messg: 000000000000 000000000330 A 000000000010
Error Point: DSNXEDS1
  
```

The sequence of values in SQLERRMC is according to their sequence in the complete warning message as can be seen in Example 11-6:

**Example 11-6** *Full SQLCODE +495 error message*

```

+495  ESTIMATED PROCESSOR COST OF estimate-amount1 PROCESSOR SECONDS
      (estimate-amount2 SERVICE UNITS) IN COST CATEGORY cost-category
      EXCEEDS A RESOURCE LIMIT WARNING THRESHOLD OF limit- amount SERVICE
      UNITS
  
```

In the above example you find in the SQLCA field SQLERRMC an estimated cost of 330 service units, belonging to cost category A, and a warning threshold of 10 service units.

**Note:** We choose such a low threshold just for demonstration purposes. It is not recommended for normal use.

According to Example 11-3 on page 173 we use a processor with a capacity of 11600 service units per second. If you want to combine the predictive governing of Example 11-3 on page 173 with the reactive governing of Example 11-4 on page 175, you should define the following rows for your RLST as shown in Example 11-7.

*Example 11-7 Combining predictive and reactive governing*

AUTHID	PLANNAME	ASUTIME	LUNAME	RLFFUNC	RLFBIND	RLFCOLLN	RLFPKG	RLFASUERR	RLFASUWARN	RLF_CATEGORY_B
BARTR4		-		7		Q		116000	58000	W
BARTR4		127600		2		Q		-	-	

According to your processor model and your time preferences the row in Example 11-7 sets a warning threshold RLFASUWARN of 58000 service units and an error threshold of 116000 service units for all packages (RLFFUNC = 7) in Collection Q (column RLFCOLLN) used by user BARTR4 (column AUTHID). This means that BARTR4 gets a SQLCODE

- ▶ +495 (SQLSTATE 01616) if the estimated cost of his dynamic SQL statement prepared by a package of collection Q exceeds 58000 service units, but is less than 116000 service units.
- ▶ -495 (SQLSTATE 57051) if the estimated cost of his dynamic SQL statement prepared by a package of collection Q exceeds 116000 service units.

In both cases the estimated cost must be of category A. For costs of category B BARTR4 receives in any case a SQLCODE +495 (SQLSTATE 01616) because the value "W" in column RLF\_CATEGORY\_B. User BARTR4 or his application program can decide whether to execute the statement or not. Note that we are more forgiving than in Example 11-3 on page 173, where we did not allow cost category B to run.

Statements with an estimated cost below 58000 service units are executed without any notification by RLF to the user.

Programs which passed predictive governing or for which the user decided to execute in spite of the warning, will be terminated, when they exceed the threshold ASUTIME of 127600 service units. This means that BARTR4 receives a SQLCODE -905 (SQLSTATE 57014) at that time.

If you define very low limits for predictive and reactive governing, the prepare itself of a complex statement may be terminated by reactive governing. In this special case predictive governing for the prepared statement cannot happen because we have not yet determined the estimated statement cost (as it is part of the prepare process).

In a subsystem using global dynamic statement caching, a short prepare may be successful in this case, but the full prepare of the same statement might be terminated by reactive governing before it can be prepared successfully. (The prepared statement must have been inserted into the global cache at a time where RLF has not been active or RLF had been started with another RLST containing higher thresholds).

If the workload of your DB2 subsystem differs between day and night (online and batch workload), you can use different RLSTs with different threshold specifications, and switch dynamically between these tables.

At installation time you can specify a default RLST which is used each time DB2 is restarted. You can also specify default actions, if the RLST is not accessible for the governor, or if the governor cannot find a row in the RLST which applies to the authorization ID, the plan or package name, or the remote system name. For more information see 9.3.1, "Resource Limit Facility (RLF)" on page 159.

## 11.5 Qualifying rows in the RLST

DB2 tries to find the closest match when determining limits for a particular dynamic statement in the RLST. In summary, the search order of matching is as follows:

1. Exact match
2. Authorization ID
3. Plan name, or collection name and package name
4. LU name
5. No row match

**Note:** When specifying rows in the RLST, you can either use a full name (for an AUTHID, PLANNAME, LUNAME, RLFCOLLN or RLFPKG indicating it applies to that specific name or specify a blank, indicating the row applies to everyone. It is not allowed to use generic names like "PLANA\*").



## Monitoring, tracing, and explaining dynamic SQL

In this chapter we discuss the performance management aspects of dynamic SQL using the DB2 PM for:

- ▶ Monitoring threads
- ▶ Monitoring EDM Pool and dynamic statement cache
- ▶ Monitoring and explaining SQL statements in the global cache
- ▶ Collecting trace information and analyzing the data

## 12.1 Introduction

In this chapter we discuss the performance management of applications using dynamic SQL. We observe the EDM pool and look into the global statement cache. We show as well, how cached statements can be explained, and how to obtain access path information for the prepared statements.

In contrast to static SQL statements dynamic SQL cannot be explained easily. For static SQL you can force developers to do explains with every precompile job. You can analyze and explain the statements in the packages easily. Using ODBC, JDBC, REXX or embedded dynamic SQL, the statements are prepared and executed dynamically and cannot be found in any package. As long as you know your SQL statement, you can explain it using the SQL EXPLAIN command. In some applications, like end user drivers, you even do not know exactly which statements will be executed dynamically. In this case the *Resource Limit Facility* (RLF) offers you the possibility to govern the resource usage, as explained in Chapter 11, “Governing dynamic SQL” on page 167.

Other possibilities to analyze a dynamic SQL statement are to explain it from the statement cache or to trace the access path information of its prepare by means of a DB2 trace.

In this redbook we use the *DB2 Performance Monitor* (DB2 PM) to show and analyze performance related data provided by the *DB2 Instrumentation Facility*.

The DB2 Instrumentation Facility gathers information about events and data in the system which can be collected and externalized. DB2 PM reporting is based on these previously collected trace records, and it allows problem analysis at a very detailed level. Collected trace data can also be loaded into the *performance database* of DB2 PM.

The data written by the DB2 instrumentation facility consists of different performance data record types, called instrumentation facility component identifiers (IFCIDs). The IFCIDs are grouped into trace classes and the classes are grouped into trace types. You find a list of trace types and classes and of IFCIDs supported for each DB2 trace class in appendix A of *DB2 for z/OS and OS/390 Tools or Performance Management*, SG24-6508.

DB2 trace types used as input to DB2 PM reporting facility are:

<b>Accounting data</b>	Provides information related to application programs
<b>Audit data</b>	Provides information about DB2 security controls
<b>Performance data</b>	Provides information about a variety of DB2 events
<b>Statistics data</b>	Provides information at the DB2 subsystem level
<b>Monitor data</b>	Used for DB2 PM Online Monitor program
<b>Global data</b>	Intended primarily for servicing DB2

The types used by DB2 PM batch are accounting, audit, performance, statistics, global. Monitor data is only used by DB2 PM online.

DB2 PM is host-based and has a workstation component. The Workstation Online Monitor provides you with some important online information you cannot find in the host-based online monitor, especially of interest to us is the information about statements in the global statement cache.

In the following sections we explain how the trace data can be used for analysis and reports on dynamic SQL. For more information about DB2 PM refer to *DB2 for z/OS and OS/390 Tools or Performance Management*, SG24-6508.

## 12.2 Using the DB2 Performance Monitor on a workstation

DB2 PM is host-based and has a workstation component, the Workstation Online Monitor, which provides you with some important online information about the dynamic statement cache you cannot find in the host-based online monitor in the current release.

The Workstation Online Monitor uses a server task at the host server site (the data collector), which collects the data DB2 generates about its own performance through the IFI. The workstation client presents this data to you in an understandable format.

The Workstation Online Monitor has further advantages over monitoring from TSO:

- ▶ It improves usability through a simple to use graphic interface.
- ▶ You can monitor more than one DB2 subsystem concurrently.

The following features are available from the Workstation Online Monitor:

<b>Thread activity</b>	Lets you examine the current activity of all active threads connected to a DB2 subsystem. From the thread summary window you can view key values for all connected threads. You can select any thread and drill down to detailed information.
<b>DB2 statistics</b>	Gives you an overview of the DB2 system activity. The statistics windows like <i>System Health</i> in Figure 12-5 on page 187 or <i>Statistics Details</i> in Figure 12-6 on page 188 show you system-wide performance information.
<b>System parameters</b>	Shows you the actual settings of your subsystem.
<b>Locking conflicts</b>	Shows all locking conflicts in a DB2 subsystem independent of a thread.
<b>Performance warehouse</b>	Lets you analyze performance data stored in the performance database. Currently this options only supports analysis of statistics data. It was introduced by APAR PQ57168 (PTF UQ62212).
<b>DB2 trace</b>	You can start and stop DB2 traces in order to produce DB2 PM batch reports.
<b>Exception processing</b>	Allows you to monitor DB2 events defined in the data collector. You can also define and monitor threshold values for threads and statistics.
<b>History</b>	Allows you to monitor the past thread activity and statistics.
<b>DB2 commands</b>	You can issue any DB2 command for which you are authorized, except “-START DB2”.

Detailed help information is available online.

In the *System Overview* window, shown in Figure 12-1, you can call those features via the roll-down menus of the menu bar, or via the icons of the left window. A click on the right mouse key opens a context menu for the referenced icon. There you find the functions that are allowed.

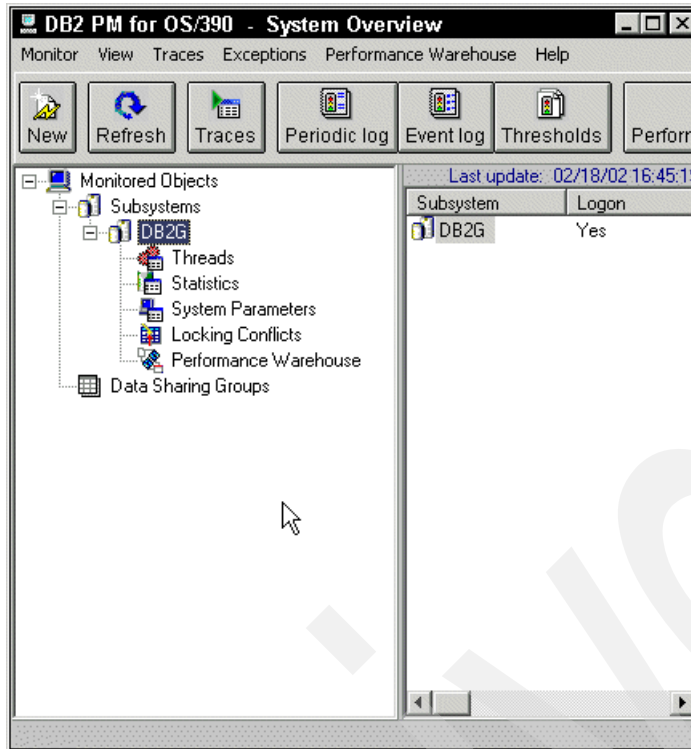


Figure 12-1 DB2 PM for OS/390 main window

If you are not acquainted with the Workstation Online Monitor refer to *DB2 for z/OS and OS/390 Tools or Performance Management*, SG24-6508, section 4.4 for information, how to start and connect.

In the following sections we discuss the functions of interest to dynamic SQL regarding:

- ▶ Thread activity
- ▶ DB2 statistics
- ▶ DB2 traces

## 12.3 Monitoring threads

In the *System overview* window (see Figure 12-1) you open the *Thread Summary* window by double-clicking on the *Threads* icon. The *Thread Summary* window shows a summary of all connected DB2 threads, as shown in Figure 12-2.

2002/02/19 08:39:35

0:00:06

2002/02/19 9:07:21

Primary Authorization	Member	Plan	Program Name	Elapsed Class 1	Elapsed Class 2	Total Class 3	CPU Class 1	CPU Class 2	Connection
BARTR2	N/P	DISTSERV	SQLLF000	0:58:01	0.73269	0.43923	0.32321	0.25039	SERVER
BARTR4	N/P	N/P	N/P	N/P	N/P	N/P	N/P	N/P	DB2CALL
STC	N/P	DGOMPOM	N/P	3:07:52	0.05674	0.00066	0.04511	0.00992	DB2CALL
STC	N/P	DGOMPOM	N/P	3:07:52	0.02568	0.00082	0.04576	0.01154	DB2CALL
STC	N/P	N/P	N/P	3:17:59	0.03047	0.00816	0.06393	0.01874	DB2CALL
STC	N/P	N/P	N/P	3:18:03	0.01119	0.00434	0.02012	0.00612	DB2CALL
STC	N/P	DGOMPOM	DGO@SDOB	3:18:03	5.79601	5.67023	0.09941	0.09481	DB2CALL
STC	N/P	DGOMPOM	DGO@DB2I	3:18:04	6.63823	0.27440	1.75878	0.73590	DB2CALL
PAOLOR2	N/P	ADB	ADBMAIN	0:12:46	7.01196	6.79728	0.17249	0.15373	TSO
PAOLOR1	N/P	DGOMPOM	DGO@YOCI	0:16:19	0.15276	0.13779	0.01797	0.01495	DB2CALL
PAOLOR1	N/P	DGOMPOM	N/P	0:16:26	N/P	N/P	0.00000	N/P	DB2CALL
PAOLOR3	N/P	ADB	ADBMAIN	0:13:57	2.59685	2.26819	0.36228	0.27690	TSO

Figure 12-2 Monitoring DB2 threads

In this window you can change the order of the columns shown and also modify the order in which threads are displayed by qualifying threads or sorting them by any of the columns. The following columns are available:

- ▶ Primary authorization
- ▶ Member
- ▶ Plan
- ▶ Program Name
- ▶ Elapsed time Class 1
- ▶ Elapsed time Class 2
- ▶ Total time Class 3
- ▶ CPU time Class 1
- ▶ CPU time Class 2
- ▶ Connection ID
- ▶ Request Count
- ▶ Getpage requests
- ▶ Connection type
- ▶ Status
- ▶ Correlation ID
- ▶ Requesting location
- ▶ Collection ID
- ▶ End User ID
- ▶ Workstation name
- ▶ Transaction name

You can activate an automatic refresh or refresh the information manually using the *Refresh* button (blue arrow).

With the thread summary you get not only an overview of all connected threads, but you can also recognize, if there are threads consuming an unusual amount of resources.

To get more details about a specific thread, double click on it. You open a *Thread Detail* window. As shown in Figure 12-3 the left part of the window shows the information offered as tree structure. The right part of the window shows the information details of the selected topic as blocks of DB2 PM performance counters (shown on the left pane). Some windows, such as the *SQL Activity* window allows you to drill down for more detailed information.

The Thread Detail window shows information about the following topics:

- ▶ Overview
- ▶ Identification
  - LUWs and others
  - Requester correlation
- ▶ DBRM/Package
  - Suspensions (Class 8)
- ▶ Times
  - Class 1,2,3
  - Suspensions
  - Other
- ▶ Locking
- ▶ Locked Resources
- ▶ RID List
- ▶ SQL Activity (DML)
  - DCL
  - DDL
  - Miscellaneous
- ▶ Buffer Manager
- ▶ SQL Statement
- ▶ Distributed Data
- ▶ Data Capture/Logging
- ▶ Query Parallelism
- ▶ Data Sharing Locking
- ▶ Group Buffer Pool
- ▶ Nested SQL Activity

This is basically all available DB2 accounting information.

The values shown for counters in the thread detail windows are snapshots taken at a single moment in time. A snapshot is taken when the window is opened and when you press the *Refresh* button. A refresh can also be done automatically at any interval you choose.

For threads using parallelism, values are aggregated across all parallel tasks within the thread. This means some counters can show unexpectedly large values. These do not necessarily indicate problems. For example, the CPU time can be larger than the real elapsed time.

For dynamic SQL we look at the DML statement counters as shown in Figure 12-3. This window shows the number of executions for DML statements issued by the thread:

- ▶ Total DML
  - Select
  - Insert
  - Update
  - Delete

- Prepare (QXPREP; this is the number of explicit prepare requests, from the SQL interface point of view)
- Describe
- Describe table
- Open
- Close
- Fetch

**Note:** Throughout this chapter we also give the IFCID field name between parenthesis where appropriate and related to dynamic SQL, like (QXPREP) to allow people that are using OEM products to find the corresponding field in their software.

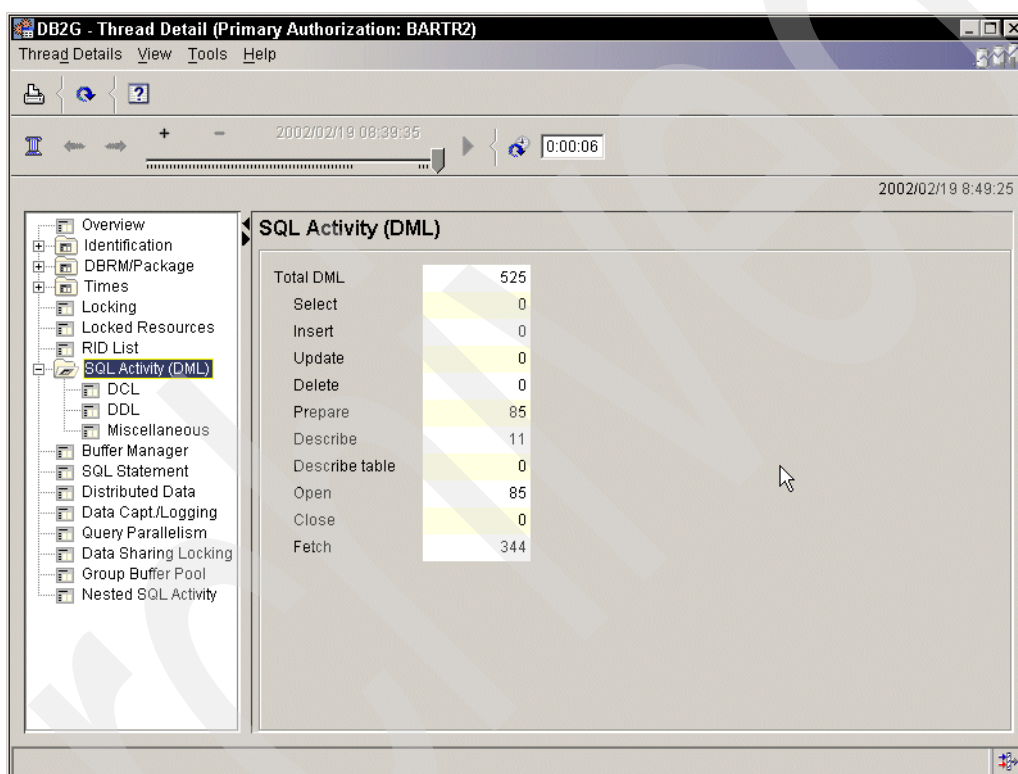


Figure 12-3 Thread Detail SQL activity

In Figure 12-3 the thread of TSO user BARTR2 has executed a total of 525 DML statements: 85 PREPAREs, 11 DESCRIBEs, 85 OPENs, and 344 FETCHes. Please note that the “all DML” field calculated a the sum of all the other items shown. It is not a true representation of all executed DML statements by the thread. It does, for example, not include EXECUTE IMMEDIATE statements.

**Important:** Prepares caused by EXECUTE IMMEDIATE statements are not represented here. They are not part of the number of Prepares, nor the total DML counter.

Note also that the number of SQL PREPARE statements executed at the server location might not match the user application because of DDF's internal processing.

For more detailed information about prepares and statement caching of this thread, you can use the *miscellaneous* counters of the *SQL Activity* window, as shown in Figure 12-4.



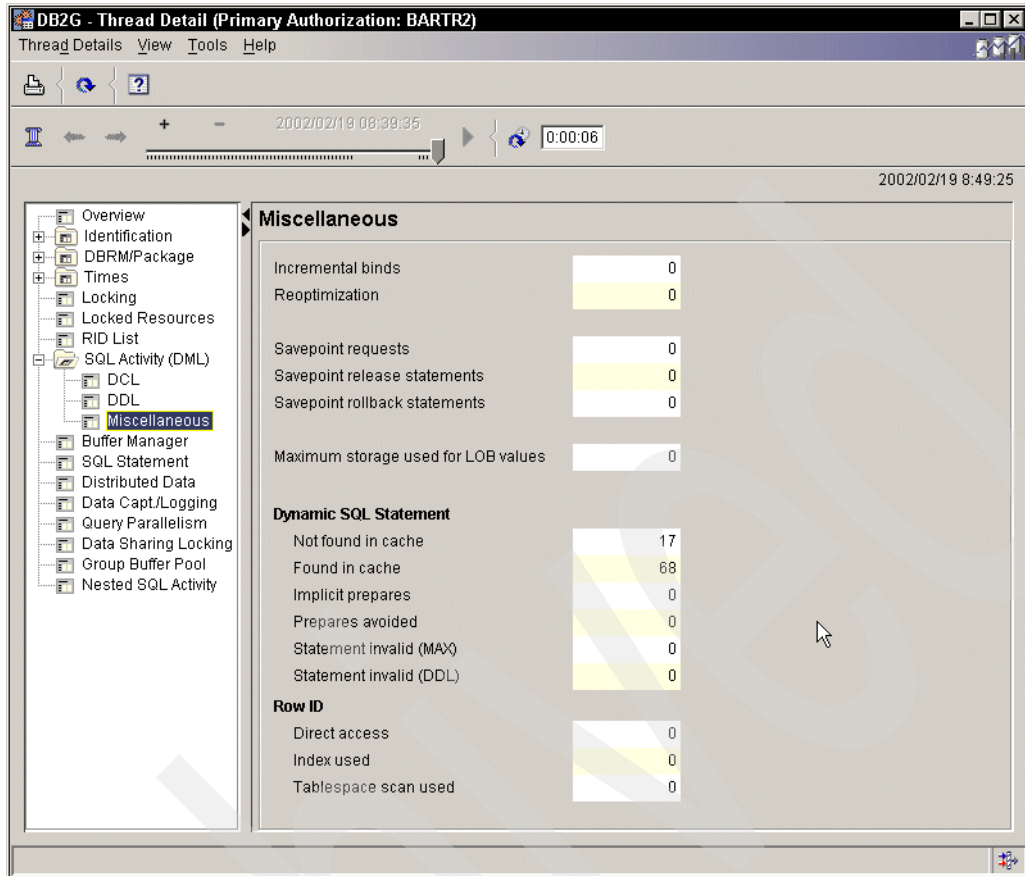


Figure 12-4 Thread detail dynamic SQL statement counters

The counters related to dynamic SQL and statement caching are:

- ▶ Not found in cache (QXSTNFND)
- ▶ Found in cache (QXSTFND)
- ▶ Implicit prepares (QXSTIPRP)
- ▶ Prepares avoided (QXSTNPRP)
- ▶ Statement invalid (MAX) (QXSTDEXP)
- ▶ Statement invalid (DDL) (QXSTDINV)

For the thread of user BARTR2, 17 statements were *not found in the cache*. This value is *not* necessarily the same as the number of *inserts into the cache*. Normally, when the statement cache is active and a statement is not found in the cache, the statement is prepared and inserted into the cache and both counters (QXSTNFND and QISEDSI) are incremented.

However, if the system “receives” a prepare request, it first checks if the statement is in the cache (before any work is done on the statement to get the most benefit from using the cache). If the statement is not found in the cache, the QXSTNFND counter is incremented. If the statement later fails during prepare processing, for example, because a user does not have the authorization to access the table, no statement is inserted into the global cache and the QISEDSI counter is not incremented.



**Attention:** There are two counters used in DB2:

- ▶ QXSTNFND shows the number of statements not found in the global cache
- ▶ QISEDSI shows the number of statements inserted into the global cache.

In this report, QXSTNFND is used.

In this case, 68 short prepares have been executed, because the statements have been found in the cache and could be copied into the local storage. Please note that the sum of full and short prepares may not be equal to the number of PREPARE statements (QXPREP) in Figure 12-3 on page 185.

Prepare operations invoked by EXECUTE IMMEDIATE statements are counted as full or short prepares (depending in whether or not they are found in the global cache), but not as PREPARE statements.

For more information about the counters of statement caching refer to 12.4.4, “Monitoring dynamic statement cache” on page 195.

## 12.4 Monitor statistics

In the *System overview* window (see Figure 12-1 on page 182) you enter the statistics feature of the Workstation Online Monitor by double-clicking on the *Statistics* icon. The *System Health* window shows a summary of important global subsystem information at a glance in graphic format as shown in Figure 12-5.

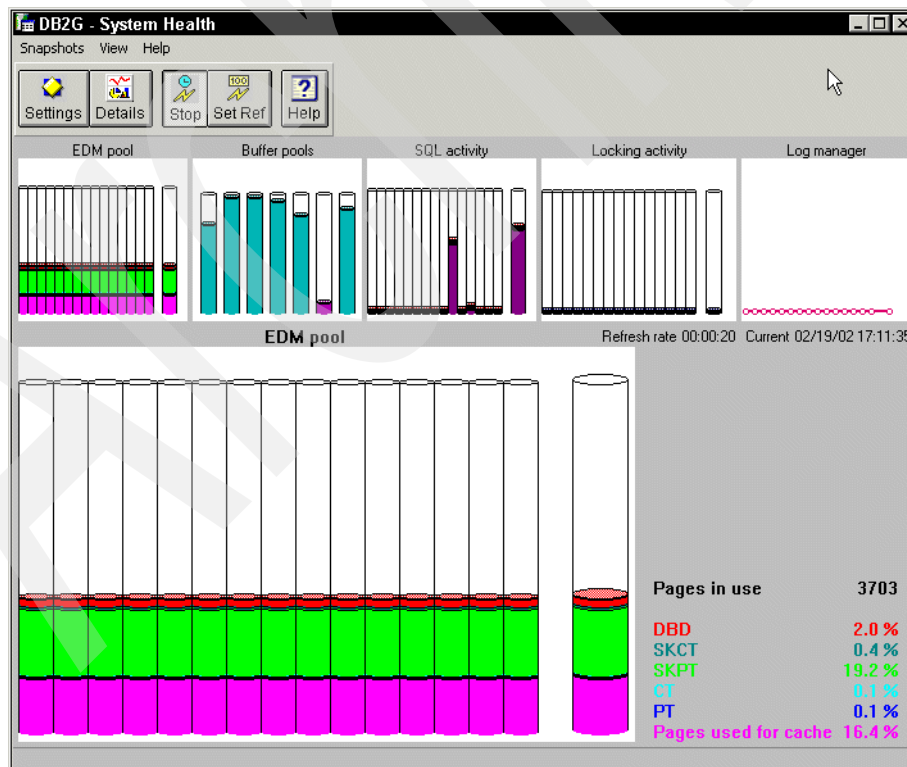


Figure 12-5 System health window of monitor statistics

Figure 12-5 provides information about the following areas:

- ▶ EDM pool
- ▶ Buffer pool
- ▶ SQL activity
- ▶ Locking activity
- ▶ Log manager

The upper portion of the window shows graphs for each of the areas. With a single mouse click on one of these graphs you can expand the selected graph in the lower portion of the window.

In the lower portion of Figure 12-5, the values for the EDM pool are shown in the form of stacked bar charts, one column per snapshot.

You can alter the frequency of the snapshots and the number of snapshots displayed via the *Settings* windows.

Clicking on one bar and holding the left mouse key, you can look at the numeric values of that snapshot. The chart shows the following values for the EDM pool:

- ▶ EDM pool pages in use (%)
- ▶ EDM pool failure - shown as a delta value
- ▶ DBD
- ▶ SKCT
- ▶ SKPT
- ▶ CT
- ▶ PT
- ▶ Pages used for cache

For detailed information about your subsystem click on the *Details* button to open the *Statistics Detail* window, as shown in Figure 12-6.

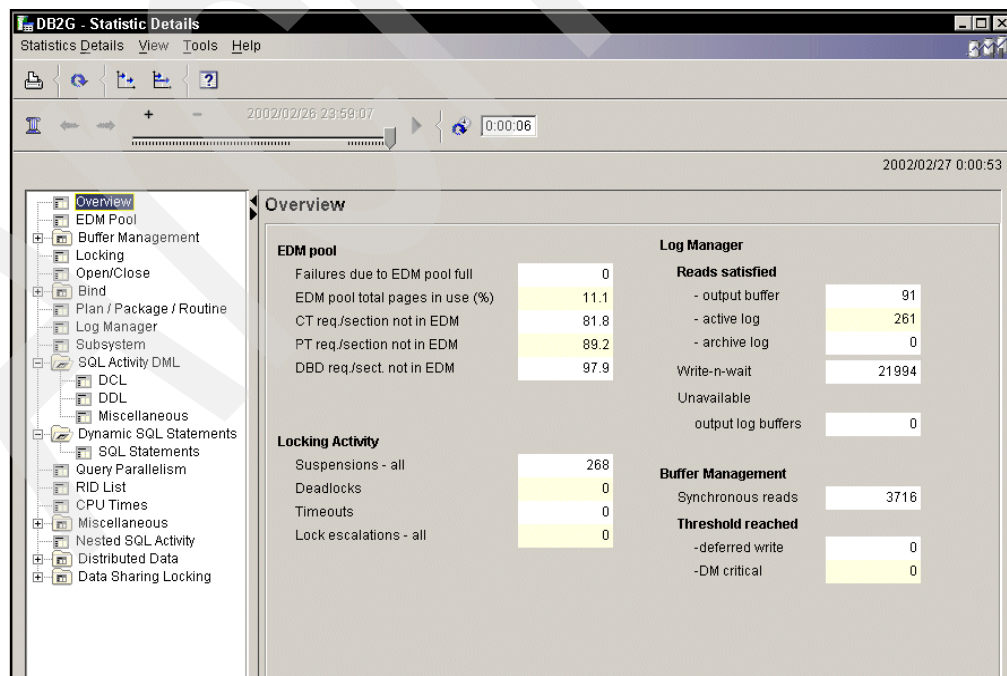


Figure 12-6 Statistics detail overview

As shown in Figure 12-6, the left part of the window shows the information offered as a tree structure. The right part of the window shows the information details of the selected topic as blocks of DB2 PM counters. Some windows, such as the *SQL Activity DML* window allow you to drill down for more detailed information.

The main topics are the following:

- ▶ Overview
- ▶ EDM Pool
- ▶ Buffer Management
  - Group Buffer Pool
    - Global GBP
- ▶ Locking
- ▶ Open/Close
- ▶ Bind
  - Rebind/Autobind
- ▶ Plan/Package/Routine
- ▶ Log Manager
- ▶ Subsystem
- ▶ SQL Activity DML
  - DCL
  - DDL
  - Miscellaneous
- ▶ Dynamic SQL Statements
  - SQL Statements
- ▶ Query Parallelism
- ▶ RID List
- ▶ CPU Times
- ▶ Miscellaneous
  - DB2 Commands
    - Display/Other
  - Instrumentation
  - IFC Destinations
  - Data Capture
  - IFI Interface
- ▶ Nested SQL Activity
- ▶ Distributed Data
  - Location
- ▶ Data Sharing Locking
  - Suspension/Notify

Most DB2 statistics field values accumulate while the DB2 subsystem is active. Some fields, however, are snapshot values or high-water mark values. A snapshot value is a current value, and is updated each time the statistics values are displayed. A high-water mark is a highest value reached since startup, and is updated each time the statistics values are displayed. DB2 PM also has the capability to show the values between two snapshots (DELTA mode) or the accumulated values from a starting point (timestamp) defined by user (INTERVAL mode).

### 12.4.1 Monitoring the EDM pool

When using (global) dynamic statement caching the EDM pool is key to good overall performance. The Workstation Online Monitor provides detailed information about the EDM pool. The *Overview* window shows selected key values of the EDM pool (Figure 12-6 on page 188).

You can use the window in Figure 12-6 to examine the efficiency of the EDM buffer pool, to determine the size of the EDM pool for optimum system performance.

The EDM pool should be large enough to store the cursor tables (CTs), package tables (PTs), and database descriptors (DBDs) for the most frequently used applications.

A critical performance factor for online applications is I/O activity. Therefore, the necessary skeleton cursor tables (SKCTs), skeleton package tables (SKPTs), and DBDs should already be in the EDM pool whenever they are needed for a new thread. This means that the EDM pool should be large enough to prevent these from being stolen.

Important values to monitor are the ratios shown in the number of successful requests from the EDM pool to the total number of requests in:

- ▶ CT req./section not in EDM
- ▶ PT req./section not in EDM
- ▶ DBD req./sect. not in EDM

The ratios can approach a high value with high volume, re-referencing transactions. For some other environments, a ratio of 5 to 10 is acceptable. Make sure these ratios do not deteriorate when you start using the global statement cache. If they do, consider enlarging the EDM pool or use put the dynamic statement cache in a data space.

More detailed information you find in the *EDM pool* window as shown in Figure 12-7. This window also provide efficiency numbers for the EDM pool in the form of hit ratios:

- ▶ DBD hit ratio (%)
- ▶ CT hit ratio (%)
- ▶ PT hit ratio (%)

Generally hit ratios are more easy to understand than the ratios from Figure 12-6.

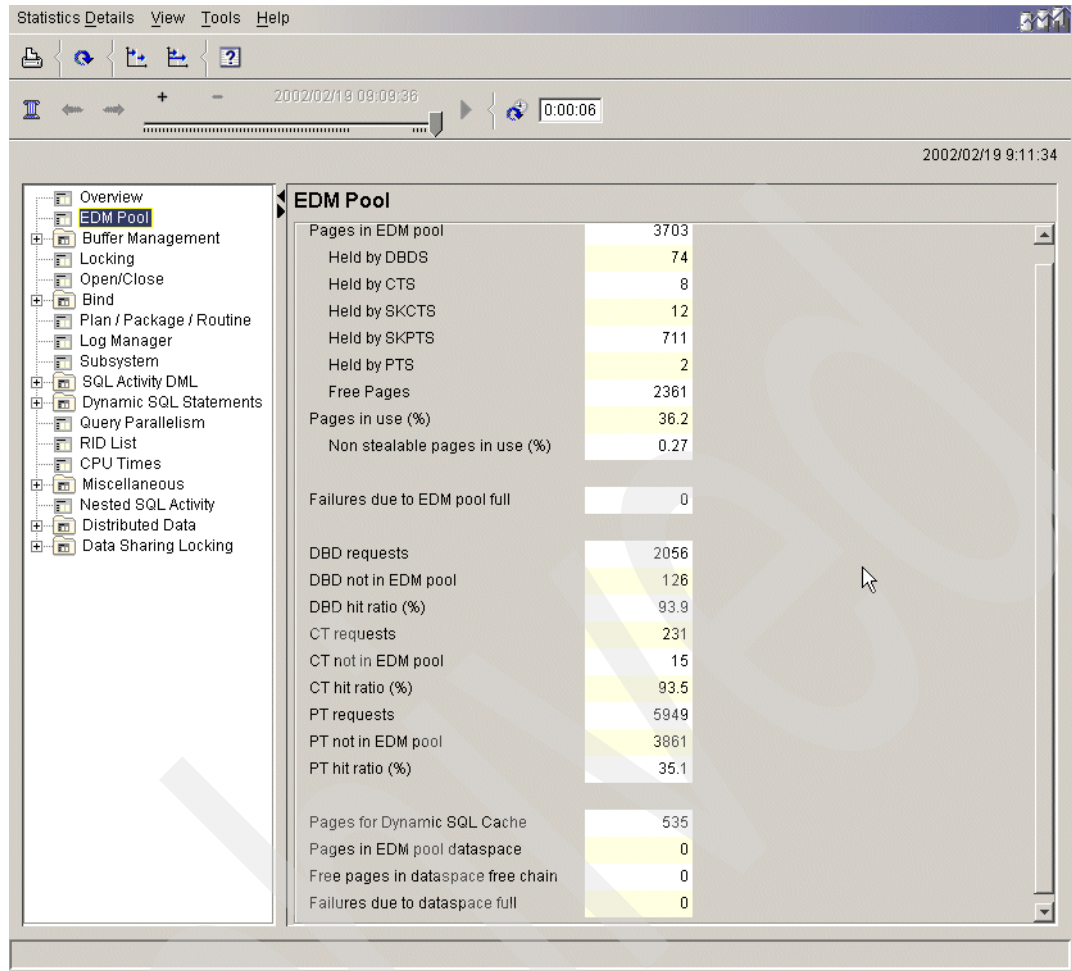


Figure 12-7 EDM pool statistics

The EDM pool statistics provide the following values:

- ▶ Pages in EDM pool
- ▶ Pages used (DBD)
- ▶ Pages used (CT)
- ▶ Pages used for SKCT
- ▶ Pages used for SKPT
- ▶ Pages used (PT)
- ▶ Free pages
- ▶ Pages in use (%)
- ▶ Non stealable pages in use (%)
- ▶ Failures due to EDM pool full
- ▶ Request for sections (DBD)
- ▶ Section not in EDM pool (DBD)
- ▶ DBD hit ratio (%)
- ▶ Request for sections (CT)
- ▶ Section not in EDM pool (CT)
- ▶ CT hit ratio (%)
- ▶ Request for sections (PT)
- ▶ Section not in EDM pool (PT)
- ▶ PT hit ratio (%)
- ▶ Pages used for dynamic SQL cache (QISEDSC)

- ▶ Pages in EDM pool data space (QISEDPGE)
- ▶ Free pages in data space free chain (QISEDFRE)
- ▶ Failures due to data space full (QISEDFA)

### Recommendations

During peak periods, the usage of EDM pool pages should be between 80 and 100 percent. Because stealable pages are included in this percentage, a value close to 100 percent does not necessarily indicate an EDM pool constraint.

When the value approaches 100, examine the hit ratios for DBDs, CTs, and PTs to verify that acceptable levels are met. If the EDM pool is too small, it causes:

- ▶ Increased I/O activity for table spaces SCT02, SPT01, and DBD01 in database DSNDB0.1
- ▶ Increased response times, due to loading the SKCTs, SKPTs, and DBDs. If dynamic statement caching is used, and the needed SQL statement is not in the EDM pool, that statement has to be re-prepared.
- ▶ Fewer threads used concurrently, due to a lack of storage.

If the percentage of used pages is consistently less than 80%, the EDM pool size is probably too large.

The total number of failures caused by an full EDM pool should be 0. Any other value indicates that the EDM pool is too small for the amount of activity being performed.

## 12.4.2 Monitoring SQL activity

Using dynamic statement caching we are interested in detailed information about the global and local cache hit ratios and in the amount of prepare activity.

In Figure 12-8 we find the statistics for the DML activity.

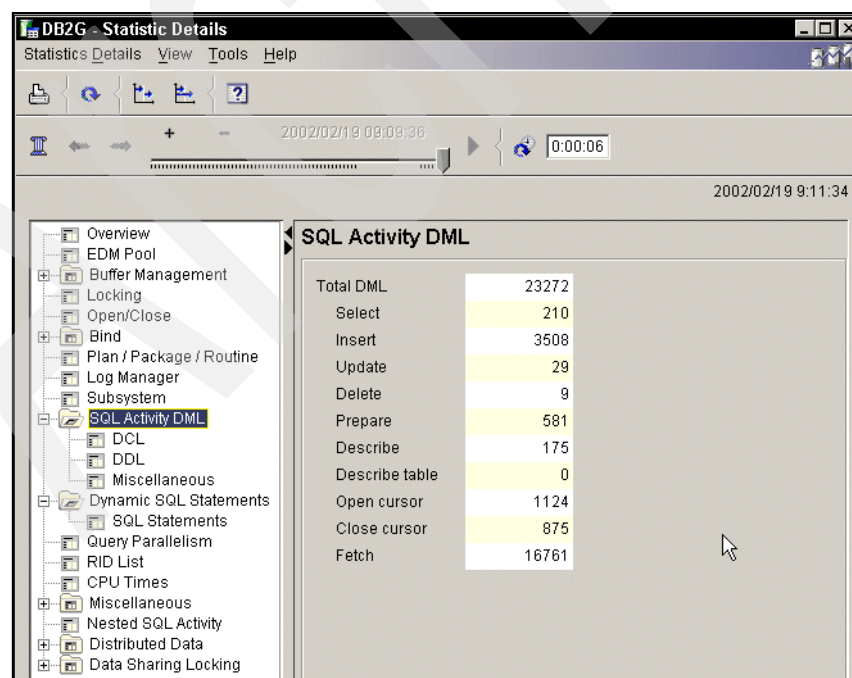


Figure 12-8 SQL DML activity statistics



Figure 12-8 shows only the number of executions for Data Manipulation Language (DML) statements issued by the threads. It shows the total and the counters for following statements:

- ▶ Select
- ▶ Insert
- ▶ Update
- ▶ Delete
- ▶ Prepare
- ▶ Describe
- ▶ Describe table
- ▶ Open cursor
- ▶ Close cursor
- ▶ Fetch

The information in the statistics DML section is similar to that of the accounting DML section, discussed in 12.3, “Monitoring threads” on page 182, with the same restrictions. Needless to say that the statistics information is at the subsystem level, whereas the accounting information is at the tread level.

### 12.4.3 Monitoring dynamic SQL statements

In Figure 12-9 we find statistics information about prepares and the cache usage.

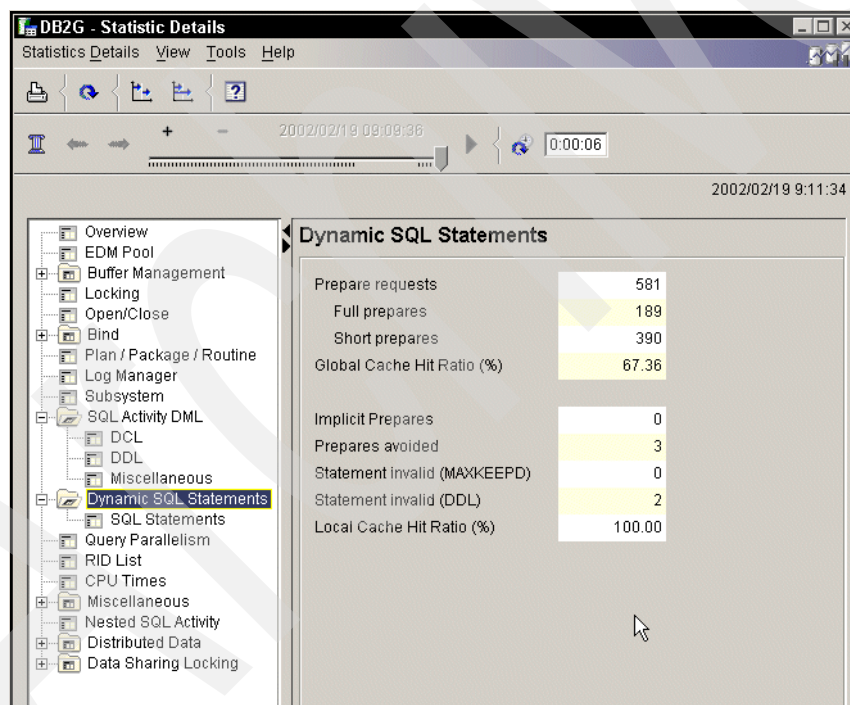


Figure 12-9 Dynamic SQL cache statistics

Figure 12-9 shows:

#### Prepare requests

Number of *Prepare requests*. It is not the sum of full or short prepares shown below. It is the sum of the number of explicit (QXPREP) and implicit (QXSTIPRP) prepare requests.

#### Full prepares

Number of full prepares caused by prepare requests and EXECUTE IMMEDIATE statements (QISEDSI).

<b>Short prepares</b>	Number of short prepares due to global caching. (This number is calculated by the Workstation Online Monitor as QISED SG - QISED SI).
<b>Global Cache Hit Ratio (%)</b>	The ratio of field “ <i>short prepares</i> ” to the sum of “ <i>full prepares</i> ” and “ <i>short prepares</i> ” ((QISED SG - QISED SI)/QISED SG).

**Attention:**

QXSTNFND shows the number of statements not found in the global cache.  
QISED SI shows the number of statements inserted into the global cache.  
QISED SG shows the number of request for the global cache (caused by either PREPAREs and EXECUTE IMMEDIATEs inserting into it (full prepares) or copies from it (short prepares)).

As mentioned before, the counters QXSTNFND and QISED SI may differ.

The number of short prepares here is calculated as QISED SG - QISED SI. Therefore failed prepares may be counted as short prepares!

<b>Implicit prepares</b>	Number of implicit prepares due to local caching (QXSTIPTP).
<b>Avoided prepares</b>	Number of avoided prepares due to full caching (QXSTNPRP).
<b>Statement invalid (MAXKEEPD)</b>	The number of times a statement was discarded from local cache, because MAXKEEPD has been reached (QXSTD EXP)
<b>Statement invalid (DDL)</b>	The number of times a statement was purged from the cache because of DDL or RUNSTATS on depending objects (QXSTD INV)
<b>Local Cache Hit Ratio (%)</b>	The ratio of field “ <i>Avoided prepares</i> ” to the sum of “ <i>Avoided prepares</i> ” and “ <i>Implicit prepares</i> ”. With full caching this ratio is 100% as long as statements have not been invalidated because MAXKEEPD has been reached. With local or global caching only it is zero. (QXSTNPRP/(QXSTNPRP + QXSTIPTP))

Look at 8.2, “The different levels of statement caching” on page 142 for detailed information about the different kind of prepares.

Failed prepares (for example because of a lack of authority) may increment the number of PREPARE statements (QXPREP) and the number of statements not found in the global cache (QXSTNFND), but not the number of full prepares (QISED SI).

Executing the SQL EXPLAIN statement increments the number of PREPARE statements, but the statement is not inserted into the cache.

**Note:** The QX\* counters are available in DB2 accounting and statistics traces class 1. Therefore they are available both at the thread (accounting) level and at the subsystem (statistics) level. The QISE\* counters are available only when the DB2 statistics class 1 is active.



## 12.4.4 Monitoring dynamic statement cache

The Workstation Online Monitor allows a detailed look into the dynamic statement cache as shown in Figure 12-10 and Figure 12-11.

When you first open this window, this window may be blank because a filter is activated to block to limit the number of rows to be displayed. You can disable filtering, or modify the filter to an appropriate value to get information about statements in the cache. Then click on the check box to enable the data to be transferred.

**Note:** The information is only downloaded from the host (data collector) when you request it. Ensure that the check box is checked. To update the information, use the *Refresh* button on the upper tool bar.

The detailed information about the statement cache is only available to online monitor programs. The information can not be externalized to SMF or GTF, and processed by a DB2 PM batch job. In order to obtain the information, the IFI program has to request IFCID 316. IFCID 317 can be used to obtain the full SQL statement text (as IFCID 316 contains only the first 60 bytes of the statement text). These IFCIDs are available when monitor trace class 1 is active. IFCID 316 contains a large number of fields.

By default the system only gathers basic status information about the cached statement, like the number of times it was reused, when it was inserted, .... When you activate IFCID 318, this is an indicator to fill in all available fields in IFCID 316 including the statistical fields. IFCID 318 itself does not contain any data. It acts only as a switch for IFCID316 to collect additional information. (In order for IFCID 318 to trigger the collection of the suspension timers (wait times) accounting class 3 has to be active.

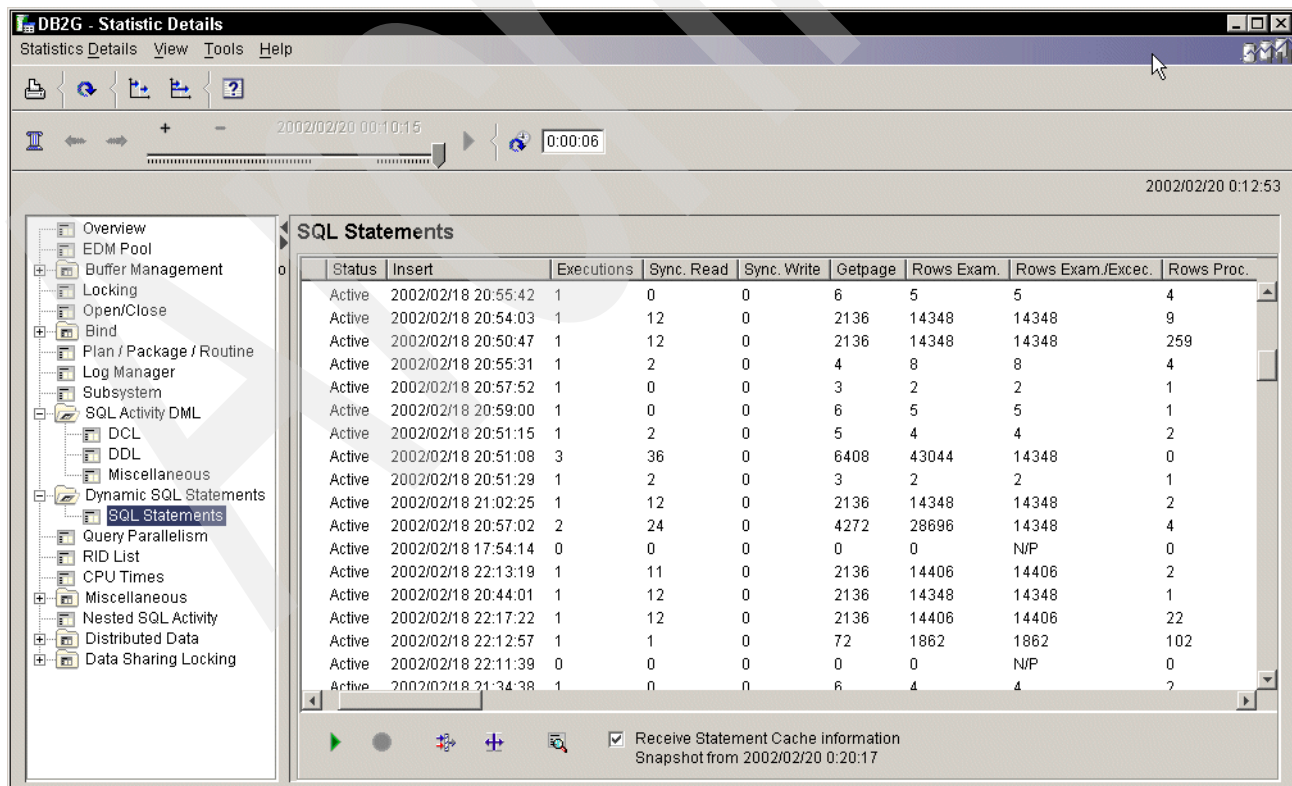


Figure 12-10 Monitoring the statement cache: execution statistics

The information shown in the window is a list of statements in the cache with some basic information and the beginning of the statement string. As mentioned before, to get more detailed information about the statements, you must activate IFCID 318, which allows all IFCID 316 information to be gathered. A control icon (green triangle) below the window allows you to activate IFCID 318.

**Attention:** This trace can add about two percent overhead to your DB2 system.

When you start the statement cache trace, it continues until you explicitly stop it using the red button, the trace will continue even when you close down your DB2 PM workstation client.

The control icons below the data area of this window (listed from left to right) request the following functions:

- ▶ Start collecting additional IFCID 316 data from the Dynamic SQL statement cache. This starts the following DB2 trace:

**-START TRACE(MON) IFCID(318)**

- ▶ Stop collecting additional data from the cache.
- ▶ Filter the statements shown on this screen.
- ▶ Customize the columns shown in this window.
- ▶ View details of a selected statement
- ▶ Use the check box to download collected trace data from the DB2 PM host.

Figure 12-10 lists the contents of the dynamic SQL statement cache using the statement cache trace. The window shows the following columns:

- |                            |                                    |
|----------------------------|------------------------------------|
| ▶ Accumulated CPU time     | ▶ Read by other threads wait time  |
| ▶ Copies                   | ▶ Ref. table name                  |
| ▶ Executions               | ▶ Rows exam/exec                   |
| ▶ Current SQL Id           | ▶ Rows exam                        |
| ▶ Currentdata bind         | ▶ Rows proc/exec                   |
| ▶ Current degree           | ▶ Rows proc                        |
| ▶ Current precision        | ▶ SQL Statement                    |
| ▶ Current rules            | ▶ Sort                             |
| ▶ Cursor hold              | ▶ Statement line number            |
| ▶ DB2 limit exceeded       | ▶ Status                           |
| ▶ Dynamic rules bind       | ▶ Storage limit exceeded           |
| ▶ Elapsed time             | ▶ Synch I/O wait time              |
| ▶ Elapsed time/exec        | ▶ Synch exec unit switch wait time |
| ▶ Getpage                  | ▶ Synch read                       |
| ▶ Global locks wait time   | ▶ Synch write                      |
| ▶ Groups                   | ▶ Table name                       |
| ▶ Index scan               | ▶ Table qualifier                  |
| ▶ Insert time              | ▶ Table space scan                 |
| ▶ Isolation bind           | ▶ Transaction                      |
| ▶ Lock and latch wait time | ▶ User Id                          |
| ▶ Program name             | ▶ Users                            |
| ▶ Qualifier bind           | ▶ Write by other threads wait time |

When a statement is in the statement cache you have very detailed performance information at the SQL statement level (but not at the execution level since the information is grouped across all executions and across all programs that use the statement). To obtain this information for non-cached statement requires that you start a set of performance traces that are a lot more expensive.

Most of the numbers above are updated at the end of execution. The exception being cursors. The open and each fetch increment the counters as they end. In this case, you do not have to wait until the cursor closes before the numbers get incremented. But in general, the numbers here are not as “accurate” as numbers in for example accounting, where the counter is updated as the event takes place. Therefore, if a query scans a lot of rows before a qualifying row is returned between FETCH operations, the counters do not get updated for a long time and using the statement cache information to monitor long running queries should be used with this information in mind.

Figure 12-11 lists the contents of the dynamic SQL statement cache showing the first part of the statement strings. These fields can also be found when you zoom in on a statement, as shown in Figure 12-12 on page 198.

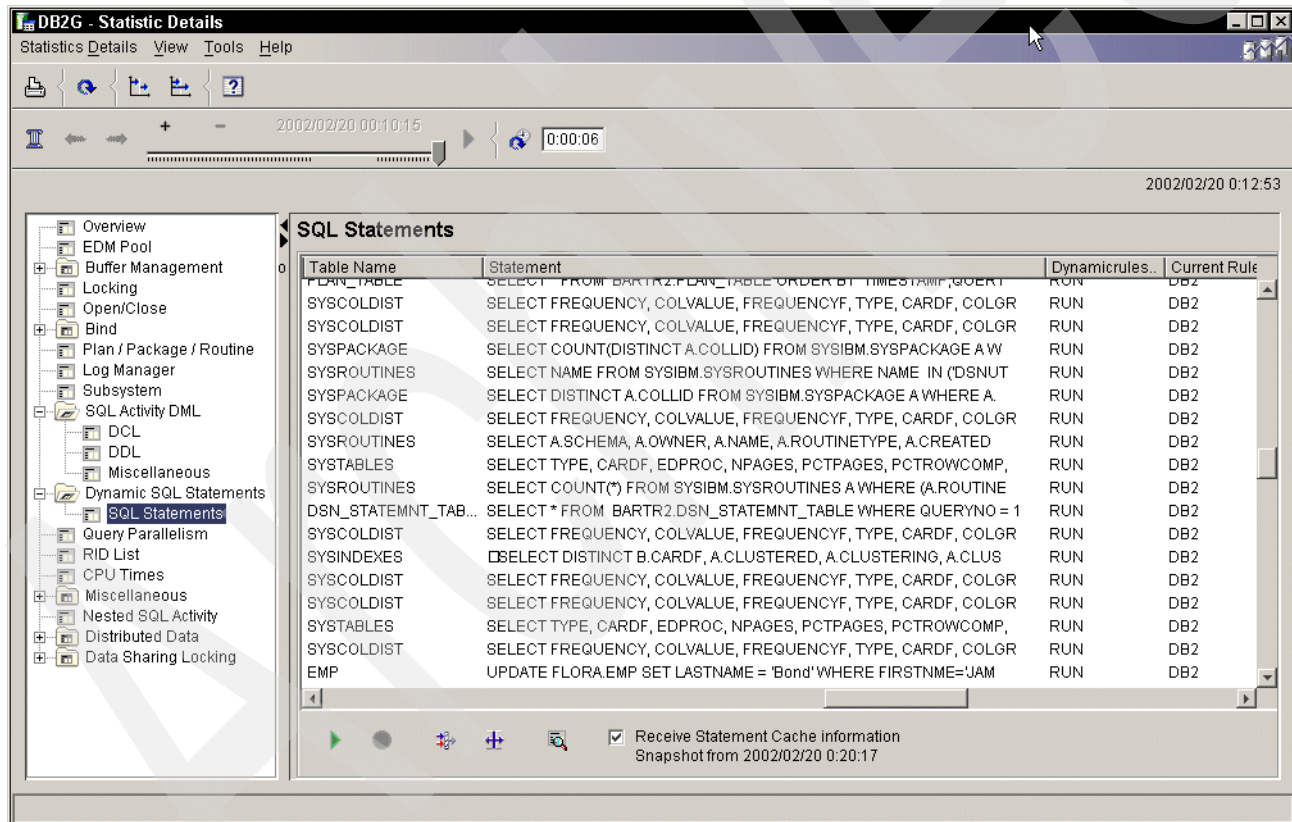
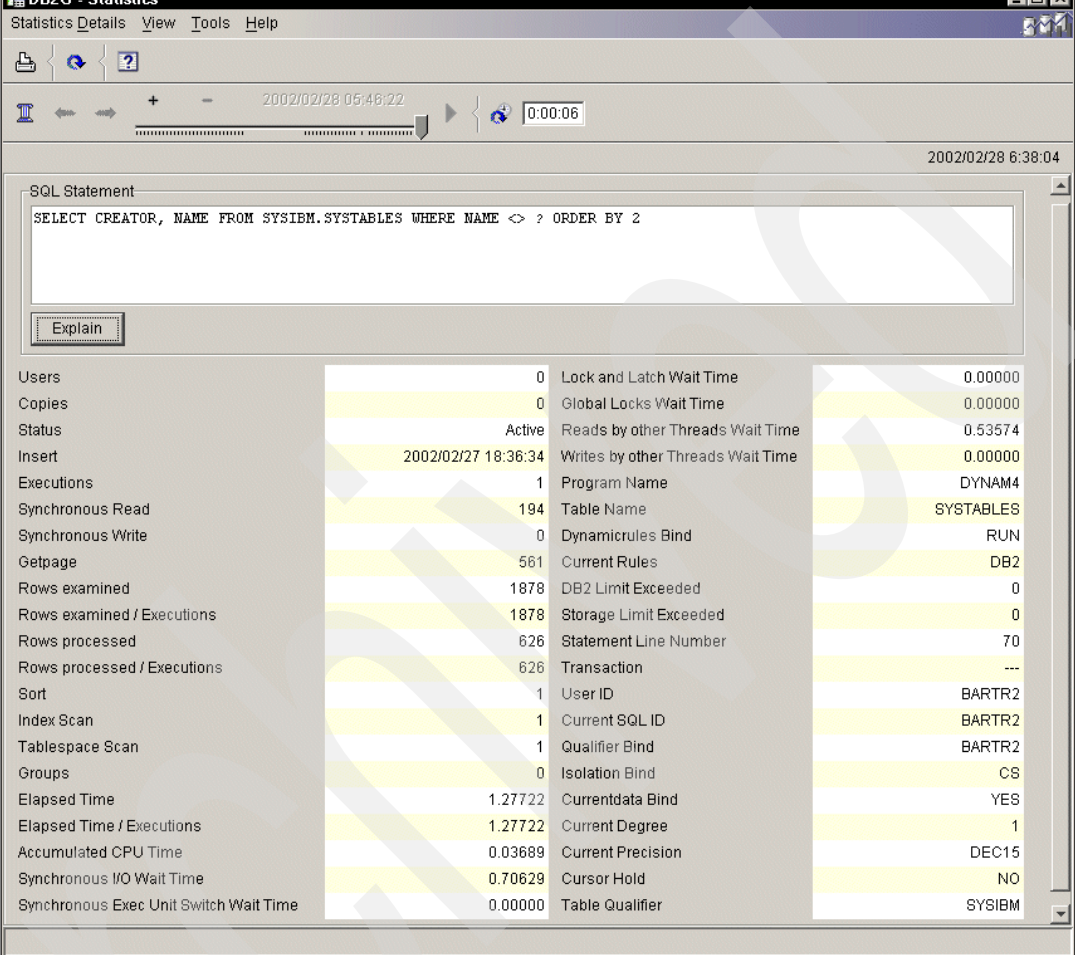


Figure 12-11 Monitoring the statement cache: statement strings

## 12.4.5 Explaining a statement from the cache

You can select a statement which you want to examine and double click it to view the same information as in the *SQL statements* window in a the *Statement detail* window together with the full statement text as shown in Figure 12-12.



The screenshot shows the 'DB2 Statistics - Statement Details' window. At the top, there's a menu bar with 'Statistics', 'Details', 'View', 'Tools', and 'Help'. Below it is a toolbar with icons for file operations and a search icon. A status bar at the top right shows the date and time '2002/02/28 6:38:04'. The main area is divided into two sections. The top section, titled 'SQL Statement', contains a text box with the SQL query: `SELECT CREATOR, NAME FROM SYSIBM.SYSTABLES WHERE NAME <> ? ORDER BY 2`. Below the text box is an 'Explain' button. The bottom section displays a table of execution statistics. The table has four columns: a category, a value, a description, and another value. The categories include Users, Copies, Status, Insert, Executions, Synchronous Read, Synchronous Write, Getpage, Rows examined, Rows examined / Executions, Rows processed, Rows processed / Executions, Sort, Index Scan, Tablespace Scan, Groups, Elapsed Time, Elapsed Time / Executions, Accumulated CPU Time, Synchronous I/O Wait Time, and Synchronous Exec Unit Switch Wait Time. The values are mostly integers or floating-point numbers. The descriptions include 'Lock and Latch Wait Time', 'Global Locks Wait Time', 'Reads by other Threads Wait Time', 'Writes by other Threads Wait Time', 'Program Name', 'Table Name', 'Dynamicrules Bind', 'Current Rules', 'DB2 Limit Exceeded', 'Storage Limit Exceeded', 'Statement Line Number', 'Transaction', 'User ID', 'Current SQL ID', 'Qualifier Bind', 'Isolation Bind', 'Currentdata Bind', 'Current Degree', 'Current Precision', 'Cursor Hold', and 'Table Qualifier'. The values for these descriptions are mostly zeros or small numbers.

Category	Value	Description	Value
Users	0	Lock and Latch Wait Time	0.00000
Copies	0	Global Locks Wait Time	0.00000
Status	Active	Reads by other Threads Wait Time	0.53574
Insert	2002/02/27 18:36:34	Writes by other Threads Wait Time	0.00000
Executions	1	Program Name	DYNAM4
Synchronous Read	194	Table Name	SYSTABLES
Synchronous Write	0	Dynamicrules Bind	RUN
Getpage	561	Current Rules	DB2
Rows examined	1878	DB2 Limit Exceeded	0
Rows examined / Executions	1878	Storage Limit Exceeded	0
Rows processed	626	Statement Line Number	70
Rows processed / Executions	626	Transaction	---
Sort	1	User ID	BARTR2
Index Scan	1	Current SQL ID	BARTR2
Tablespace Scan	1	Qualifier Bind	BARTR2
Groups	0	Isolation Bind	CS
Elapsed Time	1.27722	Currentdata Bind	YES
Elapsed Time / Executions	1.27722	Current Degree	1
Accumulated CPU Time	0.03689	Current Precision	DEC15
Synchronous I/O Wait Time	0.70629	Cursor Hold	NO
Synchronous Exec Unit Switch Wait Time	0.00000	Table Qualifier	SYSIBM

Figure 12-12 Details of cached statement

In this window you find an *Explain* button. If Visual Explain is installed on your workstation, you can use this button to explain the shown statement.

For a simple SELECT statement like the one in Figure 12-12, a diagram like Figure 12-13 is generated by Visual Explain. Refer to the tutorial and the online help for more information about the capabilities of Visual Explain, or refer to *DB2 UDB for OS/390 Version 6 management Tools Package*, SG24-5759.

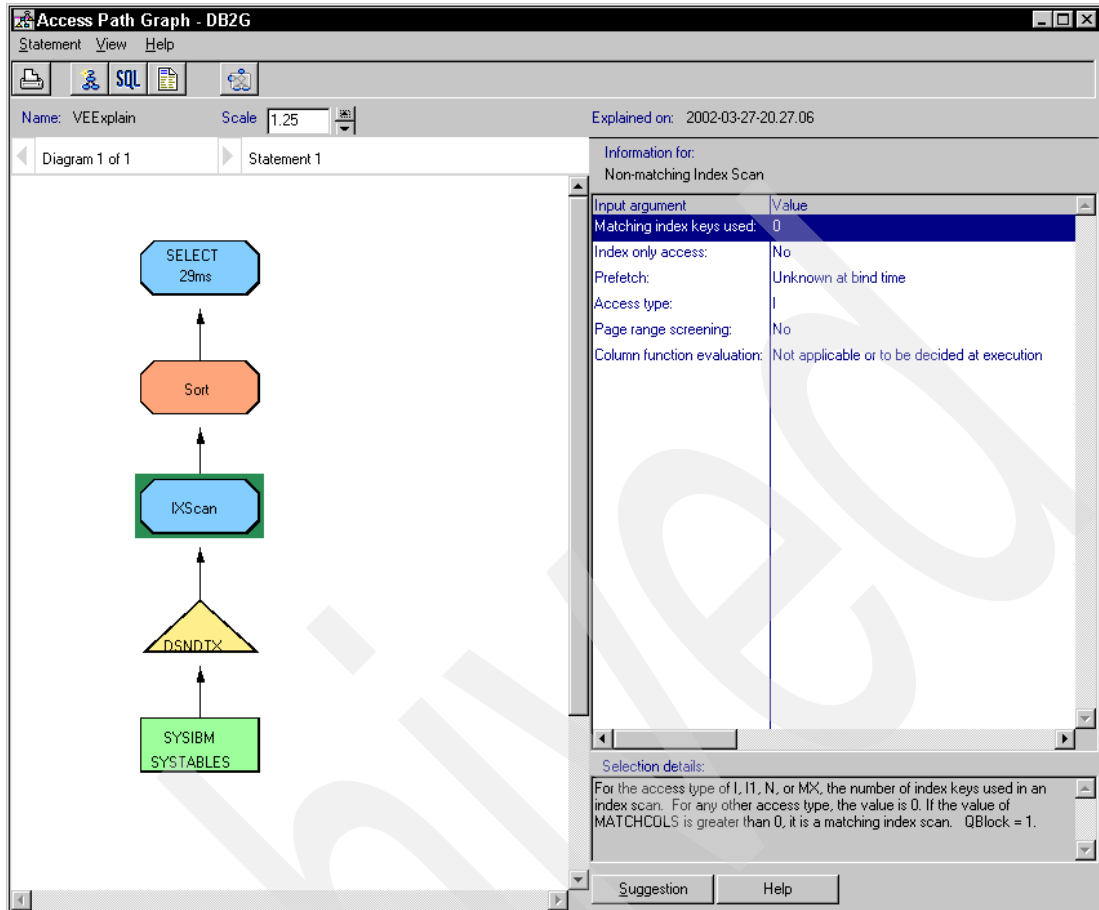


Figure 12-13 Using Visual Explain for a cached statement

Be aware that Visual Explain re-prepares the statement using the actual catalog information to present the information above. The explained access path is not necessarily be the same as the cache statement really uses. To obtain the real access path that is used by a statement that resides in the cache, you must have a trace active that collects IFCID 22 (the mini-plan) at the time the statement is inserted into the cache. More information on how to exploit this information can be found in 12.5.5, “Analyzing performance data” on page 210.

Note that the usage of parameter markers, which is recommended for statement caching, may cause the DB2 optimizer to choose a sub-optimal access path, because he cannot evaluate predicates exactly, especially in cases where the data is not distributed uniformly or when correlation between columns exists.

## 12.5 Using the performance database of DB2 PM

The performance database of DB2 PM allows you to perform additional evaluations on DB2 trace data. For example you can analyze the access paths of dynamic SQL statements using IFCID 22. You can store and accumulate the trace data over several days in DB2 tables and analyze it using SQL.

In the following sections we explain, how the DB2 PM performance database can be used for analyzing dynamic SQL and give some considerations on how to use the information.



## 12.5.1 Overview

When you invoke the DB2 trace facility, you can specify the kind of trace information you want to collect. In the **-START TRACE** command, first specify the type of data, then the classes within that type. You can also specify particular IFCIDs.

It is recommended that you turn on a *performance trace* only if there is a specific need for it because some of the performance trace classes cause a significant performance overhead. If you need to start a performance trace, please consider starting only the requested IFCIDs instead of starting a class. To do this, you may use the performance class 30, 31 or 32 which require that you specify the IFCIDs explicitly. If you specify the IFCID keyword together with a CLASS that already has IFCIDs assigned to it by DB2 (all the other classes 1-29), it will trace the IFCIDs you specify on the IFCID keyword in addition to the IFCIDs assigned to the CLASS. Turn off all traces you do not need!

In addition to the System Management Facility (SMF) and Generalized Trace Facility (GTF) data sets, DB2 PM collects DB2 trace records in a sequential data set. DB2 PM tracing provides functions which allow you to collect trace data asynchronously depending on manual or automatic “collect start/stop” criteria for a specific time frame or specific application. It is possible to specify trace data to be collected by DB2 PM reports and categories, and select/deselect single IFCIDs and record this data in a sequential data set.

## 12.5.2 Configuring traces in the Workstation Online Monitor

You can configure and start traces not only in the host based part of DB2 PM, but also in the Workstation Online Monitor. To configure a new trace, you can click on the *Traces* button in the *System Overview* window (Figure 12-1 on page 182) or on the *Traces* menu in the menu bar. In any other window you find the trace function in the *Tools* menu.

Figure 12-14 shows the *Configure* window to create a new trace definition.

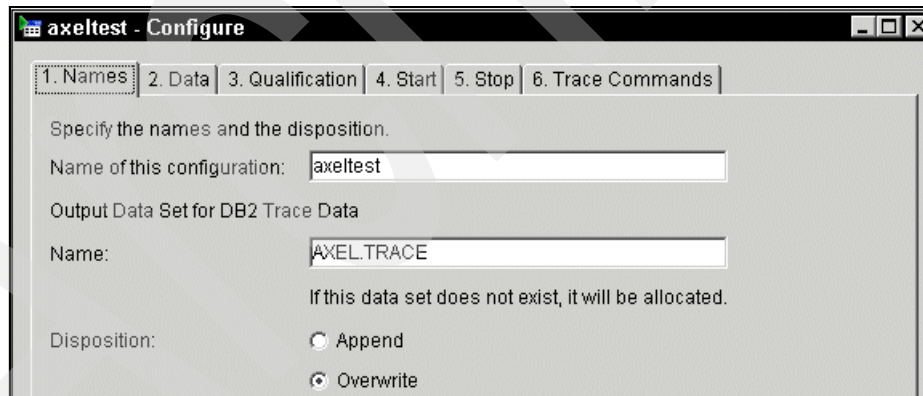


Figure 12-14 Configure a DB2 trace

You can also edit a selected trace configuration in this window.

The configuration information is spread over different pages/tabs within this window. The pages have the following contents:

<b>Names</b>	Name for this trace configuration and output data set details
<b>Data</b>	Selection of the trace data to be collected
<b>Qualification</b>	Criteria to the data collection process
<b>Start</b>	Start conditions for the collection process
<b>Stop</b>	Stop conditions for the collection process
<b>Trace commands</b>	DB2 START TRACE commands produced by your definitions

The **OK** button saves your trace configuration and branches to the *Trace Configurations* window. If this is a new configuration, it is shown in the Trace Configurations window.

You can switch back and forth between pages by clicking the tabs. Click **OK** to finish this trace configuration. If a page contains incorrect or missing data when you try to finish your configuration by clicking **OK**, a message box appears. Correct the data and continue.

The **Cancel** button branches to the *Trace Configurations* window. No Modifications are saved.

The **Help** button calls a window with context oriented help information.

You should specify a meaningful name for this trace configuration in the entry field. This name is shown in the *Trace Configurations* window.

You must specify an output data set for the DB2 trace data according to the rules for data set names on OS/390. As disposition parameter you can select:

**Append** Specifies that new trace data is appended, if the data set already exists.

**Overwrite** Specifies that new trace data overwrites old trace data, if the data set already exists.

In the second page of the *Configurations* (Figure 12-15) window you specify the trace data to be collected.

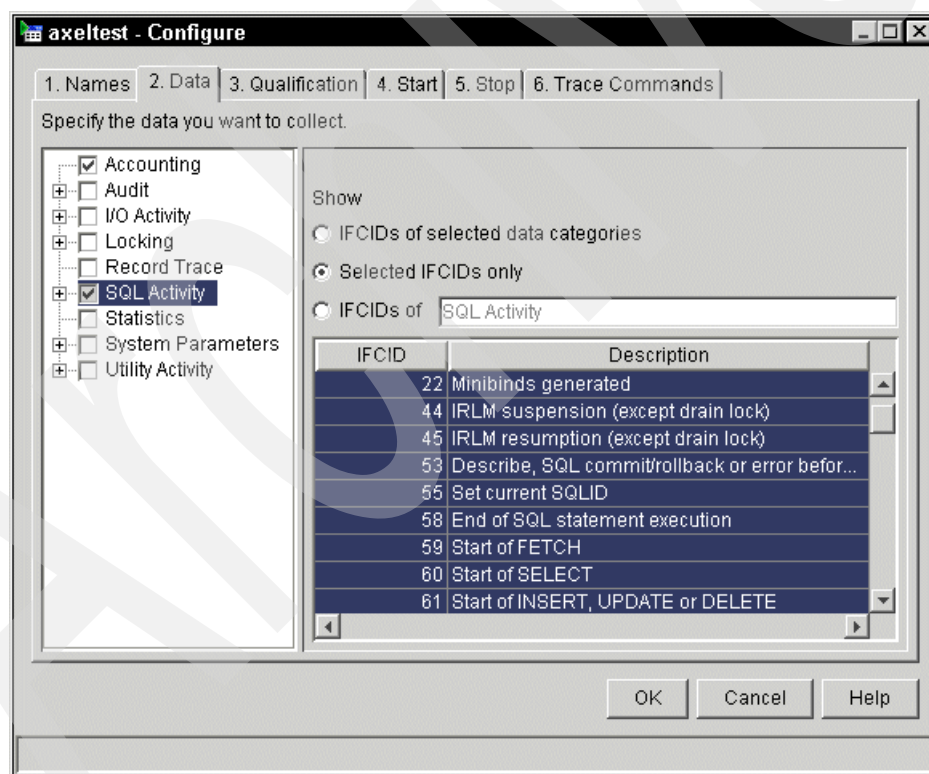


Figure 12-15 Specification of trace data

You can select one or more trace types in the left part of the page. In the right scrolling window you find the IFCIDs for trace types. With the radio buttons you can select which subset of IFCIDs you want to view. Marked IFCIDs belong to the types you selected. You can select or deselect IFCIDs shown. The trace types correspond with DB2 PM reports. If you select a trace type, DB2 PM will by default start traces for all IFCIDs that are required to produce that DB2 PM report.

We select the *SQL Activity* trace type including only the *Base*, *Accounting* and *Locking* subtypes for our test.

The marked IFCIDs are included in the data collection process when the trace is activated. You must mark at least one IFCID on this page to leave this window with the OK button.

On the *Qualifications* page you can specify that the trace data should only be gathered for selected primary authorization IDs, plan names, or requesting locations.

On the next pages you can specify conditions for the start and stop of your trace.

The default condition for starting the trace is the immediate start after the activation. Other possibilities are to start the trace at a certain time or based on defined exception events.

The default condition for stopping the trace is after an elapsed time of 5 minutes. This may be too short for your needs. You can change the time limit for example to 15 minutes as shown in Figure 12-16.

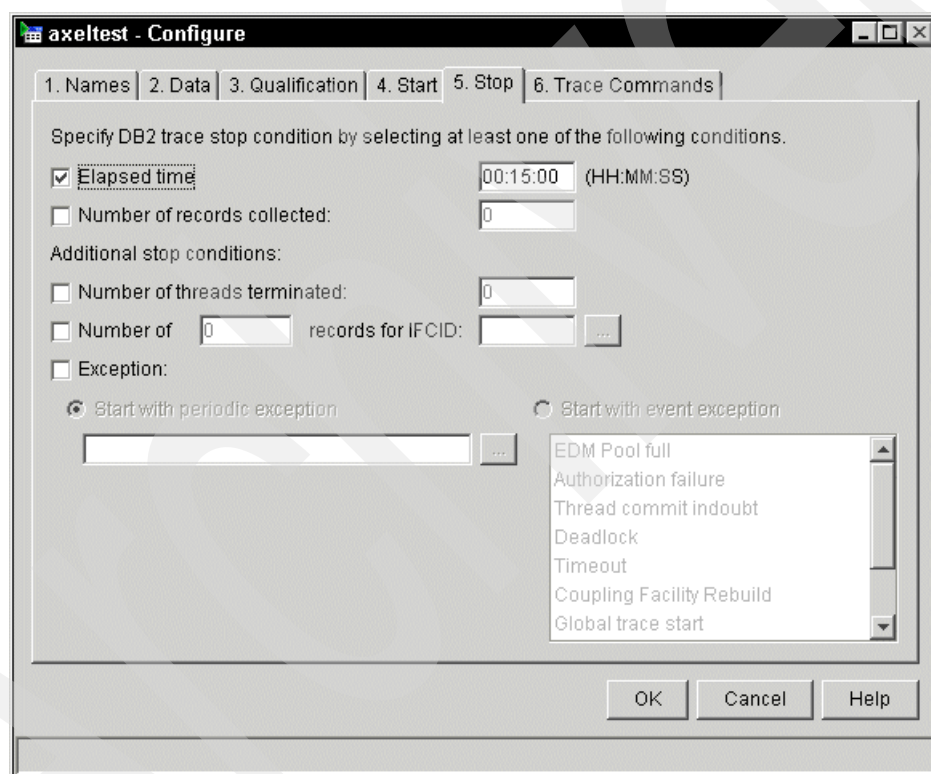


Figure 12-16 Stop conditions for a trace

You can stop the trace

- ▶ After a certain number of records have been collected
- ▶ After a number of threads have terminated
- ▶ After a number of records for a IFCID have been collected
- ▶ Based on a selected exception event

Finally you can review the generated DB2 **START TRACE** commands as shown in Figure 12-17.



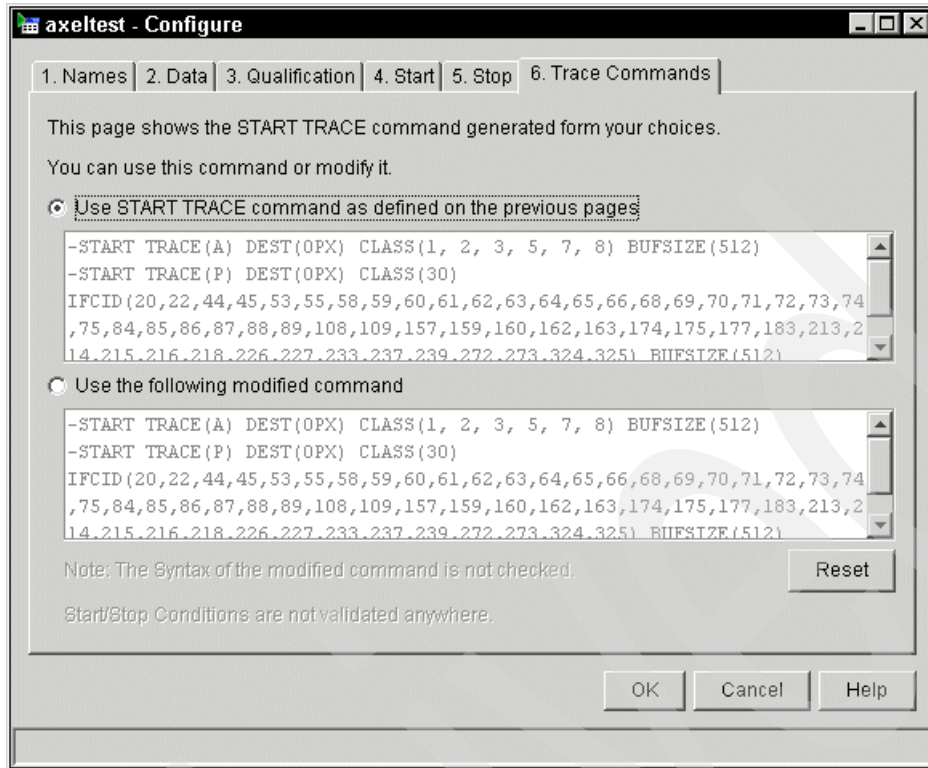


Figure 12-17 Generated START TRACE commands

In this window you find two text fields:

- The DB2 **START TRACE** commands based on the entries you made on the previous windows. You cannot modify the content of this field.
- A copy of the previous commands. You can modify the DB2 **START TRACE** commands in this field directly. If you need fresh copy of the unmodified commands, click the *Reset* button.

If you use the modified trace commands, pages *Data* and *Qualification* are disabled because their contents do not match the modified commands.

### 12.5.3 Activating traces in the workstation monitor

After defining and configuring the trace we can start it. To start a trace, we can click on the *Traces* button in the *System Overview* window (Figure 12-1 on page 182) or on the *Traces* menu in the menu bar. In any other window we find the trace activation function in the *Tools* menu.

Figure 12-18 shows the *Trace Activation* window we use for starting and controlling traces.

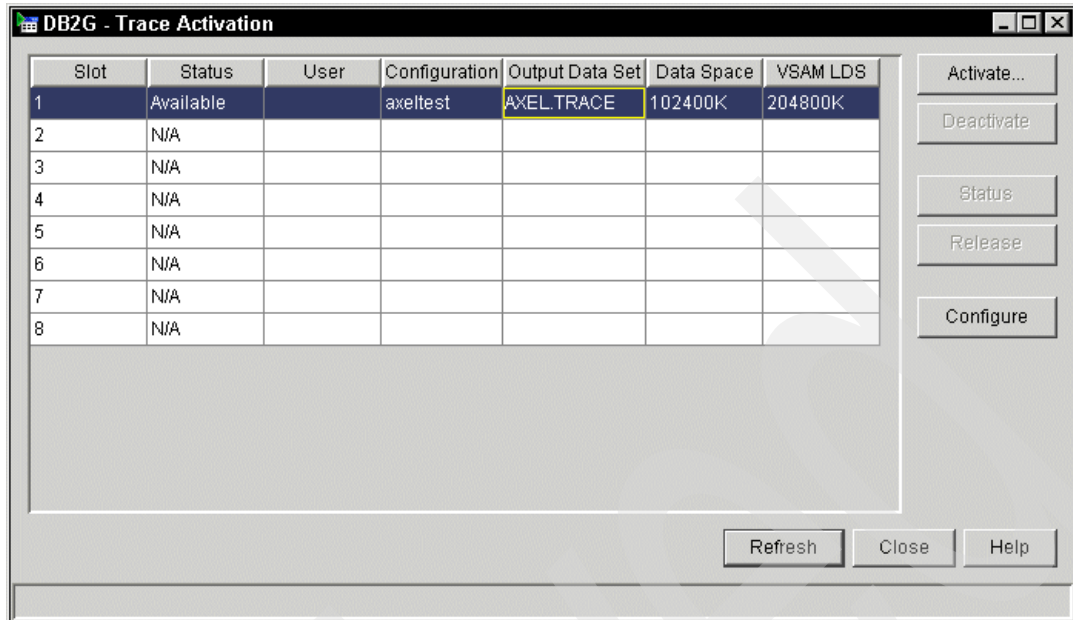


Figure 12-18 Trace activation window

This window shows the status of up to eight slots that are reserved in the DB2 PM *Data Collector* task (running on the host) for trace data. The number depends on the *Data Collector* startup parameters. Our current trace configuration is preselected.

Grayed slots are not selectable. These slots are occupied by traces controlled by other users.

The following information is shown for each slot:

- Status shows the current usage of this slot. Possible values are:

- N/A** The slot is not available because it was not configured at Data Collector startup time.
- Available** The slot is available. A collect report data task can be activated in this slot.
- In use** The slot is currently in use by the user shown in the User field. If this is a different user, you cannot use this slot.
- Collecting** A trace is active. Data collection has started.
- Writing output data** A stop condition was triggered. Data collection stopped. Results are currently written to the output data set.
- Trace stopped** Data collection completed. Results were written to the output data set.
- Start failed** Data collection could not start because of an error condition, for example, an incorrect DB2 START TRACE command, or a lack of authorization.
- Start pending** A trace is active in this slot with a start condition that has not been triggered yet.

- User ID
- Configuration name
- Output data set name
- Data space
- VSAM LDS

Click one of the available slots, or accept the preselected slot with your current trace configuration.

For the selected slot you can use one of the following buttons depending on the status of the slot:

<b>Activate</b>	Opens the Activate Slot dialog box and lets you activate the selected trace configuration in the selected slot
<b>Deactivate</b>	Stops the active trace in the selected slot
<b>Status</b>	Opens the <i>Trace Status</i> window for the selected slot
<b>Release</b>	Releases an occupied slot and makes it generally available
<b>Configure</b>	Opens the <i>Trace Configuration</i> window for the selected trace configuration in the selected slot
<b>Refresh</b>	Updates the information in this window and in all <i>Trace Status</i> windows
<b>Close</b>	Returns to the previous window

To start a trace we activate the selected slot showing our trace configuration. In the following dialog box we confirm the selected trace configuration and the correct output data set shown.

You can change the name of the output data set, if you want to prevent temporarily overwriting an existing data set.

After activating a trace which starts immediately, the *Trace Activation* window in Figure 12-19 shows the trace status as *Collecting*.

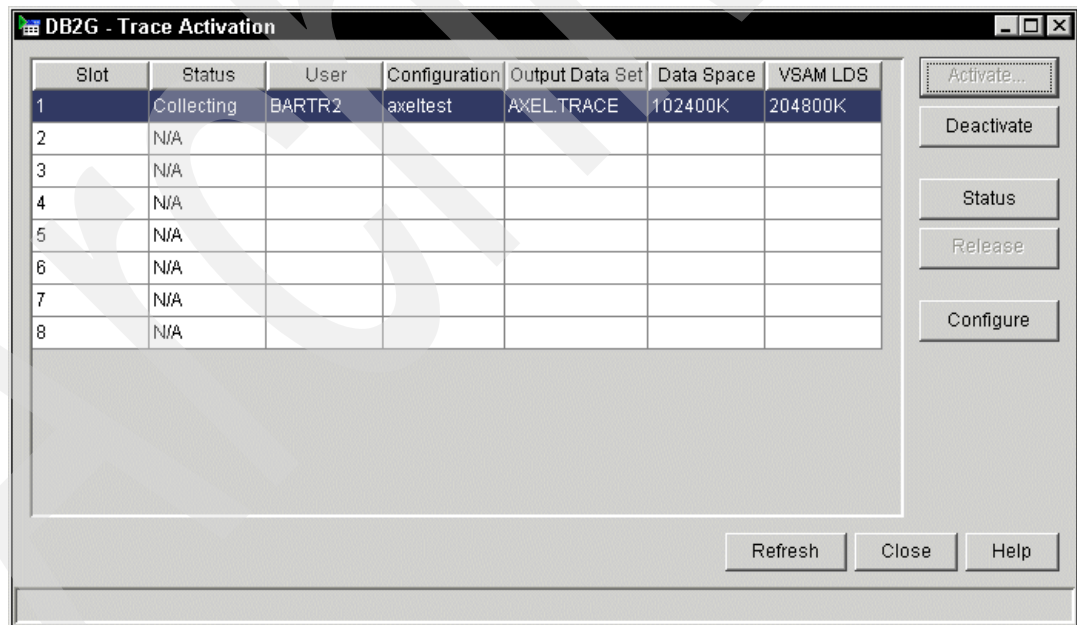


Figure 12-19 Started trace collecting data

You can use the *Status* button to get detailed information about the running trace.

When you *deactivate* the trace, you have to confirm that the collected data should be saved to the configured data set as shown in Figure 12-20.

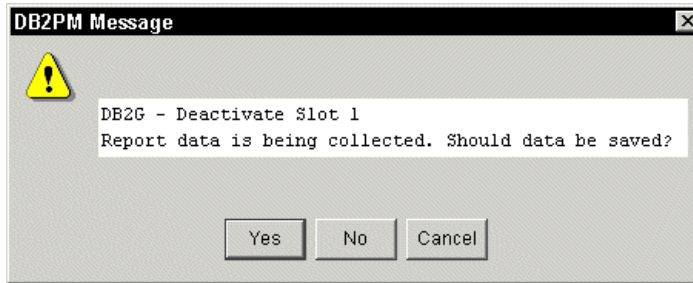


Figure 12-20 Confirmation to save collected data

After terminating, the slot has the status *Data available*, as shown in Figure 12-21.

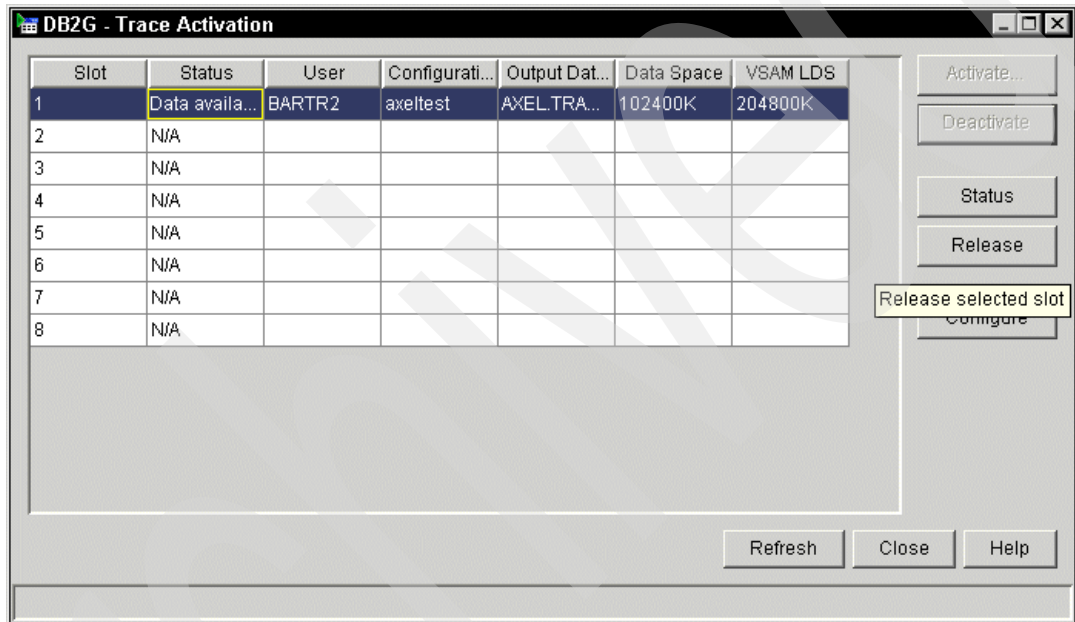


Figure 12-21 Trace status after termination

Before you can reuse this slot, you must release it.

The saved data can now be loaded into the performance database.

#### 12.5.4 Loading trace data into the performance database

The performance database of DB2 PM consists of several tables which are logically grouped by trace types. Here we use the tables of the accounting and the record trace groups. The performance database is not created automatically with the installation of DB2 PM. You have to create it when you use it for the first time. You create another dedicated one for investigating special problems independently of other analysis. You have full control over where and what gets created.

Figure 12-22 shows the data structure of the *accounting group tables*. The main table is the *General Data* table containing the detailed accounting information. There is one row per DB2 accounting record, which is written when the thread terminates, is being reused or becomes inactive.

Note that some tables in the figures describing the performance database layout, have a lighter color. These tables can be created but we do not use them for our analysis.

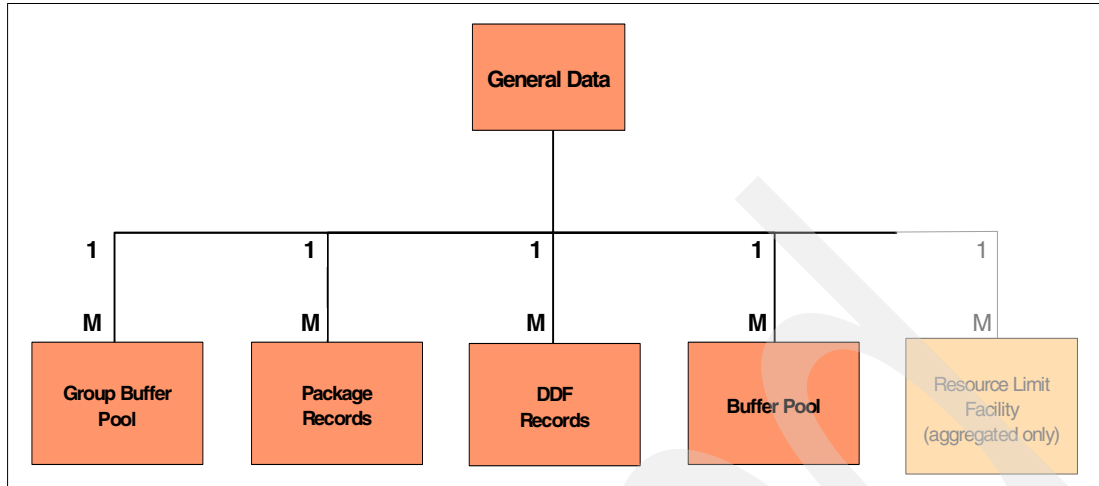


Figure 12-22 The accounting tables of the performance database

Sample table definitions and their descriptions can be found in the following members in the <prefix>.SDGOSAMP data set, shown in Table 12-1:

Table 12-1 Members for table creates and descriptions of the accounting data

Type of data	Creates	Descriptions
General	DGOACFGE	DGOABFGE
Group Buffer Pool	DGOACFGP	DGOABFGP
Buffer Pool	DGOACFBU	DGOABFBU
DDF records	DGOACFDF	DGOABFDF
Package records	DGOACFPK	DGOABFPK

Since we do not use the aggregate data we will not load the RLF table.

Figure 12-23 shows the *Record Trace* group data structure. We concentrate here on the *SQL Statement* table which contains the IFCID 63 records, and the *Miniplan* table which contains the IFCID 22 records.

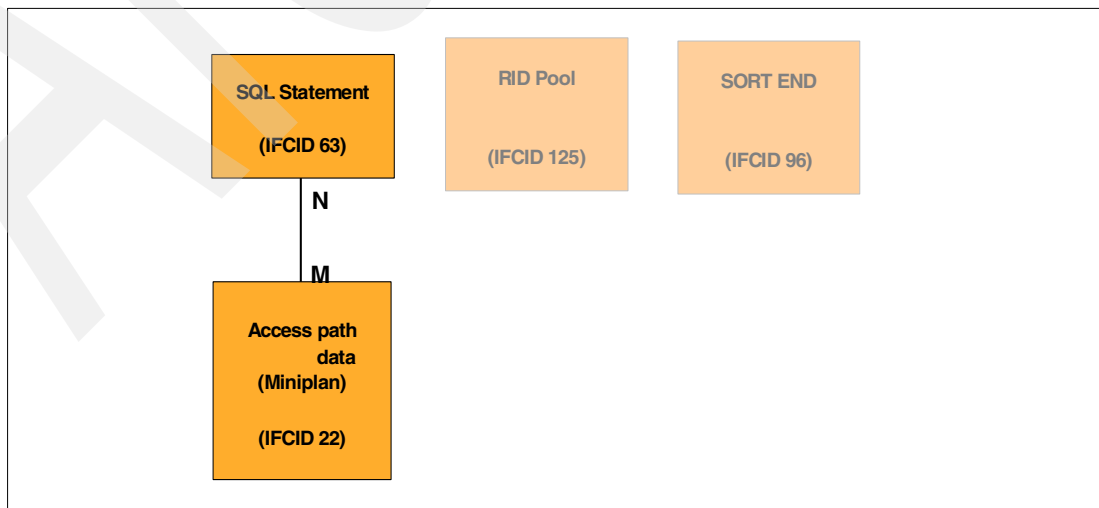


Figure 12-23 The record trace tables of the performance database

Sample table definitions and their descriptions can be found in the following members of the <prefix>.SDGOSAMP data set, shown in Table 12-2:

Table 12-2 Members for table creates and descriptions of the record trace data

Type of Data	Creates	Descriptions
SQL Statement	DGONCFSQ	DGONBFSQ
Miniplan	DGONCFMB	DGONBFMB

Before you can load these tables, the data must be converted into non-aggregated format compatible with the DB2 LOAD utility. The FILE subcommand of the DB2 PM report facility generates a data set containing this non-aggregated data. Refer to *DB2 Performance Monitor for OS/390 Version 7 Report Reference*, SC27-0853 for detailed information about the FILE subcommand. Figure 12-24 shows you the general flow of loading the performance database.

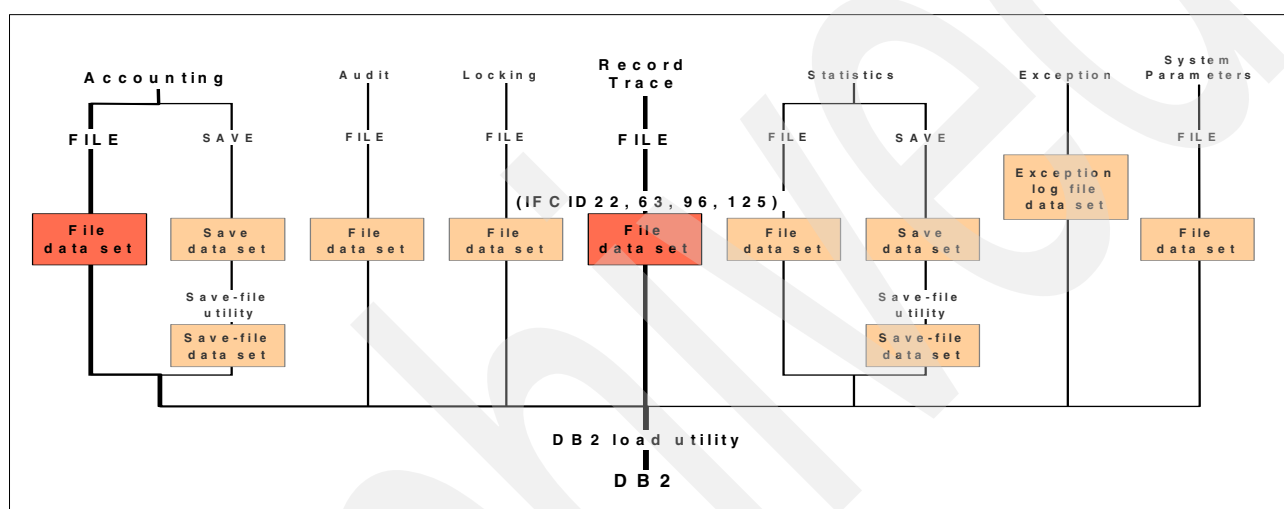


Figure 12-24 Logical flow of loading trace data into the performance database

In Example 12-1 we generate non-aggregated accounting data filtered by the primary authorization IDs of users BARTR2 and BARTR4.

Example 12-1 FILE subcommand for accounting data

```

//SYSIN DD *
DB2PM
* *****
* ACCOUNTING REPORTS
* *****
ACCOUNTING
      FILE
      NOEXCEPTION
      DDNAME(ACFILDD1)
      INCLUDE(AUTHID(BARTR2,BARTR4))
EXEC
  
```

The output data set is sequentially organized.

In Example 12-2 we generate non-aggregated trace record data filtered for the primary authorization IDs of users BARTR2 and BARTR4.

*Example 12-2 FILE subcommand for record trace data*

```
//SYSIN DD *
DB2PM
* *****
* RECORD TRACE REPORTS
* *****
RECTRACE
      FILE
      DDNAME(TRFILDD1)
      INCLUDE(AUTHID(BARTR2,BARTR4))
EXEC
```

Do not use the same output data set as long as you have not loaded the data into the database.

To load this information into the tables of the performance database, you can use load control statements which you find in the members shown in Table 12-3. Be aware that the members contain the REPLACE clause for the load. If you created several tables in one table space, as proposed in the create members, you must change this clause to RESUME YES in order not to delete the data already loaded.

*Table 12-3 Load control members for the performance database*

Type of data	Member name
General	DGOALFGE
Group Buffer Pool	DGOALFGP
Buffer Pool	DGOALFBU
DDF records	DGOALFDF
Package records	DGOALFPK
SQL Statement	DGONLFSQ
Miniplan	DGONLFMB

Example 12-3 shows a sample load job to load general accounting data from a FILEd data set into the DB2 PM performance database.

*Example 12-3 Sample load job for accounting data from FILE data set*

```
//BARTR2LD JOB (999,POK),'DB2 LD',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
//JOBPARM SYSAFF=SC63
//PRCLB JCLLIB ORDER=DB2V710G.PROCLIB
//*
//* STEP 12: LOAD DATA INTO PM DB2PM TABLES
//*
//PH01S12 EXEC DSNUPROC,PARM='DB2G,LDNPEMP'
//SYSUT1 DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SORTOUT DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSREC DD DSN=AXEL.LOAD.ACCT,DISP=OLD
//SYSERR DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSDISC DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSMAP DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSIN DD *
LOAD DATA INDDN(SYSREC)
RESUME YES LOG NO
INTO TABLE DB2PMFACCT_GENERAL
```

WHEN (9:9) = ' '	POSITION(3) SMALLINT,
(DB2PM_REL	POSITION(10) CHAR(2),
DB2_REL	POSITION(47) CHAR(16),
LOCAL_LOCATION	POSITION(63) CHAR(8),
GROUP_NAME	POSITION(71) CHAR(4),
SUBSYSTEM_ID	POSITION(75) CHAR(8),
MEMBER_NAME	POSITION(21) TIMESTAMP EXTERNAL,
TIMESTAMP	POSITION(91) CHAR(8),
NET_ID	POSITION(99) CHAR(8),
LUNAME	POSITION(107) CHAR(12),
INSTANCE_NBR	POSITION(119) SMALLINT,
LUW_SEQNO	POSITION(223) CHAR(16),
REQ_LOCATION	POSITION(239) CHAR(8),
REQ_PRODUCT_ID	POSITION(187) CHAR(8),
CONNECT_TYPE	POSITION(145) CHAR(8),
CONNECT_ID	
.....	

---

### 12.5.5 Analyzing performance data

An accounting record is generated, when:

- ▶ A thread terminates
- ▶ A thread is reused
- ▶ A thread goes inactive
- ▶ A commit (*SRRCMIT*) is issued, for a thread using the *Recoverable Resource Manager Attachment Facility* (RRSAF) that specified “COMMIT” as accounting interval parameter on its SIGNON request

An accounting record is uniquely identified by fully the qualified network name and a Logical Unit of Work (LUW) instance number. You can use this identification to join the related rows of the accounting tables.

In *DB2 Performance Monitor for OS/390 Version 7 Batch User's Guide*, SC27-0857, as well as *DB2 Performance Monitor for OS/390 Version 7 Report Reference*, SC27-0853, you find detailed information, how to interpret the information in the accounting records.

Accounting information can be obtained at the plan level or the package level. When you use the dynamic statement cache, SQL statements that are stored in the cache also maintain accounting related counters.

Appendix , “Comparison of plan, package, and cache information” on page 246 compares the different accounting related counters that are available when gathering DB2 measurement data through standard DB2 traces.

An *SQL Statement* record (IFCID 63) is written each time a prepare, either full or short, is done. It is uniquely identified by the fully qualified network name and a LUW instance number (LUW\_INSTANCE).

*Miniplan* records are written at bind or full prepare time. One miniplan row is loaded into the performance database for each table and access block in a query (corresponding with the number of rows you would have retrieved from the PLAN\_TABLE is you had used EXPLAIN; QBLOCKNO, PLANNO, MIXOPSEQ). Therefore more than one record can be generated for one SQL statement.

**Note:** Using global statement caching miniplans are only generated for a SQL statement when it is inserted into the cache on a full prepare.



Miniplan records are uniquely identified by the fully qualified network name (NETWORK and LUNAME column), a LUW instance number (LUW\_INSTANCE column). The instance number is a timestamp. You can use the fully qualified network name and the instance number to join the miniplan information with the related SQL statements (using the same column names) or the related accounting information (that uses NET\_ID, LUNAME and INSTANCE\_NBR).

If more than one miniplan for the same statement text has been generated by DB2, because several full prepares on the same statement have been executed for some reason, you must use the timestamps as well to join the related row. Example 12-4 shows how you can join accounting data, the statement text and the miniplan using network names and the timestamps (instance number) via a simple SQL statement.

Example 12-4 Sample query

Query:

```
SELECT G.INSTANCE_NBR, G.TIMESTAMP, G.CLASS2_ELAPSED, M.PLANNO,
       T.TIMESTAMP, M.TNAME, M.CREATOR, M.PROGNAME, M.METHOD,
       M.ACCESSTYPE, M.PREFETCH, M.JOINTYPE, T.STATEMENT_TEXT
FROM DB2PMFACCT_GENERAL G, DB2PMFTRC_SQLTEXT T, DB2PMFTRC_MINIPLN M
WHERE G.INSTANCE_NBR = 'B739A4EBA923'
AND G.INSTANCE_NBR = T.LUW_INSTANCE
AND M.LUW_INSTANCE = T.LUW_INSTANCE
AND G.NET_ID = T.NETWORK
AND T.NETWORK = M.NETWORK
AND G.LUNAME = T.LUNAME
AND M.LUNAME = T.LUNAME
ORDER BY M.PLANNO
```

Result:

INSTANCE NBR	TIMESTAMP	CLASS2 ELAPSED	PLANNO			
B739A4EBA923	2002-02-22-01.48.46.872517	0.030661	1			
B739A4EBA923	2002-02-22-01.48.46.872517	0.030661	2			
B739A4EBA923	2002-02-22-01.48.46.872517	0.030661	3			
B739A4EBA923	2002-02-22-01.48.46.872517	0.030661	4			
TIMESTAMP1	TNAME	CREATOR	PROGNAME	METHOD		
2002-02-22-01.48.46.839637	DB2PMFACCT_GENERAL	BARTR2	DSQCFSQL	0		
2002-02-22-01.48.46.839637	DB2PMFRTRC_MINIPLN	BARTR2	DSQCFSQL	1		
2002-02-22-01.48.46.839637	DB2PMFRTRC_SQLTEXT	BARTR2	DSQCFSQL	1		
2002-02-22-01.48.46.839637	DB2PMFACCT_PROGRAM	BARTR2	DSQCFSQL	1		
ACCESSTYPE	PREFETCH	JOINTYPE	STATEMENT TEXT			
R	S	N	SELECT P.INSTANCE_NBR, M.ACCESSNAME, ...			
R	S	I	SELECT P.INSTANCE_NBR, M.ACCESSNAME, ...			
R	S	I	SELECT P.INSTANCE_NBR, M.ACCESSNAME, ...			
R	S	I	SELECT P.INSTANCE_NBR, M.ACCESSNAME, ...			
-----						
FROM DB2PMFACCT_PROGRAM P, DB2PMFRTRC_MINIPLN M, DB2PMFRTRC_SQLTEXT T,						
FROM DB2PMFACCT_PROGRAM P, DB2PMFRTRC_MINIPLN M, DB2PMFRTRC_SQLTEXT T,						
FROM DB2PMFACCT_PROGRAM P, DB2PMFRTRC_MINIPLN M, DB2PMFRTRC_SQLTEXT T,						
FROM DB2PMFACCT_PROGRAM P, DB2PMFRTRC_MINIPLN M, DB2PMFRTRC_SQLTEXT T,						

```

-----
DB2PMFACCT_GENERAL G WHERE T.....
DB2PMFACCT_GENERAL G WHERE T.....
DB2PMFACCT_GENERAL G WHERE T.....
DB2PMFACCT_GENERAL G WHERE T.....

```

The miniplan in Example 12-4 has 4 rows, one for every table access.

For a detailed description of the records refer to *DB2 Performance Monitor for OS/390 Version 7 Report Reference Version 7*, SC27-0854. There you find an explanation of the possible column values as well.

You can relate miniplans to packages using the fully qualified network name, a LUW instance number (LUW\_INSTANCE) and the package name (column PROGRAM).

Figure 12-25 shows the relationship between the tables.

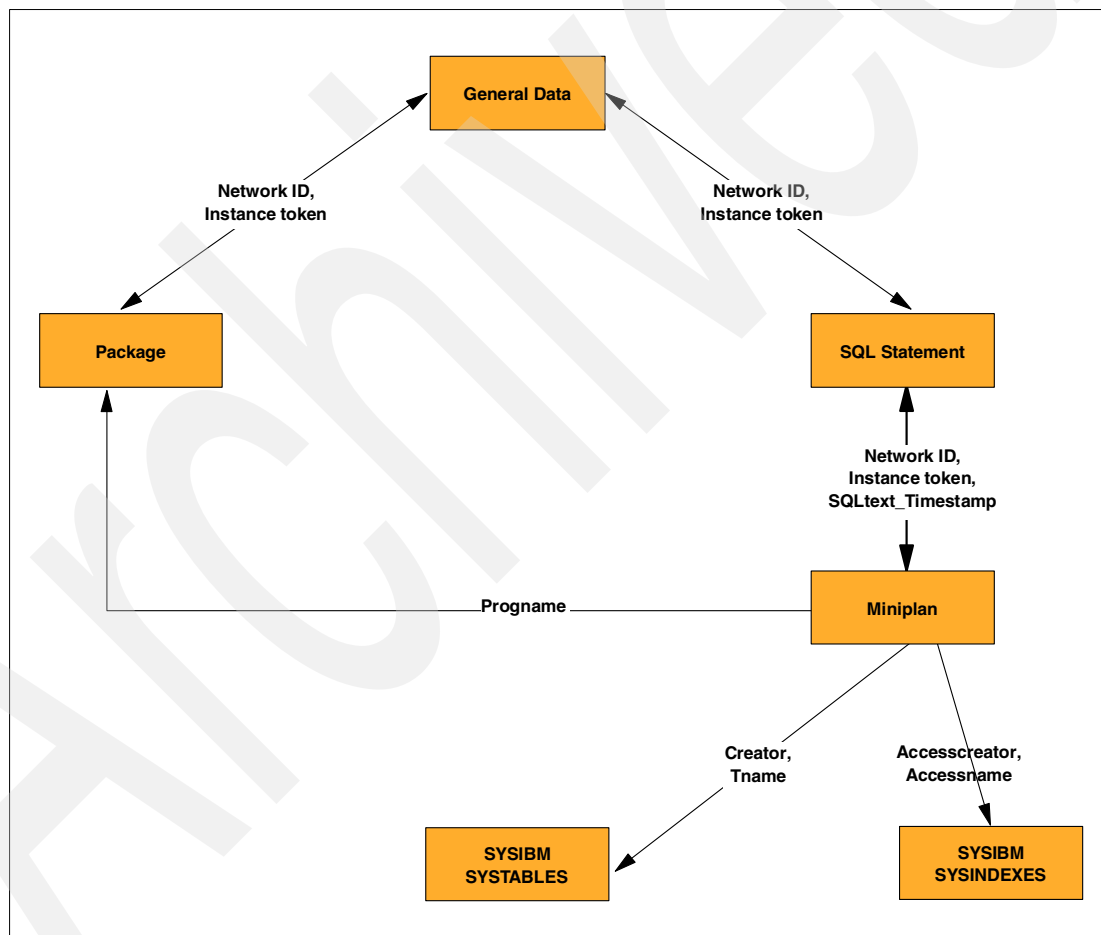


Figure 12-25 Relations between relevant tables of the performance database

You also can relate miniplans and the accessed tables and indexes using the qualified table and index name. You can retrieve indexes belonging to the accessed tables using the qualified table name as TBCREATOR and TBNAME of SYSIBM.SYSINDEXES. This way you can determine whether indexes exist that are not used by any SQL statement.

In *DB2 Performance Monitor for OS/390 Version 7 Batch User's Guide*, SC27-0857 you find detailed information, how to interpret the information in the miniplans and the related records.

If you recognize a performance problem related to application using dynamic SQL statements you can start a trace and load the performance database with the trace data to analyze the reasons for this problem. If you identify the thread causing in the problem, you can identify the statements used by the thread. However, there is no guarantee to find the related miniplans, because the full prepares might be done outside the scope of the trace and the statement is reused from the local or global cache.

In order to make sure you catch the access paths selected by the optimizer (IFCID 22 - miniplan), you must:

1. Start a trace again
2. Invalidate the statements causing in the performance problem in the global cache (for example via RUNSTATS UPDATE NONE)
3. Trace the full prepares
4. Recreate the problem, if possible
5. Stop the trace
6. Add the new trace data to your performance database
7. Analyze the information

Be aware that, since the statements are prepared again after being invalidated, you cannot guarantee that the new access path will be the same as the one that seemed to be causing the problem. But if it is not the case, the new access paths might already help to solve your problem.

This is even more true when dealing with declared temporary tables. Re-explaining a statement that uses a DTT is very likely to produce a different access path than the original statement that had the performance problem. Since the DTT is volatile (and statistics are collected and used on-line) during the incremental bind of the statement at execution time, it is virtually impossible to do an EXPLAIN after the facts.

In conclusion, if your environment relies heavily on dynamic SQL, it is a good idea to trace IFCID 22 and IFCID 63 on a permanent basis.

## 12.6 Analyzing dynamic SQL using batch reports

In case you do not have the DB2 PM performance database available or you just want to analyze trace data to solve a particular performance problem, you can also execute DB2 PM batch jobs to produce the necessary reports that enable you to look at the SQL statements that are executed and the access paths involved.

To obtain the necessary trace data from DB2, similar traces as described in 12.5.1, "Overview" on page 200 need to be activated. To list the SQL statements that have been executed (only the first 4KB is available in IFCID 63) and the access paths they used (miniplan IFCID 22), you can use the DB2 PM command listed in Example 12-5.

*Example 12-5 DB2 PM command to list IFCID 22 and 63*

---

```
RECTRACE
      TRACE
      INCLUDE(IFCID(22,63))
      LEVEL(LONG)
```

---

The result is shown in Example 12-6.

Example 12-6 Listing SQL statements and miniplans

PRIMAUTH	CONNECT	INSTANCE	END_USER	WS_NAME	TRANSACT
ORIGAUTH	CORRNAME	CONNTYPE	RECORD TIME	DESTNO ACE	DATA
PLANNAME	CORRNMBR		TCB CPU TIME	IFC ID	DESCRIPTION
BART	TSO	B791C80BE169	'BLANK'	'BLANK'	'BLANK'
BART	BART		04:14:29.88356070	3132	1 63 SQL STATEMENT
DSNESPCS	'BLANK'		0.38109651		NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1
					OPTIONS: X'04' HOST LANG: N/A SQL STATEMENT
					SELECT * FROM
					DSN8710.EMP E , DSN8710.DEPT D WHERE
					E.WORKDEPT=D.DEPTNO
BART	TSO	B791C80BE169	'BLANK'	'BLANK'	'BLANK'
BART	BART	TSO	04:14:29.88881668	3270	1 22 MINIBIND
DSNESPCS	'BLANK'		0.38559425		NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1
QUERYNO : 1 PLANNAME : DSNESPCS COST : 12 PARALLELISM_DISABLED : N/A QBLOCKNO : 1 COLLID : DSNESPCS PROGNAME : DSNESM68 CONSISTENCY_TOKEN : 149EEA901A79FE48 APPLNAME : N/P WHEN_OPTIMIZE : 'BLANK' OPT_HINT_IDENT : N/P OPTIMIZE_HINTS_USED : N/P UNITS : 0 MILLI_SEC : 0 COST_CATEGORY : N/P PARENT_Q_BLOCKNO : 0 REASON : BIND_TIME: 2002/05/03 00:14:29.880000 VERSION : N/P MEMBER : N/P STATEMENT_TYPE: SELECT TIMESTAMP : 2002/05/03 00:14:29.88 QW0022LC : 0 QW0022GC : 0					
PLANNO : 1 METHOD : FIRST TABLE ACCESSED SORTN_UNIQ : NO SORTC_UNIQ : NO DATABASE : DSN8D71A NEXTSTEP : NESTED-LOOP JOIN SORTN_JOIN : NO SORTC_JOIN : NO OBJECT : 11 ACCESTYPE: TABLE SPACE SCAN (R) SORTN_ORDERBY : NO SORTC_ORDERBY : NO CREATOR : DSN8710 PAGE_RANGE : NO SORTN_GROUPBY : NO SORTC_GROUPBY : NO TNAME : DEPT JOIN_TYPE : NO SORTN_PGROUP_ID : 0 SORTC_PGROUP_ID : 0 CORRELATION_NAME: D MERGE_JOIN_COLS : 0 ACCESS_DEGREE : 0 JOIN_DEGREE : 0 TSLOCKMODE : IS PARALLELISM_MODE: NO ACCESS_PGROUP_ID: 0 JOIN_PGROUP_ID : 0 COLUMN_FN_EVAL : N/A INDEX_NUMBER : 0 PREFETCH : SEQ DIRECT_ROW_ACC : NO PAGES_FOR_TABLE : 1 TAB_CARDINALITY : 14 STARJOIN : NO ACCESS_NAME : N/A ACCESS_CREATOR : N/A INDEXONLY : N/A MATCHCOLS : N/A PREFETCH_INDEX : N/A OPERATION : N/A MIXOPSEQ : N/A QW0022FF: N/A TABLE_TYPE : TABLE (T) QW0022BX: 2 QW0022DX: 0 QW0022LR: X'00000000' QW0022AP: X'00000000' QW0022AG: X'02000000' QW0022ID: 0 QW0022CL: 4295M QW0022TR: X'7D87B4F8' QW0022JP: X'00000000' QW0022A2: X'00000000' QW0022DT: X'00' QW0022P5: X'41E00000' QW0022WF: X'00' QW0022XX: X'00000000000' QW0022DS: X'41E00000' QW0022DR: X'41E00000' QW0022RD: X'41E00000'					
PLANNO : 2 METHOD : NESTED-LOOP JOIN SORTN_UNIQ : NO SORTC_UNIQ : NO DATABASE : DSN8D71A NEXTSTEP : NOT APPLICABLE SORTN_JOIN : NO SORTC_JOIN : NO OBJECT : 18 ACCESTYPE: TABLE SPACE SCAN (R) SORTN_ORDERBY : NO SORTC_ORDERBY : NO CREATOR : DSN8710 PAGE_RANGE : NO SORTN_GROUPBY : NO SORTC_GROUPBY : NO TNAME : EMP JOIN_TYPE : INNER SORTN_PGROUP_ID : 0 SORTC_PGROUP_ID : 0 CORRELATION_NAME: E MERGE_JOIN_COLS : 0 ACCESS_DEGREE : 0 JOIN_DEGREE : 0 TSLOCKMODE : IS PARALLELISM_MODE: NO ACCESS_PGROUP_ID: 0 JOIN_PGROUP_ID : 0 COLUMN_FN_EVAL : N/A INDEX_NUMBER : 0 PREFETCH : SEQ DIRECT_ROW_ACC : NO PAGES_FOR_TABLE : 2 TAB_CARDINALITY : 50 STARJOIN : NO ACCESS_NAME : N/A ACCESS_CREATOR : N/A INDEXONLY : N/A MATCHCOLS : N/A PREFETCH_INDEX : N/A OPERATION : N/A MIXOPSEQ : N/A QW0022FF: N/A TABLE_TYPE : TABLE (T) QW0022BX: 1 QW0022DX: 0 QW0022LR: X'00000000' QW0022AP: X'00000000' QW0022AG: X'02000000' QW0022ID: 0 QW0022CL: 4295M QW0022TR: X'00000000' QW0022JP: X'00000000' QW0022A2: X'00000000' QW0022DT: X'00' QW0022P5: X'42320000' QW0022WF: X'00' QW0022XX: X'00000000000' QW0022DS: X'42320000' QW0022DR: X'41392494' QW0022RD: X'41392494'					
BART	TSO	B791C80BE169	'BLANK'	'BLANK'	'BLANK'
BART	BART	TSO	04:14:29.88912326	3287	1 22 MINIBIND
DSNESPCS	'BLANK'		0.38585688		NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1
QUERYNO : 1 PLANNAME : DSNESPCS COST : 12 PARALLELISM_DISABLED : N/A QBLOCKNO : 0 COLLID : DSNESPCS PROGNAME : DSNESM68 CONSISTENCY_TOKEN : 149EEA901A79FE48 APPLNAME : N/P WHEN_OPTIMIZE : 'BLANK' OPT_HINT_IDENT : N/P OPTIMIZE_HINTS_USED : N/P UNITS : 5 MILLI_SEC : 49 COST_CATEGORY : A PARENT_Q_BLOCKNO : 0 REASON : BIND_TIME: 2002/05/03 00:14:29.880000 VERSION : N/P MEMBER : N/P STATEMENT_TYPE: SELECT TIMESTAMP : 2002/05/03 00:14:29.88 QW0022LC : 0 QW0022GC : 0					

The miniplan record (IFCID 22) shows similar information than is available from the PLAN\_TABLE and DSN\_STATEMNT\_TABLE. When using a performance monitor like DB2 PM, the monitor translates most of the information, that is very cryptic in the PLAN\_TABLE, especially for the users that do not have to deal with the PLAN\_TABLE on a daily basis, into plain English. For example, a METHOD = '1' in the PLAN\_TABLE is nicely spelled out as "Nested Loop Join" in the formatted miniplan record.

The corresponding information in the PLAN\_TABLE and DSN\_STATEMNT\_TABLE for this SQL statement looks like Example 12-7.

Example 12-7 PLAN\_TABLE and DSN\_STATEMNT\_TABLE info

PLAN_TABLE ROWS											
QUERYNO	QBLOCKNO	APPLNAME	PROGNAME	PLANNO	METHOD	CREATOR	TNAME	TABNO	ACCESSTYPE	MATCHCOLS	
1	1		DSNESM68	2	1	DSN8710	EMP	1	R		0
1	1		DSNESM68	1	0	DSN8710	DEPT	2	R		0
ACCESSCREATOR	ACCESSNAME		INDEXONLY	SORTN_UNIQ	SORTN_JOIN	SORTN_ORDERBY	SORTN_GROUPBY				
			N	N	N	N	N	N			
			N	N	N	N	N	N			
SORTC_UNIQ	SORTC_JOIN	SORTC_ORDERBY	SORTC_GROUPBY	TSLOCKMODE	TIMESTAMP						
N	N	N	N	IS	2002050300142988						
N	N	N	N	IS	2002050300142988						
PREFETCH	COLUMN_FN_EVAL	MIXOPSEQ	VERSION								
S			0								
S			0								
COLLID	ACCESS_DEGREE	ACCESS_PGROUP_ID	JOIN_DEGREE	JOIN_PGROUP_ID	SORTC_PGROUP_ID	SORTN_PGROUP_ID					
DSNESPCS											
DSNESPCS											
PARALLELISM_MODE	MERGE_JOIN_COLS	CORRELATION_NAME	PAGE_RANGE	JOIN_TYPE	GROUP_MEMBER						
		E									
		D									
WHEN_OPTIMIZE	QBLOCK_TYPE	BIND_TIME	OPTHINT	HINT_USED	PRIMARY_ACCESSTYPE	PARENT_QBLOCKNO	TABLE_TYPE				
	SELECT	2002-05-03-00.14.29.880000							0	T	
	SELECT	2002-05-03-00.14.29.880000							0	T	
DSN_STATEMNT_TABLE ROWS											
QUERYNO	APPLNAME	PROGNAME	COLLID	GROUP_MEMBER	EXPLAIN_TIME	STMT_TYPE	COST_CATEGORY				
1		DSNESM68	DSNESPCS		2002-05-03-00.14.29.880000	SELECT	A				
PROCMS	PROCSU	REASON									
5		49									

## 12.7 Analyzing deadlocks in a dynamic SQL environment

In a dynamic SQL environment it is not uncommon that deadlocks occur, especially in an environment that allows free form SQL to be entered.

When a deadlock occurs, messages similar to those of Example 12-8 are displayed on the system console. These are the same messages as in a non-dynamic SQL environment. The threads that are involved in the deadlock, with their plan names and correlation-ids are presented in the DSNT375I message. (The DSNT501I message shows the thread that was picked as the victim in the deadlock.) Note that the actual packages that were involved, are not shown in any of the messages. This might change in a future release or version of DB2 for z/OS.

However, since we are dealing with dynamic SQL, in this example we use the DSNTPE2 program (PLAN = DSNTPE71), looking at the statements that make up the plan to determine how we ended up in a deadlock situation, is not very meaningful. Inside the plan we only see the PREPARE and EXECUTE statements and not the actual SQL.

*Example 12-8 Deadlock messages in the system log*

```
DSNT375I  -DB2G PLAN=DSNTEP71 WITH 588
CORRELATION-ID=BARTTEP1
CONNECTION-ID=BATCH
LUW-ID=USIBMSC.SCPDB2G.B791C827F564=270
THREAD-INFO=BART:*.*:
IS DEADLOCKED WITH PLAN=DSNTEP71 WITH
CORRELATION-ID=BARTTEP2
CONNECTION-ID=BATCH
LUW-ID=USIBMSC.SCPDB2G.B791C82B78C0=271
THREAD-INFO=BART:*.*:
ON MEMBER DB2G

DSNT501I  -DB2G DSNILMCL RESOURCE UNAVAILABLE 589
CORRELATION-ID=BARTTEP1
CONNECTION-ID=BATCH
LUW-ID=*
REASON 00C90088
TYPE 00000302
NAME DSN8D71A.DSN8S71D.X'000002'
```

Example 12-9 shows the DB2PM deadlock record (IFCID 172), but with this information, even though it shows all resources involved in the deadlock, you are still not able to determine which sequence of events that lead up to the deadlock.

*Example 12-9 Deadlock record (IFCID 172)*

PRIMAUTH	CONNECT	INSTANCE	END_USER	WS_NAME	TRANSACT
ORIGAUTH	CORRNAME	CONNTYPE	RECORD TIME	DESTNO ACE IFC	DATA
PLANNAME	CORRNMBR		TCB CPU TIME	ID	DESCRIPTION
BART	BATCH	B791C827F564	'BLANK'	'BLANK'	'BLANK'
BART	BARTTEP1	TSO	04:15:21.92376714	4011 4 172	DEADLOCK DATA
DSNTEP71	'BLANK'		0.01966186		NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1

DEADLOCK HEADER	
INTERVAL COUNT:	67814 WAITERS INVOLVED: 2
TIME DETECTED: 05/03/02 04:15:21.920533	

UNIT OF WORK	
R E S O U R C E	
LOCK RES TYPE: DATA PAGE LOCK	OBID: DSN8S71E RESOURCE ID: X'00001200'
LOCK HASH VALUE: X'00019412'	

B L O C K E R	
PRIMAUTH : BART	PLAN NAME: DSNTEP71
NETWORKID: USIBMSC	LUNAME: SCPDB2G
MEMBER : N/A	DURATION: COMMIT

CORR ID: BARTTEP1	
UNIQUENESS VALUE: X'B791C827F564'	OWNING WORK UNIT: 117
STATE: EXCLUSIVE	ACE: 4

```

TRANSACTION : 'BLANK'
STATUS      : HOLD
QW0172HF   : X'12'

WS_NAME: 'BLANK'
END_USER: 'BLANK'

W A I T E R
CORR ID: BARTTEP2
CONN: BATCH
UNIQUENESS VALUE: X'B791C82B78C0' OWNING WORK UNIT: 94
STATE: SHARED
ACE: 5
WS_NAME: 'BLANK'
END_USER: 'BLANK'
DB2S ASIC: 77 REQ WORK UNIT: 94
EB PTR: X'0D645E98' REQ FUNCTION: LOCK
WORTH: X'12'
QW0172WG   : X'30'

LOCK RES TYPE: DATA PAGE LOCK
LOCK HASH VALUE: X'00041402'

R E S O U R C E
DBID: DSN8D71A
OBID: DSN8S71D RESOURCE ID: X'00000200'

B L O C K E R
CORR ID: BARTTEP2
CONN: BATCH
UNIQUENESS VALUE: X'B791C82B78C0' OWNING WORK UNIT: 94
STATE: EXCLUSIVE
ACE: 5
WS_NAME: 'BLANK'
END_USER: 'BLANK'

W A I T E R
CORR ID: BARTTEP1
CONN: BATCH
UNIQUENESS VALUE: X'B791C827F564' OWNING WORK UNIT: 117
STATE: SHARED
ACE: 4
WS_NAME: 'BLANK'
END_USER: 'BLANK'
DB2S ASIC: 77 REQ WORK UNIT: 117
EB PTR: X'0D6475D8' REQ FUNCTION: LOCK
WORTH: X'11'
QW0172WG   : X'30'

```

Again IFCID 63 comes to the rescue. Using the LUWIDs of the threads involved in the deadlock, we can look up the SQL statements that they executed.

You can use the DB2 PM command input stream of Example 12-10 to extract the SQL statement text of the two threads involved:

*Example 12-10 DB2 PM command to list SQL statements for a list of threads*

```

RECTRACE
TRACE
INCLUDE(IFCID(63),
        INSTANCE(
            B791C827F564,
            B791C82B78C0
        )
    )
LEVEL(LONG)

```

The output is shown in Example 12-11. When analyzing the information, it becomes immediately clear what causes the deadlock. Job BARTTEP1 inserts a row into the EMP table. BARTTEP2 inserts a row into the DEPT table. Then BARTTEP2 wants to select all rows from the EMP table (but runs into to the lock held by the newly inserted row from BARTTEP1) whereas BARTTEP1 wants to read all rows from the DEPT table and gets blocked by the lock held by BARTTEP2's newly inserted row.

*Example 12-11 Listing the SQL statements issued by a list of threads*

PRIMAUTH	CONNECT	INSTANCE	END_USER	WS_NAME	TRANSACT
ORIGAUTH	CORRNAME	CONNTYPE	RECORD TIME	DESTNO ACE IFC	DATA
PLANNAME	CORRNMBR		TCB CPU TIME	ID	
BART	BATCH	B791C827F564	'BLANK'	'BLANK'	'BLANK'
BART	BARTTEP1	TSO	04:14:59.24625158	3439 1 63	SQL STATEMENT
DSNTEP71	'BLANK'		0.01602907		

NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1  
 OPTIONS: X'04' HOST LANG: N/A SQL STATEMENT:  
 INSERT INTO DSN8710.EMP VALUES ( '000999', 'BART', 'J',  
 'AAS', 'A00', '3978', '1965-01-01', 'PRES', 18,  
 'F', '1933-08-14', 800000, 1000, 4220 )

BART	BATCH	B791C82B78C0	'BLANK'	'BLANK'				'BLANK'
BART	BARTTEP2	TS0	04:15:02.93008493	3511	2	63	SQL STATEMENT	NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1
DSNTEP71	'BLANK'		0.01850309					OPTIONS: X'04' HOST LANG: N/A SQL STATEMENT:
			04:15:14.35900871	3654	2	63	SQL STATEMENT	INSERT INTO DSN8710.DEPT VALUES ( 'A99' , 'TESTJE' ,
			0.02133773					'000010' , 'A00' , 'HOME' )
								'BLANK'
								NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1
								OPTIONS: X'04' HOST LANG: N/A SQL STATEMENT:
BART	BATCH	B791C827F564	'BLANK'	'BLANK'				SELECT * FROM DSN8710.EMP
BART	BARTTEP1	TS0	04:15:14.35998483	3691	1	63	SQL STATEMENT	'BLANK'
DSNTEP71	'BLANK'		0.01850998					NETWORKID: USIBMSC LUNAME: SCPDB2G LUWSEQ: 1
								OPTIONS: X'04' HOST LANG: N/A SQL STATEMENT:
								SELECT * FROM DSN8710.DEPT

Sometimes the reason for the deadlock is less obvious than the example shown above. In those cases the mini-plan can also be very useful to help determining the reason for the deadlock. Accessing the same object in a different way (using a different access path) by two SQL statements can lead to a deadlock. When using dynamic SQL analyzing IFCID 22 (miniplan) will show that the access path chosen by both statements is different; for example query1 is using a table space scan, whereas query2 is using an index to update the data and this leads up to the deadlock. Without the access path information used by the statements you can only guess how we ended up in this deadlock situation.

A similar analysis can be conducted for timeouts (IFCID 196).

## 12.8 Identifying users to track resource usage

Monitoring tools such as SMF, RMF, DB2 PM, and reporting tool such as Performance Reporter are available to collect, store and exploit information about resources usage.

Being able to identify end-users in the system is essential to make a rigorous query/user segmentation or to implement a charge-back accounting application. Unfortunately, remote applications often connect to DB2 with a general userid. If you want to segregate end-users in remote applications using DRDA, you need to be able to identify them in a better way.

### Identifying remote users

DRDA allows to send four types of data to DB2 for z/OS, to help in the identification of a user or an application:

- ▶ A client accounting string.
- ▶ A string to identify the client application.
- ▶ A string to identify the client workstation name.
- ▶ A string to identify the client userid. This string is for identification only; it is not a DB2 authorization-id.

These strings can be used for accounting or monitoring purposes. The first string is reported in the QMDA section of the DB2 accounting records. The three other strings are reported in identification fields QWHCEUTX, QWHCEUWN and QWHCEUID. For more information about reporting based on these fields, see *DB2 Performance Monitor for OS/390 Version 7 Report Reference*, SC27-0853.



This information can be passed to DB2 by one of the following methods:

- ▶ A call to the *sqlesact* API can be used to set the client accounting string. This string can also be set via the “Client Settings” button in the Client Configuration Assistant (DB2 parameter = `dft_account_str`).  
If none of these methods is applied, the DB2ACCOUNT registry value is used as a default accounting string.
- ▶ A call to the *sqleseti* API can be used to pass the 3 other strings.
- ▶ Via DB2 CLI, the four strings can be set via a call to the `SQLSetConnectAttr()`.

For more information about these functions, you can refer to *DB2 Connect User's Guide*, SC09-2954, *DB2 Universal Database Administrative API Reference*, SC09-2947, and *DB2 Universal Database Call Level Interface Guide and Reference*, SC09-2950.

Archived



## Security aspects of dynamic SQL

One of the primary reasons cited for avoiding dynamic SQL is the potential security exposure and the extra burden of maintaining authorizations. In this chapter we discuss the security aspects of using dynamic SQL. We examine the options and possible solutions to limit the security exposure in a server-based, client/server and Web-enabled environment.

## 13.1 Is there a security exposure with dynamic SQL?

Consider an application program that contains the following *embedded static SQL statement*:

```
UPDATE    DSN8710.EMP
SET       SALARY = SALARY + :RAISE
WHERE     EMPNO = :EMPNO
```

A user running this program needs to have only an *execute authority on the plan or package* that contains this program. No explicit authorization to update the EMP table directly is needed and typically not granted. This means that the user can issue this statement form *within* the application but cannot do so form *outside* the application (such as through SPUFI or QMF).

Now consider a similar program that contains a similar *embedded dynamic SQL statement*:

```
UPDATE    DSN8710.EMP
SET       SALARY = SALARY + ?
WHERE     EMPNO = ?
```

In this case, the user needs *execute authority on the plan or package* (as before). Assuming we use the default bind option of DYNAMICRULES(RUN), the user requires an *explicit authorization to issue the update on the specific table*. This has two implications:

- ▶ All authorizations for users on the objects they access via dynamic SQL must be maintained, creating *extra work* for the DBA that is not needed using static SQL
- ▶ Such users can bypass the application and issue the statements directly *outside of the application* (such as via SPUFI or QMF, if they are authorized to use the SPUFI or QMF plan).

In the rest of the chapter we discuss alternatives that address these concerns.

## 13.2 Using DYNAMICRULES(BIND)

In Section 7.4, “DYNAMICRULES option” on page 134, we discussed the various options for this bind parameter and their meaning. Loosely speaking, using DYNAMICRULE(BIND) makes dynamic SQL appear like static from an authorization perspective. A program bound with DYNAMICRULES(BIND) uses the authorization of the package or plan owner to determine whether or not the operation is permitted. For example above, if the owner of the package has authority to update the EMP table, so does the user executing the package.

In a server-only environment, or any environment where the user cannot issue any free-form SQL (that is, all generated SQL is controlled by the application), this option is easy to implement and virtually eliminates the security exposure. When a user is allowed to issue any free-form SQL, care must be taken to make sure that the package is bound by a user who has *just the sufficient authority*. For example, if the package is bound by a user with the SYSADM authority, *all* dynamic SQL will be authorized — equivalent to granting access to all objects to PUBLIC, which is probably not the intent. This is discussed in detail in 13.5, “Controlling security without an application server” on page 223.

## 13.3 Using stored procedures to overcome security issues

If a stored procedure containing only static SQL is used to replace an ODBC application, the only authorization that a user needs is the ability to execute the stored procedure package (EXECUTE authorization on the package) and the ability to execute the stored procedure (EXECUTE ON PROCEDURE). This means no grants to the base objects (tables) need to be issued, nor can the user access the base objects outside of the application. In general, this option is not always possible. The stored procedure may require the use of dynamic SQL embedded in a stored procedure (see 13.8, “Security for dynamic SQL in stored procedures” on page 227 for the security implications of this).

## 13.4 Impact on ODBC/JDBC applications

Like applications using embedded SQL, applications using JDBC or ODBC need DB2 packages to run. There is however a major difference between ODBC/JDBC packages and packages generated from an embedding application:

- ▶ In the embedded approach, the packages are tied to the application and are not usable outside the programs which they are created for
- ▶ In ODBC or JDBC applications, the packages are generic and can be used by *any* C, C++ (for ODBC) or Java (for JDBC) programs

For this reason, binding DB2 ODBC or JDBC packages with the DYNAMICRULES(BIND) option does not provide the same security level as obtained with embedded SQL.

More precisely, when using this bind option, you must be aware of the following:

- ▶ Users of an ODBC/JDBC application inherit the DB2 authorizations of the package owner, each time they use the ODBC or JDBC interface.
- ▶ As a consequence, they can create their own ODBC/JDBC programs, or use any ODBC/JDBC enabled tool to do any data manipulations that the ODBC/JDBC package owner is authorized to do.
- ▶ Moreover, if two applications use the same ODBC/JDBC packages, users of one application can use programs or tools to manipulate data belonging to the other application.

To solve the latter problem, you can isolate your ODBC applications by creating a set of ODBC/JDBC packages for each application:

- ▶ At bind time, choose a collection name that represents the application, and choose an bind owner who has the right authorizations on the application objects.
- ▶ At runtime, use the ODBC keyword COLLECTIONID (on z/OS) or CURRENTPACKAGESET (on NT) to point to the correct packages.

A more detailed analysis of the possible security issues in ODBC/JDBC applications can be found in 13.5, “Controlling security without an application server” on page 223 and 13.6, “Controlling security with an application server” on page 226.

## 13.5 Controlling security without an application server

The Table 13-1 summarizes the possible security issues when an application containing dynamic SQL statements is directly run by the end users, without passing through an application server. More precisely, the analysis is based on the following three assumptions:

- ▶ The application does not call stored procedures or user-defined functions. The security implication of this are dealt with in 13.3, “Using stored procedures to overcome security issues” on page 223 and 13.8, “Security for dynamic SQL in stored procedures” on page 227.
- ▶ End users are directly connected to DB2 to run the SQL processor program. They are consequently granted the authorization to EXECUTE the DB2 package or plan.
- ▶ The SQL processor implements either:
  - A generic interface allowing any free form SQL statements without any programmed control. Packages as JDBC, ODBC, QMF and SPUFI belong to this category.
  - A restricted interface that controls the SQL statements submitted to the database.

Table 13-1 Security implications without application server

Does the SQL processor allow free-form SQL?	Does the user have DB2 authorization to objects accessed?	DYNAMIC-RULES bind option	Security implications	Recommendations
YES	YES	RUN	<ul style="list-style-type: none"> <li>▶ Within SQL processor: access to authorized objects but any SQL allowed</li> <li>▶ Outside SQL processor: access to authorized objects but any SQL allowed</li> </ul>	<ul style="list-style-type: none"> <li>▶ Access from outside the SQL processor is a security hole</li> </ul>
YES	YES	BIND	<ul style="list-style-type: none"> <li>▶ Within SQL processor: access to objects bind owner is authorized for but any SQL allowed</li> <li>▶ Outside SQL processor: access to authorized objects but any SQL allowed</li> </ul>	<ul style="list-style-type: none"> <li>▶ Access from within the SQL processor is probably too permissive since bind owner will typically have all-inclusive access</li> <li>▶ Access from outside SQL processor is a security hole</li> </ul>
YES	NO	RUN	<ul style="list-style-type: none"> <li>▶ No access from within or outside SQL processor</li> </ul>	<ul style="list-style-type: none"> <li>▶ Meaningless</li> </ul>
YES	NO	BIND	<ul style="list-style-type: none"> <li>▶ Within SQL processor: access to objects bind owner is authorized for but any SQL allowed</li> <li>▶ Outside SQL processor: no access</li> </ul>	<ul style="list-style-type: none"> <li>▶ Access from within the SQL processor is probably too permissive since bind owner will typically have all-inclusive access</li> <li>▶ No access from outside SQL processor</li> </ul>

Does the SQL processor allow free-form SQL?	Does the user have DB2 authorization to objects accessed?	DYNAMIC-RULES bind option	Security implications	Recommendations
NO	YES	RUN	<ul style="list-style-type: none"> <li>▶ Within SQL processor: access to authorized objects and SQL controlled by SQL processor</li> <li>▶ Outside SQL processor: access to authorized objects and any SQL allowed</li> </ul>	▶ Access from outside SQL processor is a security hole
NO	YES	BIND	<ul style="list-style-type: none"> <li>▶ Within SQL processor: access to objects bind owner is authorized for and SQL controlled by SQL processor</li> <li>▶ Outside SQL processor: access to authorized objects and any SQL allowed</li> </ul>	▶ Access from outside SQL processor is a security hole
NO	NO	RUN	▶ No access from within or outside SQL processor	▶ Meaningless
NO	NO	BIND	<ul style="list-style-type: none"> <li>▶ Within SQL processor: access to objects bind owner is authorized for and SQL controlled by SQL processor</li> <li>▶ Outside SQL processor: no access</li> </ul>	▶ Most secure

From Table 13-1 the following conclusions can be drawn:

- ▶ If the SQL processor does not use a generic interface like ODBC or JDBC and does not use any application like QMF or SPUFI that allows free-form SQL, a high level of security can be obtained by the DYNAMICRULES(BIND) option if the user is not given any explicit authorization on application tables. The security model is similar to the one for static SQL.
- ▶ If the SQL processor makes use of a generic interface like ODBC or JDBC or allows free form SQL, there is *no* secure implementation based on any combination of authorizations and bind options. This is the reason for incorporating an application server, which is discussed in the following section.

## 13.6 Controlling security with an application server

In a 3-tier architecture, the end users access an application server which is connected to the database server. The application server checks the user's authorization and connects to the database server using a set of common IDs different from the user IDs. By avoiding giving any authorization to end users, you can implement a secure application, as long as the userid and password of the application server and the package binder are kept secure. This implementation is shown in the Figure 13-1:

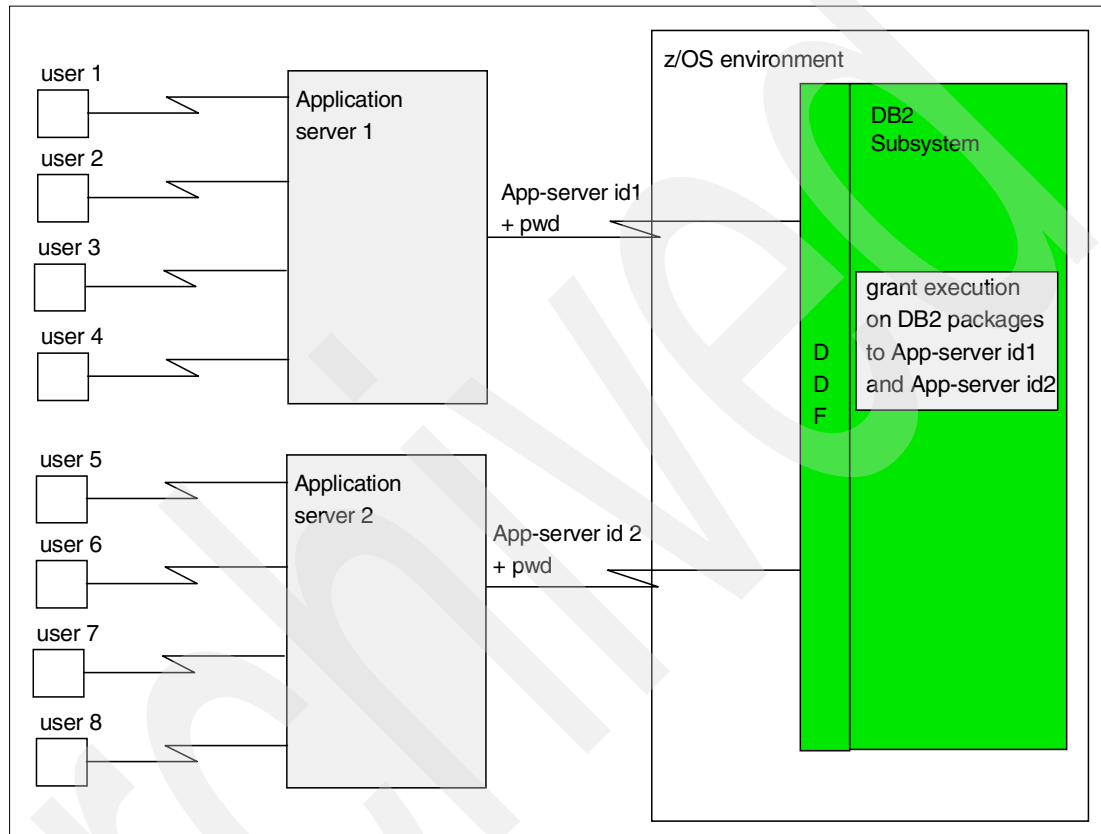


Figure 13-1 Using application servers to improve security

## 13.7 Using network facilities to control database access

In a 3-tier architecture (client services on the PC, accessing an application server which itself accesses the database server), the application is the only layer which needs to reach the IP address and port number of the DB2 server, and fire-walls can be used to stop any other intruders.

SNA offers even more security facilities. For more information about controlling remote connections to a DB2 server, see chapter 12 of *DB2 Universal Database for OS/390 and z/OS Administration Guide*, SC26-9931.



## 13.8 Security for dynamic SQL in stored procedures

Table 13-2 shows how DYNAMICRULES and the run-time environment affect dynamic SQL statement behavior when the statement is in a package that is invoked from a stored procedure (or user-defined function).

Table 13-2 How is run-time behavior determined?

DYNAMICRULES value	Stored procedure or user-defined function environment	Authorization ID
BIND	Bind behavior	Plan or package owner
RUN	Run behavior	Current SQLID
DEFINEBIND	Define behavior	Owner of user-defined function or stored procedure
DEFINERUN	Define behavior	Owner of user-defined function or stored procedure
INVOKEBIND	Invoke behavior	Authorization ID of invoker
INVOKERUN	Invoke behavior	Authorization ID of invoker

Section 7.4, “DYNAMICRULES option” on page 134, discussed the various options for this bind parameter and their meaning in more detail.

Figure 13-2 below shows how complex the choices can be when invoking a stored procedure:

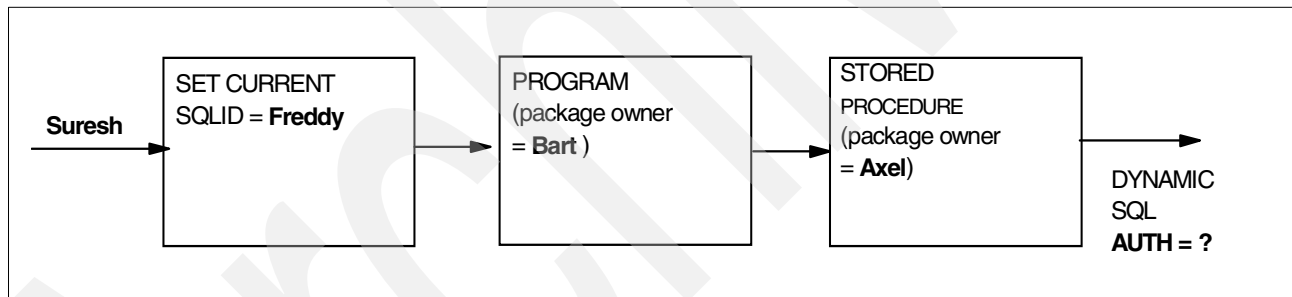


Figure 13-2 Security implications of dynamic SQL in a stored procedure

Consider a user, Suresh, using a current SQLID of Freddy, who executes a package bound by Bart. This package calls a stored procedure bound by Axel. Whose authority is checked at run time? Depending on the option chosen, the authorization ID used to determine whether or not the execution of dynamic SQL within this stored procedure is permitted could be:

- ▶ Suresh (invoker)
- ▶ Freddy (current SQLID)
- ▶ Bart (owner of package) or
- ▶ Axel (owner of stored procedure)

Due to the variety of options available, it is difficult to make a general recommendation that applies to all situations.

## 13.9 Summary

It is clear that a certain amount of “security exposure” is inherent in the use of SQL processors that allow free form dynamic SQL. By being aware of this exposure, you can design applications and incorporate controls to minimize (even eliminate) this exposure.



## Remote dynamic SQL

In this chapter we explain the usage and particularities of dynamic SQL in distributed environments. We discuss the following topics:

- ▶ Protocols supported by remote SQL
- ▶ Reusing database access threads
- ▶ Minimizing message flows across the network
- ▶ Miscellaneous considerations

## 14.1 Protocols supported by remote SQL

Remote applications can connect to DB2 using either DRDA or the DB2 private protocol. The choice is done by DB2, based on the bind option DBPROTOCOL or the system default protocol (DBPROTCL) set in the DSNZPARM parameters module. The private protocol is not evolving, and the use of DRDA is highly recommended.

The DRDA protocol is characterized by the following:

- ▶ DRDA is the Open Group standard protocol for distributed connectivity. Any system that supports DRDA can accept SQL statements issued in the DRDA format.
- ▶ It supports SNA as well as TCP/IP.
- ▶ It allows the execution of any SQL statement that the server supports.
- ▶ It supports remote access through three part names or alias (starting with DB2 Version 6), and explicit connections via the CONNECT statement.
- ▶ It supports the execution of packages of dynamic or static SQL statements to remote DB2 servers.

The DB2 private protocol has the following characteristics:

- ▶ It is a private protocol. The requesters and servers must both be DB2 subsystems on OS/390 or z/OS.
- ▶ It only supports SNA.
- ▶ It supports only DML statements: INSERT, UPDATE, DELETE, SELECT, OPEN, FETCH and CLOSE. It does not allow the invocation of user-defined functions or stored procedures, the use of LOBs or distinct types.
- ▶ It only supports three-part names, and does not support explicit connections. .

**Important:** Using the DB2 private protocol, SQL statements coming from a remote requester are *always* dynamically bound at the DB2 server. The statement is bound the first time it is executed in the transaction. The prepared statement is kept until the end of the transaction, but must be rebound after a COMMIT or a ROLLBACK. These so-called deferred dynamic SQL statements used by the DB2 private protocol do not benefit from the dynamic statement cache.

## 14.2 Reusing database access threads

An *allied thread* represents a work started at the local DB2. The parameter CTHREAD fixes the maximum number of allied threads that can be concurrently allocated.

A *database access thread* represents work requested by a remote application using either DRDA or the DB2 private protocol. It is created when the SQL statement is received and runs in the DDF address space. The maximum number of database access threads that are concurrently allowed is determined by the parameter MAXDBAT.

The total number of *active threads*, the sum of the values assigned to CTHREAD and MAXDBAT, cannot exceed 2000.

Large distributed systems need to manage much more than 2000 connections. For this reason, DB2 allows database access threads to become inactive. DB2 manages inactive threads separately from the active threads, and permits a larger amount of inactive, connected threads. You enable the use of inactive threads by setting the DSNZPARM CTMSTAT to “inactive”. The maximum number of concurrent *remote connections* is determined by the parameter CONDBAT and can have a value up to 150000.

After the end of a transaction (i.e. after the COMMIT or ROLLBACK), DB2 checks if the thread can become inactive. If so, the thread is set to an “inactive” status, for possible reuse:

- ▶ When the DB2 private protocol is used, the thread becomes a “type 1 inactive” thread. In this case, the thread can only be reused by the same connection.
- ▶ When DRDA is used, the thread becomes a “type 2 inactive” thread. In this case, the thread and the connection are split. The thread goes in a pool of inactive threads, and can be reused for another connection. When the inactive connection receives a new request, a database access thread from the pool is assigned to the connection. If all threads are assigned, the connection must wait for the availability of a thread.

A database access thread that has a held cursor, or a held LOB locator cannot become inactive, except after a ROLLBACK, which releases the lock status.

A database access thread that has a declared temporary table that is not dropped before commit, cannot become inactive.

The major point about database access threads with remote dynamic SQL is the following:

**Important:** A database access thread that touched a package bound with KEEP\_DYNAMIC(YES) cannot become inactive. Binding packages with this option can therefore generate a large amount of active threads, and the maximum limit of 2000 concurrent threads can be reached easily. The benefits from the KEEP\_DYNAMIC(YES) option can be jeopardized by this limit.

More information about remote threads and tuning in a distributed environment can be found chapters 29 and 35 of *DB2 Universal Database for OS/309 and z/OS Administration Guide*, SC26-9931.

## 14.3 Minimizing message flows across the network

DB2 uses several techniques to avoid message flows across the network. Remote dynamic SQL can benefit from all of them, but we focus here only on elements specific or relevant to dynamic SQL. (For more details and explanations, see chapter 35 of *DB2 Universal Database for OS/309 and z/OS Administration Guide*, SC26-9931.)

- ▶ With the *block fetch* technique, DB2 avoids sending a message for every row retrieved in a query, by blocking them in the transmitted message buffer. This technique is used by DB2 when it can detect that the application contains no updates nor deletes. With dynamic SQL, DB2 cannot detect which statements are going to be executed by the program, and its decision to use block fetch is only based on the declaration of the cursor; that is:
  - The cursor must be defined with the clause FOR FETCH ONLY or FOR READ ONLY.
  - The cursor is non-scrollable, and read-only cursor. The conditions to have a read-only cursor are given in Chapter 5 of *DB2 Universal Database for OS/390 and z/OS Version 7 SQL Reference*, SC26-9944. The condition about read-only view is not checked by DB2 and therefore does not apply to trigger block fetch.

- The cursor is a scrollable cursor declared as `INSENSITIVE` (such cursors are read-only).
- The cursor is a scrollable and read-only cursor declared as `SENSITIVE`, and the value of the bind option `CURRENTDATA` is `NO`.
- The cursor is ambiguous and the value of the bind option `CURRENTDATA` is `NO`.
- In certain circumstances during the execution of a remote query, DB2 can automatically do a close of a cursor, when it fetches the last row, or the *n*th row if `FETCH FIRST n ROWS ONLY` is specified. This implicit close can save network transmissions. The conditions are the following:
  - The query uses limited block fetch.
  - The query retrieves no LOBs.
  - The cursor is not a scrollable cursor.
  - One of the following conditions is true:
    - The cursor is not defined `WITH HOLD`.
    - The cursor is defined `WITH HOLD`, in a package (or plan) bound with the `KEEPDYNAMIC(YES)` option.

**Important:** DB2 can trigger fast implicit close of remote cursors defined `WITH HOLD` if the plan or package is bound with `KEEPDYNAMIC(YES)`. However, keep in mind that this bind option may consumes many active threads, as a thread which has touched a package bound with this option cannot become inactive.

- Use the `DEFER(PREPARE)` bind option, which improves performance of dynamic SQL on both the DRDA and the private protocol.  
 Using this option, the submission of the `PREPARE` statement is deferred until the related `EXECUTE`, `OPEN`, or `DESCRIBE` statement is executed, and network traffic can therefore be reduced. (Note however that `PREPARE` with the `INTO` clause is not deferred.)  
 This option has an impact on the programming. Errors that are normally issued by the `PREPARE`, are now delayed until the execution of the `EXECUTE`, `DESCRIBE`, or `OPEN` statement and the application program must be able to handle them at that time.  
`NODEFER(PREPARE)` and `REOPT(VARS)` are incompatible options. When `REOPT(VARS)` is specified, DB2 automatically use `DEFER(PREPARE)`.  
 To really benefit from `DEFER(PREPARE)` in DRDA, the current `PACKAGESET` must be the collection of the package containing the statement which prepare must be deferred, or the first entry in the package list must refer to this package. If this is not the case, you loose the benefit of reduced message traffic. Every time DB2 has to look for a package in a different collection on a remote server causes the flow of a network message. Therefore, if the `PKLIST` contains 5 collections and the package you need is in the last collection, it takes 5 network trips to find it, hereby eliminating the saving of one message by using `DEFER(PREPARE)`.
- With the DRDA protocol, when the requester must send a `PREPARE` statement for a query, it anticipates a cursor, and sends a `PREPARE`, `DESCRIBE`, and `OPEN` in the same message. The DB2 server then sends a single output message for the three requests. This behavior is valid even if the package is bound with the `NODEFER(PREPARE)` option. The `OPEN` is retained in DDF storage until the program issues it. When the `OPEN` is issued, DDF processes it. If the `OPEN` is not issued, the following happens:
  - If the program issues a `PREPARE`, a `CLOSE` statement is chained before the `PREPARE`.
  - If the program issues a `COMMIT`, the cursor is closed, or the reply is retained in DDF if the query has a `WITH HOLD` option.
  - If the program aborts, the cursor is closed during the abort processing.

When using ODBC applications on NT or UNIX, you can also benefit from the following keywords and attributes:

- ▶ **DEFERREDPREPARE.** When this keyword is on, which is the default value, the DB2 driver defers the PREPARE call until the moment it really needs it (until the execution or a description of parameters is asked).  
This option provides the same effect as DEFER(PREPARE) on z/OS applications.
- ▶ **EARLYCLOSE.** This keyword can be set to ask DB2 Connect to automatically send close requests when it receives the last record of the cursor or the *n*th record when FETCH *n* ROWS ONLY is used. Note however that DB2 on z/OS itself does implicit fast closes under specific circumstances, and informs DB2 Connect of the close.
- ▶ **SQL\_ATTR\_CLOSEOPEN.** When this statement attribute is set to 1, the driver automatically closes a open cursor if a second cursor is opened on the same handle. This option can save network flows, but the application program must be written to avoid unnecessary closes.

## 14.4 Miscellaneous considerations

Here we describe several other considerations about remote dynamic SQL:

- ▶ Using the DB2 private protocol, DB2 always does a dynamic PREPARE at execution time, and it is therefore not recommended to use the option REOPTS(VAR), which sometimes generates the PREPARE statement twice.
- ▶ For remote applications, DB2 Connect may be required to implement the DRDA requester function. Some tools integrate the DRDA requester function, as for example QMF for Windows or Datajoiner.
- ▶ Governing remote dynamic requests entering a DB2 is done by the local site limits, but the following must be observed:
  - Governing applications using DRDA is done by package name. The PLANNAME column of the RLST table must be blank.
  - Governing applications using the private protocol is done by plan name. The RLFCOLLN and RLFPKG columns of the RLST table must be blank.
  - Governing applications using TCP/IP by LUNAME is not possible. The value of the LUNAME column in the RLST table must be PUBLIC.
  - If there is no row in the RLST table for reactive governing for requests coming from a remote location, or if the RLST table is not available, DB2 uses the RLFERRD parameter to decide which action to take:
    - If is NOLIMIT, the SQL statement can run without limit.
    - If it is NORUN, the statement is terminated with an SQL error code.
    - If it has a value between 1 and 5000000, this value is used as the limit to determine when to terminate the statement.

Chapter 11, “Governing dynamic SQL” on page 167 gives detailed information about governing dynamic SQL.

- ▶ The only DYNAMICRULES options valid with the private protocol are the BIND and RUN behaviors. The other options are translated by DB2 at run time:
  - DEFINEBIND and INVOKEBIND are translated into BIND.
  - DEFINERUN and INVOKERUN are translated into RUN.

- ▶ Many tools or applications using remote SQL connect to DB2 with a general user ID, which can be a problem for accounting reporting. Section 12.8, “Identifying users to track resource usage” on page 218 explains how to bypass this difficulty.
- ▶ If DB2 receives a DESCRIBE statement on a static SQL statement from a remote requester supporting extended dynamic SQL (such as for example SQL/DS), it generates an error, since describing static SQL is not allowed. If you want DB2 to satisfy the DESCRIBE request, you must set the system parameter DESCSTAT to YES. In this case, a DESCRIBE SLQDA is generated at bind time and DB2 can execute the request at run time.





## Part 4

# Dynamic SQL in vendor packages

In this part of the book, we discuss how some of the more widely used enterprise solution packages use dynamic SQL.





## Dynamic SQL in ESP packages

In this chapter we show how Enterprise Solution Packages (ESPs) like PeopleSoft, SAP, and Siebel harness the features and overcome the shortcomings of dynamic SQL. It is not intended as a comprehensive overview of these ESP packages in general, nor as a tuning guide. We therefore limit the discussion to features that pertain to dynamic SQL.

## 15.1 Overview of product architectures

In this section, we provide a general overview of the 3-tiers architecture implemented by SAP, PeopleSoft, and Siebel when the database server is DB2 on z/OS or OS/390.

### 15.1.1 SAP architecture

As shown in Figure 15-1, the SAP application server runs a set of work processes (WP in the figure) which can each serve multiple users. To access data, the work process passes the SQL statement to the DBSL layer which first responsibility it to adapt the statement to the target DBMS. DBSL also maintains a cache where it keeps the text of the SQL statements sent to the DBMS (see 15.2, “Statement caching” on page 239).

The application server can be placed on the mainframe or on a UNIX or NT machine:

- ▶ When the application server is on the same mainframe, the work processes access the database through DB2 ODBC.
- ▶ When the application server is outside the mainframe, the work processes access DB2 through the *Integrated Call Level Interface* (ICLI) interface. ICLI is an optimized interface implementing only the subset of functions needed by SAP, and sending/receiving only the data needed by SAP.

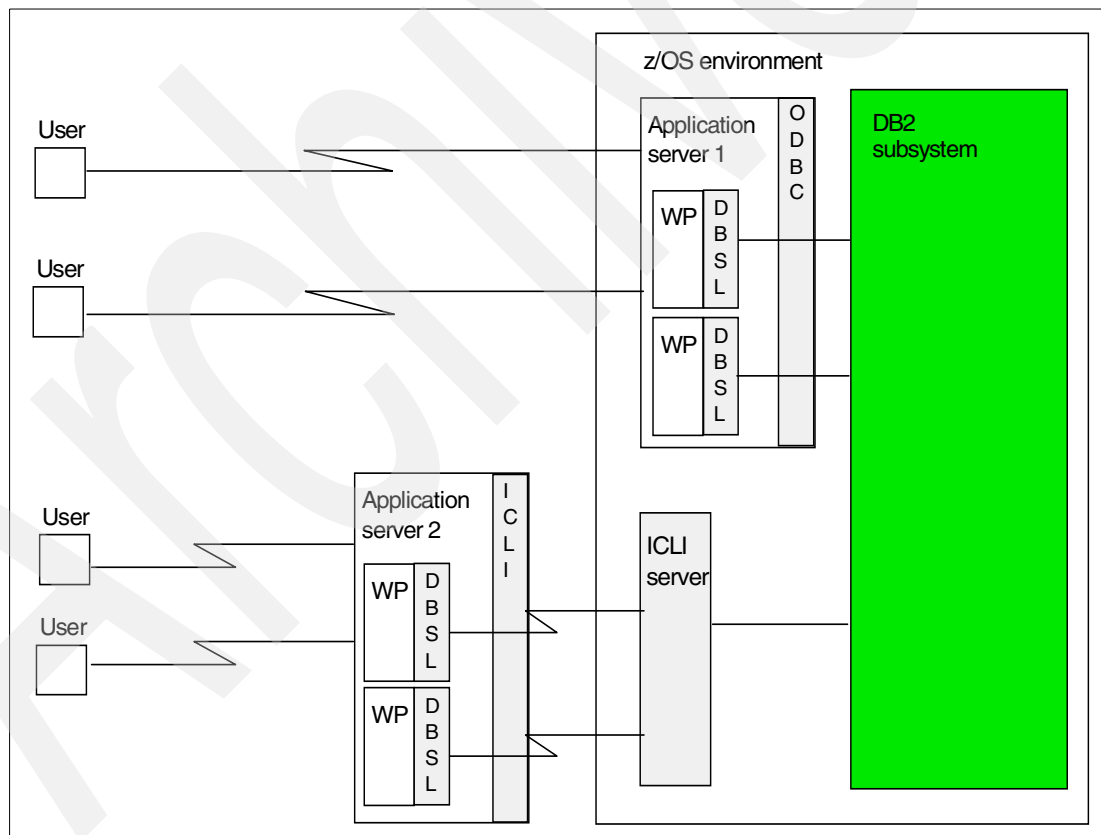


Figure 15-1 SAP architecture overview

### 15.1.2 PeopleSoft and Siebel architectures

As shown in Figure 15-2, both PeopleSoft and Siebel use the ODBC interface and DB2 Connect to access DB2 on the mainframe.

The application server can be on NT or UNIX, but not on the mainframe.

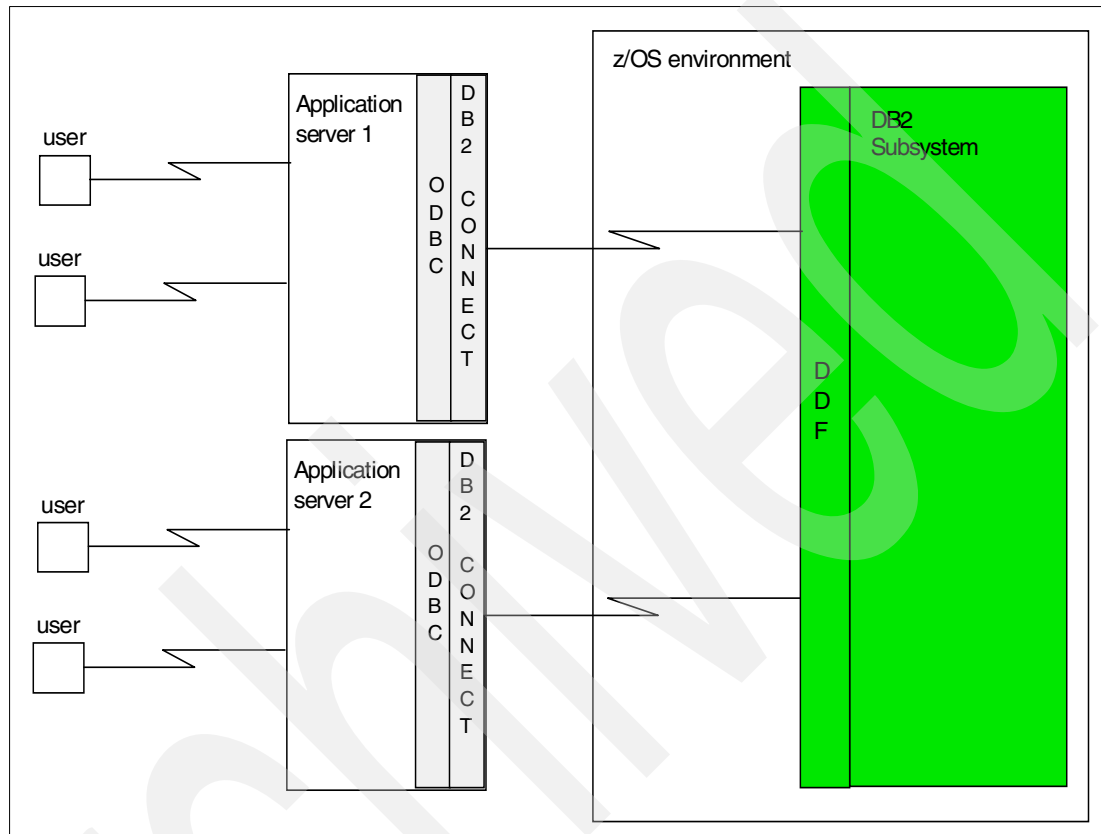


Figure 15-2 PeopleSoft and Siebel architecture overview

## 15.2 Statement caching

The ability to exploit various forms of dynamic statement caching is the focal point for performance and choices for various options discussed in the next few sections. They are geared towards attaining a good level of reuse from caching. The details of statement caching are discussed in Chapter 8, “Dynamic statement caching” on page 141. In this section we discuss how PeopleSoft, SAP and Siebel exploit it.

There are three forms of caching:

- ▶ Global cache
- ▶ Local cache
- ▶ Application server cache

Let us discuss each form in detail, and whether and how each package implements it.

### 15.2.1 Global cache

The global dynamic statement cache is allocated in the EDM pool within the DBM1 address space and is activated by setting CACHEDYN DSNZPARM to YES. This option applies to all applications using dynamic SQL and the hit ratio is based on the size of the EDMPOOL, the number of applications using dynamic SQL, and the pattern in which these applications reuse statements from the cache, since a least recently used (LRU) algorithm is used to remove the prepared statements from the cache. An application using the global cache does not need any special bind parameters or any special logic.

The hit ratio is likely to be higher for those applications:

- ▶ Using parameter markers (see 15.3, “Use of parameter markers” on page 240)
- ▶ Not using the REOPT(VARS) bind option (see 15.4, “Use of REOPT(VARS)” on page 241)
- ▶ Using a common authorization ID (see 15.5, “Authorization ID and security” on page 241).

All three packages recommend using global caching and their general design encourages a good hit ratio. The specific steps taken to ensure this are discussed in the sections that follow.

### 15.2.2 Local cache

A local dynamic statement cache is allocated in the storage of each thread in the DBM1 address space and activated by the bind option KEEP\_DYNAMIC(YES). In addition, the DSNZPARM MAXKEEPD must be set to a non-zero value. When local caching is active, an application can issue a PREPARE for a statement once and does not need to issue subsequent PREPAREs for this statement, even after a commit.

To really benefit from the use of KEEP\_DYNAMIC (YES), you should also turn on the global cache. This is also called full caching. In this case, the full executable statement is kept in the thread's local storage (not just the statement string) across commits for reuse.

No benefit is obtained from the local cache unless the application incorporates logic to avoid a subsequent PREPARE. This can be done either by keeping track of prepared statements within the application or by using an application server cache (see below). SAP employs an application server cache. Currently, PeopleSoft and Siebel do not use such a cache nor have any logic for prepare avoidance. For this reason, they will not benefit from the local cache.

### 15.2.3 Application server cache

This refers to a cache within the application server that connects to the DB2 server. Only SAP implements this cache (managed by the DBSL layer) and keeps track of up to 300 statements per thread to avoid prepares.

## 15.3 Use of parameter markers

We discussed parameter markers in 3.4, “What are parameter markers?” on page 22. The following examples provide a quick recap of why parameter markers can be used. Since dynamic SQL statements cannot contain host variables, the only permissible method to obtain the same effect is to use parameter markers (designated with a “?”) as part of the text that is PREPARED. When executing the prepared statement (or opening the associated cursor for SELECT statements), the corresponding host variable is associated with a parameter marker.

As the following example shows, a statement can be prepared for each LASTNAME and SALARY combination requested by the user:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE 'S%' AND SALARY > 50000
```

This requires that the statement be prepared each time. Alternatively, we can prepare the statement once containing parameter markers as follows:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE? AND SALARY >?
```

Then we can supply values for the parameter markers as needed.

What impact does this have on dynamic statement caching? Recall that one of the conditions that must be met before a cached statement is reused, is that the statement text must match exactly. Consider the following, without the use of parameter markers:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE 'S%' AND SALARY > 50000
```

This string will not match a similar statement issued later:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE 'P%' AND SALARY > 60000
```

Thus it will eliminate the benefits of caching.

In view of the discussion above, use of parameter markers encourages reuse of the statements via dynamic statement caching and is therefore widely utilized in all three packages.

## 15.4 Use of REOPT(VARS)

We discussed the bind option REOPT(VARS) in 7.1, “REOPT(VARS) option” on page 132 and its impact on dynamic statement caching. Since we cannot use REOPT(VARS) and still obtain benefits from dynamic statement caching, this is generally not used by most ERP programs. The only exceptions are those programs likely to benefit due to the better access path resulting from non-uniform distribution of data - *even if no dynamic statement caching is obtained.*

## 15.5 Authorization ID and security

Recall from Chapter 8, “Dynamic statement caching” on page 141 the conditions that must be met before a cached statement is used. One of the conditions is that the authorization ID must match. For example, a statement issued by user A may be cached:

```
SELECT      LASTNAME, FIRSTNAME, SALARY
FROM        DSN8710.EMP
WHERE       LASTNAME LIKE? AND SALARY >?
```

If user B issues the same statement, the cache cannot be used. Clearly, it is better to use a common authorization ID for both users to encourage dynamic statement caching and that is precisely what the three packages implement.

The details of how this is implemented in each package vary to some extent. Essentially, the security is controlled in the application and once authenticated, a “common SQL ID” used to issue all dynamic SQL statements. The combination of the primary authorization ID and the current SQLID determine whether or not dynamic statement caching is beneficial. This is discussed further in section Section 8.2.4, “Full caching” on page 147.

► PeopleSoft:

The application server first verifies that the user is authorized for the function requested and manages the connect to DB2 z/OS using a common authorization ID and a common SQLID. This SQLID is typically the creator of the production tables. As a result of this, the user does not need to have access to the tables outside of the system and cannot use the common SQLID to access and/or manipulate the data outside of the application. This leads to a very secure environment.

► SAP:

SAP uses a scheme similar to the one described above for PeopleSoft: the application server (or the ICLI server) uses one unique authorization ID and one unique SQLID (the creator of the application tables) for all users. Note, however, that different application servers (or different ICLI servers) can connect with different authorization ID.

When the application server is not on the mainframe, the security exposure is essentially eliminated since SAP uses ICLI, (a private interface different from ODBC), and a user's ability to access the data outside of the application is severely limited to begin with.

► Siebel:

When you use native TSO/RACF security, Siebel uses a scheme similar to the one used by PeopleSoft to verify that the user is authorized for the requested function. From this point on, the primary authorization ID of the user and a common SQLID (typically the creator of the tables) are used. This means that the common SQLID must be part of a user's secondary authorization ID list and the application issues a SET CURRENT SQLID to that common SQLID. This leads to a user having access to the same functions outside of the application. To limit the potential exposure, it is recommended to restrict Siebel users from accessing TSO.

When using the Siebel External Security Adapter, you maintain user information in an external authentication system, such as an LDAP repository. When users log on to the Siebel application, the External Security Adapter validates the user name and password against the information in the external system. If the External Security Adapter finds a match, it retrieves a generic set of user credentials (username/password) that are used to gain access to the actual database. Note that these can be the same credentials for every user. Using the Siebel External Security Adapter, you can avoid the potential problem of people trying to access the database outside the Siebel application.



## Appendixes

In this part of the book, we provide the following appendixes:

- ▶ Appendix A, “Comparing plan accounting, package accounting, and dynamic statement cache statistics” on page 245
- ▶ Appendix B, “Additional material” on page 251

Archived



## **Comparing plan accounting, package accounting, and dynamic statement cache statistics**

In this appendix we compare the information that can be obtained at the plan, package, and statement level (provided we are dealing with a dynamic SQL statement that is in the dynamic statement cache).

# Comparison of plan, package, and cache information

In Example A-1 we show the information that is available at the plan (accounting), package (accounting) and dynamic statement cache (via IFI READS) level.

Example: A-1 Comparing plan, package and cache info

Plan accounting				Package accounting			Statement cache		
TIMES/EVENTS	APPL(CL.1)	DB2 (CL.2)	IFI (CL.5)	DSNTEP2	TIMES				
ELAPSED TIME	19.068353	18.998459	N/P	ELAPSED TIME - CL7	18.998432		Elapsed time		
NONNESTED	19.068353	18.998459	N/A				Elapsed time/exec		
STORED PROC	0.000000	0.000000	N/A						
UDF	0.000000	0.000000	N/A						
TRIGGER	0.000000	0.000000	N/A						
CPU TIME	0.017664	0.006701	N/P	CPU TIME	0.006676		Accumulated CPU time		
AGENT	0.017664	0.006701	N/A	AGENT	0.006676				
NONNESTED	0.017664	0.006701	N/P						
STORED PRC	0.000000	0.000000	N/A						
UDF	0.000000	0.000000	N/A						
TRIGGER	0.000000	0.000000	N/A						
PAR.TASKS	0.000000	0.000000	N/A	PAR.TASKS	0.000000				
SUSPEND TIME	N/A	18.988852	N/A	SUSPENSION-CL8	18.988852				
AGENT	N/A	18.988852	N/A	AGENT	18.988852				
PAR.TASKS	N/A	0.000000	N/A	PAR.TASKS	0.000000				
NOT ACCOUNT.	N/A	0.002906	N/P	NOT ACCOUNTED	0.002905				
DB2 ENT/EXIT	N/A	137	N/A	DB2 ENTRY/EXIT	136				
EN/EX-STPROC	N/A	0	N/A						
EN/EX-UDF	N/A	0	N/A						
DCAPT.DESCR.	N/A	N/A	N/P						
LOG EXTRACT.	N/A	N/A	N/P						
SERVICE UNITS	CLASS 1	CLASS 2		CPU SERVICE UNITS	76				
CPU	203	77							
AGENT	203	77		AGENT	76				
NONNESTED	203	77							
STORED PROC	0	0							
UDF	0	0							
TRIGGER	0	0							
PAR.TASKS	0	0		PAR.TASKS	0				
CLASS 3 SUSPENSIONS	ELAPSED TIME	EVENTS		PACKAGE	TIME	EVENTS	STATEMENT	TIME	EVENTS
LOCK/LATCH(DB2+IRLM)	18.980182	2		LOCK/LATCH	18.980182	2	Lock/latch	X	N/A
SYNCHRON. I/O	0.000000	0		SYNCHRONOUS I/O	0.000000	0	Synch I/O	X	N/A
DATABASE I/O	0.000000	0							
LOG WRITE I/O	0.000000	0							
OTHER READ I/O	0.000000	0		OTHER READ I/O	0.000000	0	Read by other	X	X
OTHER WRTE I/O	0.000000	0		OTHER WRITE I/O	0.000000	0	Write by other	X	X
SER.TASK SWITCH	0.008669	1		SERV.TASK SWITCH	0.008669	1	Synch EU switch	X	N/A
UPDATE COMMIT	0.008669	1							
OPEN/CLOSE	0.000000	0							
SYSLGRNG REC	0.000000	0							
EXT/DEL/DEF	0.000000	0							
OTHER SERVICE	0.000000	0							
ARC.LOG(QUIES)	0.000000	0		ARCH.LOG(QUIESCE)	0.000000	0			
ARC.LOG READ	0.000000	0		ARCHIVE LOG READ	0.000000	0			
STOR.PRC SCHED	0.000000	0		STORED PROC SCHED	0.000000	0			
UDF SCHEDULE	0.000000	0		UDF SCHEDULE	0.000000	0			
DRAIN LOCK	0.000000	0		DRAIN LOCK	0.000000	0			
CLAIM RELEASE	0.000000	0		CLAIM RELEASE	0.000000	0			
PAGE LATCH	0.000000	0		PAGE LATCH	0.000000	0			
NOTIFY MSGS	0.000000	0		NOTIFY MESSAGES	0.000000	0			
GLOBAL CONTENTION	0.000000	0		GLOBAL CONTENTION	0.000000	0	Global locks	X	N/A
COMMIT PH1 WRITE I/O	0.000000	0							
ASYNCH IXL REQUESTS	0.000000	0							
TOTAL CLASS 3	18.988852	3		TOTAL CL8 SUSPENS.	18.988852	3			

Detailing GLOBAL CONTENTION counter

GLOBAL CONTENTION L-LOCKS ELAPSED TIME EVENTS

L-LOCKS	0.000000	0
PARENT (DB,TS,TAB,PART)	0.000000	0
CHILD (PAGE,ROW)	0.000000	0
OTHER	0.000000	0

GLOBAL CONTENTION	P-LOCKS	ELAPSED TIME	EVENTS
P-LOCKS		0.000000	0
PAGESET/PARTITION		0.000000	0
PAGE		0.000000	0
OTHER		0.000000	0

SQL DML	TOTAL
---------	-------

SELECT	0
INSERT	1
UPDATE	0
DELETE	0

DESCRIBE	4
DESC.TBL	0
PREPARE	4
OPEN	2
FETCH	53
CLOSE	2

DML-ALL	66
---------	----

SQL DCL	TOTAL
---------	-------

LOCK TABLE	0
GRANT	0
REVOKE	0
SET SQLID	0
SET H.VAR.	0
SET DEGREE	0
SET RULES	0
SET PATH	0
SET PREC.	0
CONNECT 1	0
CONNECT 2	0
SET CONNEC	0
RELEASE	0
CALL	0
ASSOC LOC.	0
ALLOC CUR.	0
HOLD LOC.	0
FREE LOC.	0
DCL-ALL	0

SQL DDL	CREATE	DROP	ALTER
---------	--------	------	-------

TABLE	0	0	0
CRT TTABLE	0	N/A	N/A
DCL TTABLE	0	N/A	N/A
AUX TABLE	0	N/A	N/A
INDEX	0	0	0
TABLESPACE	0	0	0
DATABASE	0	0	0
STOGROUP	0	0	0
SYNONYM	0	0	N/A
VIEW	0	0	N/A
ALIAS	0	0	N/A
PACKAGE	N/A	0	N/A
PROCEDURE	0	0	0
FUNCTION	0	0	0
TRIGGER	0	0	N/A
DIST TYPE	0	0	N/A

TOTAL	0	0	0
RENAME TBL	0		
COMMENT ON	0		
LABEL ON	0		

LOCKING	TOTAL
---------	-------

TIMEOUTS	0
DEADLOCKS	0
ESCAL.(SHAR)	0

SQL STATEMENTS 67

ESCAL.(EXCL)	0
MAX PG/ROW LCK HELD	2
LOCK REQUEST	7
UNLOCK REQST	4
QUERY REQST	0
CHANGE REQST	1
OTHER REQST	0
LOCK SUSPENS.	2
IRLM LATCH SUSPENS.	0
OTHER SUSPENS.	0
TOTAL SUSPENS.	2

DATA SHARING	TOTAL
-----	-----
GLB CONT (%)	N/P
FLS CONT (%)	N/P
L-LOCKS (%)	N/P
LOCK REQUEST	N/P
UNLOCK REQST	N/P
CHANGE REQST	N/P
LOCK - XES	N/P
UNLOCK-XES	N/P
CHANGE-XES	N/P
SUSP - IRLM	N/P
SUSP - XES	N/P
SUSP - FALSE	N/P
INCOMP.LOCK	N/P
NOTIFY SENT	N/P

DRAIN/CLAIM	TOTAL
-----	-----
DRAIN REQST	0
DRAIN FAILED	0
CLAIM REQST	10
CLAIM FAILED	0

RID LIST	TOTAL
-----	-----
USED	0
FAIL-NO STORAGE	0
FAIL-LIMIT EXC.	0

Storage limit exceeded  
DB2 limit exceeded

ROWID	TOTAL
-----	-----
DIR ACCESS	0
INDEX USED	0
TS SCAN	0

STORED PROC.	TOTAL
-----	-----
CALL STMTS	0
ABENDED	0
TIMED OUT	0
REJECTED	0

UDF	TOTAL
-----	-----
EXECUTED	0
ABENDED	0
TIMED OUT	0
REJECTED	0

TRIGGERS	TOTAL
-----	-----
STMT TRIGGER	0
ROW TRIGGER	0
SQL ERROR	0

QUERY PARALLEL.	TOTAL
-----	-----
MAXIMUM MEMBERS	N/P
MAXIMUM DEGREE	0
GROUPS EXECUTED	0
RAN AS PLANNED	0
RAN REDUCED	0
ONE DB2 COOR=N	0
ONE DB2 ISOLAT	0
ONE DB2 DCL TTABLE	0

Parallel groups

SEQ - CURSOR	0
SEQ - NO ESA	0
SEQ - NO BUF	0
SEQ - ENCL.SER	0
MEMB SKIPPED(%)	0
DISABLED BY RLF	NO
REFORM PARAL-CONFIG	0
REFORM PARAL-NO BUF	0

DATA CAPTURE	TOTAL
IFI CALLS	N/P
REC.CAPTURED	N/P
LOG REC.READ	N/P
ROWS RETURN	N/P
RECORDS RET.	N/P
DATA DES.RET	N/P
TABLES RET.	N/P
DESCRIBES	N/P

DYNAMIC SQL STMT	TOTAL
REOPTIMIZATION	0
NOT FOUND IN CACHE	0
FOUND IN CACHE	3
IMPLICIT PREPARES	0
PREPARES AVOIDED	0
STMT INVALID (MAX)	0
STMT INVALID (DDL)	0

LOGGING	TOTAL
LOG RECS WRITTEN	12
TOT BYTES WRITTEN	981

MISCELLANEOUS	TOTAL
MAX STOR VALUES	0

BPO	TOTAL
BPOOL HIT RATIO (%)	100
HPOOL HIT RATIO (%)	0
GETPAGES	66
GETPAGES-FAILED	0
BUFFER UPDATES	8
SYNCHRONOUS WRITE	0
SYNCHRONOUS READ	0
SEQ. PREFETCH REQS	0
LIST PREFETCH REQS	0
DYN. PREFETCH REQS	0
PAGES READ ASYNCHR.	0
HPOOL WRITES	0
HPOOL WRITES-FAILED	0
PAGES READ ASYN-HPOOL	0
HPOOL READS	0
HPOOL READS-FAILED	0

Getpage

HIGHLIGHTS
THREAD TYPE : ALLIED
TERM.CONDITION: NORMAL
INVOKE REASON : DEALLOC
COMMITTS : 0
ROLLBACK : 1
SVPT REQUESTS : 0
SVPT RELEASE : 0
SVPT ROLLBACK : 0
INCREM.BINDS : 0
UPDATE/COMMIT : 1.00
SYNCH I/O AVG.: N/C
PROGRAMS : 1
MAX CASCADE : 0
PARALLELISM : NO

Table space scan  
Index scan

Rows exam/exec  
 Rows exam  
 Rows proc/exec  
 Rows proc  
  
 Sort  
  
 SQL Statement  
 Users  
 Copies  
 Status  
 Executions  
 Insert time

Plan accounting information (available when accounting trace classes 1,2,3 are active) is a superset of the information that is available at the package accounting level (available through accounting trace classes 7,8).

At the statement level, assuming the statement we are dealing with is a dynamic SQL statement and is in the dynamic statement cache, we have access to information that was previously only available using expensive performance traces. When trace classes that contain IFCID 316,318 are active, the number of scan and sort operations that are needed to execute the statement, as well as the number of rows examined and processed are available at the SQL statement level.

Table A-1 summarizes the differences between the different types of information that are available in the three types of trace records.

*Table A-1 Summary of trace record information*

	Plan accounting	Package accounting	Statement cache
Elapsed / CPU / Suspension times	Full information with: -Stored procedure -UDF - Trigger - Parallel tasks time reported separately	Only agent and parallel tasks are reported separately	Only total elapsed and CPU time for all executions and the elapsed time per execution is available
Suspensions	26 counters	14 counters	6 counters
SQL executed	Full details by type of SQL statement	Only total number of SQL statements	N/A
Locking counters	Local locking, data sharing locking, global P/L-lock contention, drain/claim info	No lock info	No lock info
RID list info	Yes	No	Only failures
Stored procedures / UDF / Triggers	Yes	No	No
Query parallelism	Yes	No	Total number of groups
Dynamic SQL statement (cache)	Yes	No	Number of users, copies, executions of the statement
Buffer pool info	Yes	No	Number of getpages
Scan info	No	No	- # of TS scans - # of IX scans - # of rows examined - # of rows processed - # of sorts



## Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

### Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246418>

Alternatively, you can go to the IBM Redbooks Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246418.

### Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
<b>dscobol1.zip</b>	Zipped sample COBOL program with embedded dynamic SQL
<b>dscobolp.zip</b>	Zipped sample COBOL subroutine for parameter SQLDA
<b>dscobols.zip</b>	Zipped sample COBOL subroutine for receiving SQLDA
<b>dsplipg1.zip</b>	Zipped sample PL/I program with embedded dynamic SQL
<b>dynrex1.zip</b>	Zipped sample REXX exec with dynamic SQL
<b>dsodbc.zip</b>	Zipped sample C program using ODBC (9 variations provided)
<b>dsjdbc.zip</b>	Zipped sample Java program using JDBC
<b>dsjdbcthread.zip</b>	Zipped sample Java program running 2 threads
<b>SG246418.zip</b>	All samples zipped in a single file

Archived

# Abbreviations and acronyms

<b>AIX</b>	Advanced Interactive eXecutive from IBM	<b>EBCDIC</b>	extended binary coded decimal interchange code
<b>APAR</b>	authorized program analysis report	<b>ECS</b>	enhanced catalog sharing
<b>ARM</b>	automatic restart manager	<b>ECSA</b>	extended common storage area
<b>ASCII</b>	American National Standard Code for Information Interchange	<b>EDM</b>	environment descriptor management
<b>BLOB</b>	binary large objects	<b>ERP</b>	enterprise resource planning
<b>CCSID</b>	coded character set identifier	<b>ESA</b>	Enterprise Systems Architecture
<b>CCA</b>	client configuration assistant	<b>ESP</b>	Enterprise Solution Package
<b>CFCC</b>	coupling facility control code	<b>ETR</b>	external throughput rate, an elapsed time measure, focuses on system capacity
<b>CTT</b>	created temporary table	<b>FTD</b>	functional track directory
<b>CEC</b>	central electronics complex	<b>FTP</b>	File Transfer Program
<b>CD</b>	compact disk	<b>GB</b>	gigabyte (1,073,741,824 bytes)
<b>CF</b>	coupling facility	<b>GBP</b>	group buffer pool
<b>CFRM</b>	coupling facility resource management	<b>GRS</b>	global resource serialization
<b>CLI</b>	call level interface	<b>GUI</b>	graphical user interface
<b>CLP</b>	command line processor	<b>HPJ</b>	high performance Java
<b>CPU</b>	central processing unit	<b>IBM</b>	International Business Machines Corporation
<b>CSA</b>	common storage area	<b>ICF</b>	integrated catalog facility
<b>DASD</b>	direct access storage device	<b>ICF</b>	integrated coupling facility
<b>DB2 PM</b>	DB2 performance monitor	<b>ICMF</b>	internal coupling migration facility
<b>DBAT</b>	database access thread	<b>IFCID</b>	instrumentation facility component identifier
<b>DBD</b>	database descriptor	<b>IFI</b>	instrumentation facility interface
<b>DBID</b>	database identifier	<b>IPLA</b>	IBM Program Licence Agreement
<b>DBRM</b>	database request module	<b>IRLM</b>	internal resource lock manager
<b>DSC</b>	dynamic statement cache, local or global	<b>ISPF</b>	interactive system productivity facility
<b>DCL</b>	data control language	<b>ISV</b>	independent software vendor
<b>DDCS</b>	distributed database connection services	<b>I/O</b>	input/output
<b>DDF</b>	distributed data facility	<b>ITR</b>	internal throughput rate, a processor time measure, focuses on processor capacity
<b>DDL</b>	data definition language	<b>ITSO</b>	International Technical Support Organization
<b>DLL</b>	dynamic load library manipulation language	<b>IVP</b>	installation verification process
<b>DML</b>	data manipulation language	<b>JDBC</b>	Java Database Connectivity
<b>DNS</b>	domain name server	<b>JFS</b>	journaled file systems
<b>DRDA</b>	distributed relational database architecture	<b>JNDI</b>	Java Naming and Directory Interface
<b>DTT</b>	declared temporary tables		
<b>EA</b>	extended addressability		

<b>JVM</b>	Java Virtual Machine
<b>KB</b>	kilobyte (1,024 bytes)
<b>LOB</b>	large object
<b>LPL</b>	logical page list
<b>LPAR</b>	logical partition
<b>LRECL</b>	logical record length
<b>LRSN</b>	log record sequence number
<b>LUW</b>	logical unit of work
<b>LVM</b>	logical volume manager
<b>MB</b>	megabyte (1,048,576 bytes)
<b>NPI</b>	non-partitioning index
<b>ODB</b>	object descriptor in DBD
<b>ODBC</b>	Open Data Base Connectivity
<b>OS/390</b>	Operating System/390
<b>PAV</b>	parallel access volume
<b>PDS</b>	partitioned data set
<b>PIB</b>	parallel index build
<b>PSID</b>	pageset identifier
<b>PSP</b>	preventive service planning
<b>PTF</b>	program temporary fix
<b>PUNC</b>	possibly uncommitted
<b>QMF</b>	Query Management Facility
<b>QA</b>	Quality Assurance
<b>RACF</b>	Resource Access Control Facility
<b>RBA</b>	relative byte address
<b>RECFM</b>	record format
<b>RID</b>	record identifier
<b>RRS</b>	resource recovery services
<b>RRSAF</b>	resource recovery services attach facility
<b>RS</b>	read stability
<b>RR</b>	repeatable read
<b>SDK</b>	software developers kit
<b>SMIT</b>	System Management Interface Tool
<b>SU</b>	Service Unit
<b>UOW</b>	unit of work

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## IBM Redbooks and Redpapers

For information on ordering these publications, see “How to get IBM Redbooks” on page 257.

- ▶ *DB2 for z/OS and OS/390 Version 7 Selected Performance topics*, REDP0162
- ▶ *DB2 for z/OS and OS/390 Tools for Performance Management*, SG24-6508
- ▶ *DB2 for z/OS Application Programming Topics*, SG24-6300
- ▶ *DB2 for z/OS and OS/390 Version 7: Using the Utilities Suite*, SG24-6289
- ▶ *DB2 for OS/390 and z/OS Powering the World's e-business Solutions*, SG24-6257
- ▶ *DB2 for OS/390 and z/OS Performance topics*, SG24-6129
- ▶ *DB2 UDB Server for OS/390 and z/OS Version 7 Presentation Guide*, SG24-6121
- ▶ *DB2 UDB Server for OS/390 Version 6 Technical Update*, SG24-6108
- ▶ *DB2 Java Stored Procedures -- Learning by Example*, SG24-5945
- ▶ *DB2 UDB for OS/390 Version 6 Performance Topics*, SG24-5351
- ▶ *DB2 for OS/390 Version 5 Performance Topics*, SG24-2213
- ▶ *DB2 for MVS/ESA Version 4 Non-Data-Sharing Performance Topics*, SG24-4562
- ▶ *DB2 UDB for OS/390 Version 6 Management Tools Package*, SG24-5759
- ▶ *DB2 Server for OS/390 Version 5 Recent Enhancements - Reference Guide*, SG24-5421
- ▶ *DB2 for OS/390 Capacity Planning*, SG24-2244
- ▶ *Developing Cross-Platform DB2 Stored Procedures: SQL Procedures and the DB2 Stored procedure Builder*, SG24-5485
- ▶ *DB2 for OS/390 and Continuous Availability*, SG24-5486
- ▶ *Connecting WebSphere to DB2 UDB Server*, SG24-62119
- ▶ *Parallel Sysplex Configuration: Cookbook*, SG24-2076-00
- ▶ *DB2 for OS390 Application Design for High Performance*, GG24-2233
- ▶ *CCF Connectors and Database Connections Using WebSphere Advanced Edition*, SG24-5514

## Other resources

These publications are also relevant as further information sources:

- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 What's New*, GC26-9946
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Command Reference*, SC26-9934

- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Messages and Codes*, GC26-9940
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Utility Guide and Reference*, SC26-9945
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Programming Guide and Reference for Java*, SC26-9932
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Administration Guide*, SC26-9931
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Release Planning Guide*, SC26-9943
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 SQL Reference*, SC26-9944
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Text Extender Administration and Programming*, SC26-9948
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Data Sharing: Planning and Administration*, SC26-9935
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 Image, Audio, and Video Extenders*, SC26-9947
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 ODBC Guide and Reference*, SC26-9941
- ▶ *DB2 Universal Database for OS/390 and z/OS Version 7 XML Extender Administration and Reference*, SC26-9949
- ▶ *DB2 Universal Database Call Level Interface Guide and Reference*, SC09-2950
- ▶ *DB2 Universal Database Administrative API Reference*, SC09-2947
- ▶ *DB2 Connect Version 7 User's Guide*, SC09-2954
- ▶ *DB2 Performance Monitor for OS/390 Version 7 Report Reference*, SC27-0853
- ▶ *DB2 Performance Monitor for OS/390 Version 7 Batch User's Guide*, SC27-0857
- ▶ *Query Management Facility Version 7 Using QMF*, SC27-0716
- ▶ *z/OS V1R1.0 MVS Initialization and Tuning Guide*, SA22-7591

## Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ <http://www.ibm.com/software/data/db2/os390/> DB2 for OS/390
- ▶ <http://www.ibm.com/software/data/db2/os390/estimate/> DB2 Estimator
- ▶ <http://www.ibm.com/storage/hardsoft/diskdr1s/technology.htm> ESS
- ▶ <http://www.ibm.com/software/data/db2/os390/support.htm> DB2 support and services
- ▶ <http://www.ibm.com/software/data/db2/os390/odbc/> ODBC shadow catalog
- ▶ <http://www.ibm.com/software/data/db2/udb/staticcli/> ODBC static profiling
- ▶ <http://java.sun.com/products/jdbc/> JDBC specifications
- ▶ <http://www.ibm.com/software/data/db2/java/> DB2 Java Web site
- ▶ <http://www.ibm.com/software/data/db2/os390/jdbc.html> DB2 for z/OS JDBC info

- ▶ <http://www-3.ibm.com/software/data/db2/java/sqlj/> DB2 SQLJ page
- ▶ <http://www-3.ibm.com/software/data/db2/os390/sqlj.html> DB2 for z/OS SQLJ page
- ▶ <http://www.sqlj.org/> SQLJ home page

## How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

## IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Archived



# Index

## Symbols

#sql 9  
&SPRMCTH 155  
&SPRMSTH) 155

## A

Access path 199  
Access path hints 135  
Accessed table 212  
Accounting record 210  
ALIAS 10  
ALLOCATE CURSOR 118  
Allocating connection handle 65  
Applet 90  
ASUTIME 174  
Asynchronous SQL 54, 81  
ATTRIBUTES 149  
ATTRIBUTES clause 47  
Authorization ID 149  
Auto-commit 76  
Avoided prepare 148, 194

## B

BatchUpdateException object 100  
BEGIN DECLARE SECTION 80  
BIND  
    Options 150  
Bind 210  
Binding columns 72  
Binding parameters 67  
Bridge driver 92

## C

CACHEDYN 134  
CALL 21, 79  
Callable interface  
    JDBC 11  
    ODBC 11, 51  
CallableStatement 101–102, 105  
CallableStatement object 94, 105  
CAST 22  
Choices  
    Dynamic SQL and stored procedures 17  
    REOPT(VARS) and dynamic SQL 16  
Class.forName() 94, 100  
Classifying SQL 11  
    Combined execution mode 12  
    Dynamic execution 12  
    Static execution 12  
CLI 50  
CLISchema 85  
CLOSE 21, 79

COLLECTIONID 80, 223  
Column-wise binding 73  
COMMIT 76  
Compound SQL 81  
Concurrency 161  
CONNECT 21, 79  
Connection handle 53  
Connection pooling 81  
Connection.commit() 94  
Connection.createStatement() 94  
Connection.prepareStatement() 94  
Connection.rollback() 94  
CONNECTTYPE 77, 82–83  
CONTSTOR 155  
Coprocesor 9  
Cost category 171  
CREATE INDEX 150  
CTMSTAT 231  
CURRENT DEGREE 136, 149  
CURRENT OPTIMIZATION HINT 136, 149  
CURRENT PATH 149  
CURRENT PRECISION 149  
CURRENT RULES 149  
CURRENTDATA 149  
CURRENTPACKAGESET 80, 223  
CURSOR 21  
Cursor manipulation 21  
Cursor position 77  
Customize profile 9

## D

Data Collector task 204  
Data source 52, 89  
Data type conversion 54  
Database access thread 230  
DatabaseMetaData object 100  
DataSource 91  
DataSource.getConnection() 94  
DataTruncation object 99  
DB2 callable interface 10  
DB2 Instrumentation Facility 180  
DB2 Performance Monitor 180  
DB2 private protocol 230  
DB2 statistics 181  
DB2 trace facility 200  
DB2 trace types 180  
DB2CLI.INI 52  
DB2CURSORHOLD 98, 110  
db2prof 9  
DB2SQLJMULTICONTXT 110  
DB2SQLJPLANNAME 110  
DB2SQLJPROPERTIES 89  
DB2SQLJSSID 110  
DBD lock 162

- DBD size 158
- DBM1 154
- DBNAME 85
- DBPROTOCOL 230
- DBRM 9
- DECLARE 10
- DECLARE STATEMENT 80
- DECLARE TABLE 80
- Dedicated SQL 10
- DEFER(PREPARE) 132, 135, 232
- Deferred embedded dynamic SQL 10
- Deferred input arguments 68
- DEFERREDPREPARE 84
- DEFINEBIND 233
- DEFINERUN 233
- DEGREE 136
- Degree of parallelism 136
- DESCRIBE 43, 79
- DESCRIBE CURSOR 79
- DESCRIBE INPUT 79, 103
- DESCRIBE PROCEDURE 79
- Describing and retrieving results 98
- Describing parameters 67
- DESCSTAT 234
- Developing applications 1, 5
  - dft\_account\_str 219
- DISABLEMULTITHREAD 83
- Disconnecting handles 77
- Distinct types 43
- DML statement counter 184
- Double-prepare 132
- DRDA 230
- Driver 52, 89
- Driver manager 52, 81
- driver manager 89
- DriverManager 90, 100
- DriverManager class 100
- DriverPropertyInfo 91
- DSNAOINI 52
- DSNARLnn 169
- DSNHDECP 65
- DSNREXX 11, 116, 118
- DSNRLSTnn 168
- DSNTEP2 18
- DSNTIAD 18
- DSNTIAUL 18
- Dynamic SQL 184
  - Access path determination 16
  - Auditing 15
  - Choices 7, 13
  - Complexity 14
  - Developing embedded SQL applications 19
  - Flexibility 14
  - General recommendations 18
  - Governing 16
  - Importance of catalog statistics 17
  - Performance 15
  - Restrictions 20
  - Sample programs 18
  - Security 15

- The basics 21
- Dynamic statement caching 132, 141
  - Summary 151
- DYNAMICRULES 134, 149
- DYNAMICRULES(BIND) 222
- DYNAMICRULES(RUN) 222
- DYNRULS 159

## E

- EARLYCLOSE 84
- EDM pool 156, 190
- EDM pool window 190
- EDMBFIT 158
- EDMDSMAX 157
- EDMDSPAC 145
- EDMSPAC 157
- Embedded dynamic SQL 10, 222
- Embedded SQL 9
- Embedded static SQL 222
- END DECLARE SECTION 80
- END-EXEC 21
- Ending transactions 76
- Enterprise Solution Package 237
- Environment handle 53
- ESP 237
- EXEC SQL 9, 21
- EXECUTE 10, 21–22, 79
- Execute authority 222
- EXECUTE IMMEDIATE 22, 51, 79
- execute() 97
- executeQuery() 96, 103
- executeUpdate() 96, 103
- Executing statements 70, 96
- Execution statistics 195
- EXPLAIN 137
- Explain 198

## F

- FETCH 21, 79
  - USING DESCRIPTOR 39
- FETCH FIRST 10 ROWS ONLY 47
- FILE subcommand 208
- Fixed-list SELECT statements 25
  - With parameter markers 32
- FOR RESULT SET 118
- Freeing handles 77
- Full caching 147
- Full prepare 148, 193, 210, 213
- Function call 52
- Functions
  - Execute SQL 59
  - Manage results 59
  - Manipulate handle 55
  - Manipulate large object 62
  - Prepare execution 57
  - Query the catalog 62
  - Retrieve diagnostics 62
  - Retrieve driver information 62

## G

- Garbage collection 106
- General Data table 206
- Generalized Trace Facility (GTF) 200
- Generic search criteria 13
- getColumnType() 96
- getConnection() 101
- getErrorCode() 99
- getMessage() 99
- getMetaData() 100
- getMoreResults() 98
- getNextWarning() 99
- getParameterType() 95
- getXXX 103
- getXXX methods 106
- getXXX() 98
- Global Cache Hit Ratio 194
- Global caching 145
- Global statement caching
  - Data space 157
- Governor 168

## H

- Handle attribute 54
- Host variables 9

## I

- IFCID 180, 200
  - 22 199, 207
  - 316 195–196
  - 317 195
  - 318 195–196
  - 63 207
- Implicit prepare 144, 148, 194
- INCLUDE 80
- Initializing database connection 100
- Initializing driver 100
- INSENSITIVE 48
- Integrated Call Level Interface 238
- Interactive dynamic SQL 10
- INVOKEBIND 233
- INVOKERUN 233
- ISO
  - CLI 50
- ISOLATION 149
- Isolation level 129

## J

- Java 88
  - Dynamic SQL 11
  - Static SQL 9
- Java Server Page 90
- JavaBeans 89
- JDBC 9, 11, 87
  - Samples 100
- JDBC classes 92
- JDBC connection 90
- JDBC driver types 92

- JDBC security 223

## K

- KEEPDYNAMIC 12, 85, 133, 142, 144

## L

- LOBs 43
- Local Cache Hit Ratio 194
- Local cache storage 154
- Local caching 143
- Lock 161
- Lock avoidance 163

## M

- Managing connection objects 94
- Managing execution results 71, 97
- Managing statement objects 94
- Manipulate handle 55
- Manipulating statement handles 66
- Manual-commit 76
- Mapping 78
- MAXKEEPD 147, 154
- Miniplan 210
- Miniplan record 210
- Miniplan table 207
- MINSTOR 155
- Mixing
  - SELECT and non-SELECT 48
- MULTICONTTEXT 77, 82
- MVSDEFAULTSSID 52, 65

## N

- Native API driver 92
- Native java driver 92
- Net driver 92
- next() 104
- No caching 142
- NODEFER(PREPARE) 232
- Non-SELECT statements 24
- Non-uniform distribution 15
- NOREOPT(VARS) 132
- NumParams 79

## O

- OBBC
  - Automatic cursor close 86
- ODBC 11, 49, 88
  - Dynamic statement caching 85
  - Improving catalog access 85
  - Installation 80
  - Performance considerations 84
  - Programming hints 84
  - Stored procedures 86
  - Using DB2 on z/OS features 85
- ODBC security 223
- OPEN 21, 79
- Open Group 11, 50, 88

OPTHINT 135  
OPTIMIZE FOR n ROWS 47

## P

Package 212  
Parameter markers 22, 240  
    Handling unknown number and type 45  
    Typed 22  
    Untyped 22  
ParameterMetaData 103  
PeopleSoft 239, 242  
Performance 180  
    Problem 213  
Performance database 180, 199, 206  
Persistence of dynamic SQL 12  
PLANNAME 80  
Positioned delete 75, 98  
Positioned update 75, 98  
PQ24634 162  
PQ25914 158  
PQ31969 158  
PQ37894 155  
PQ51847 113  
PQ57168 181  
Predictive governor 133, 170, 172  
    Example 173  
Predictive-reactive governor combined 175  
PREPARE 10, 21–22, 79  
PREPARE INTO 35  
PreparedStatement 101–102  
PreparedStatement object 94, 105  
PreparedStatement.GetMetaData() 96  
PreparedStatement.getParameterMetaData() 95  
Preparing statements 66  
Private protocol 10  
Processor service units 169  
Profile 9, 111  
Programming interfaces 8  
    Dynamic 8  
    Embedded SQL 9  
    Static 8  
Properties file 89

## Q

QISED SI 186  
QLGetSQLCA() 78  
QMF 10, 15  
QUALIFIER 149  
QWHCEUID 218  
QWHCEUTX 218  
QWHCEUWN 218  
QXPREP 185, 187, 193  
QXSTDEXP 186, 194  
QXSTDINV 186, 194  
QXSTFND 186  
QXSTIPRP 186, 193  
QXSTIPTP 194  
QXSTNFND 186  
QXSTNPRP 186, 194

## R

Reactive governor 174  
    Example 175  
Read-only cursor 162  
Receiving-SQLDA 23, 41  
Record Trace group 207  
Redbooks Web site 257  
    Contact us xix  
RELEASE 21, 79, 133  
RELEASE(DEALLOCATE) 134  
Remote access 163  
Remote connection 21  
REOPT(VARS) 15, 132, 162, 232, 241  
REORG  
    UNLOAD EXTERNAL 18  
Resource Limit Facility 159  
Resource Limit Facility (RLF) 168  
Resource limit specification table 168  
Resource Limit Specification Tables (RLSTs) 168  
ResultSet 103  
ResultSetMetaData 104  
ResultSetMetaData object 96  
Retrieving data with bindings 72  
Retrieving data without bindings 75  
Retrieving diagnostics 78, 99  
Retrieving multiple results 98  
REXX 115  
    Development steps 120  
    Error handling 128  
    Fixed-list SELECT with parameter markers 124  
    Fixed-list SELECT without parameter markers 122  
    Non-SELECT with parameter markers 121  
    Non-SELECT without parameter markers 120  
    Pre-defined cursors 118  
    Varying-list SELECT with parameter markers 127  
    Varying-list SELECT without parameter markers 125  
RLF 159  
RLF\_CATEGORY\_B 171  
RLFASUERR 170, 172  
RLFASUWARN 170, 172  
RLFAUTH 159  
RLFERR 159  
RLFERRD 159  
RLFFUNC 169, 174  
RLFTBL 159  
RLST 168  
    Qualifying rows 178  
ROLLBACK 76  
Rollback 107  
Row-wise binding 73  
RXSUBCOMM 117

## S

SAP 238, 242  
SCHEMALIST 85  
Scrollable cursors 81  
Security 241  
SELECT INTO 79  
Sending-SQLDA 23

SENSITIVE STATIC 48  
 Servlet 90  
 SET  
     CONNECTION 21, 79  
     CURRENT APPLICATION ENCODING SCHEME 80  
     CURRENT PACKAGESET 21, 80  
     Host variable 21, 80  
 SetSQLState() 99  
 setXXX() 95, 102  
 Short prepare 148, 194  
 Siebel 239, 242  
 Size of the global statement cache 156  
 Sizing global statement cache 156  
 Special attributes 47  
 SPUFI 10, 18  
 SQL Activity 184  
 SQL Activity DML window 189  
 SQL Activity trace 202  
 SQL Activity window 185  
 SQL Statement record 210  
 SQL Statement table 207  
 SQL Statements window 198  
 SQL\_ATTR\_AUTOCOMMIT 84  
 SQL\_ATTR\_CLOSEOPEN 84, 233  
 SQL\_ATTR\_CONNECTTYPE 64  
 SQL\_ATTR\_CURSOR\_HOLD 66, 84  
 SQL\_ATTR\_MAXCONN 84  
 SQL\_ATTR\_NOSCAN 66, 84  
 SQL\_ATTR\_ROWSET\_SIZE 73  
 SQL\_BIND\_BY\_COLUMN 73  
 SQL\_CLOSE 79  
 SQL\_COMMIT 76  
 SQL\_CONCURRENT\_TRANS 82  
 SQL\_COORDINATED\_TRANS 64, 82  
 SQL\_DATA\_AT\_EXEC 67, 69  
 SQL\_DRIVER\_NOPROMPT 65  
 SQL\_ERROR 63  
 SQL\_HANDLE\_DBC 64–65  
 SQL\_HANDLE\_ENV 64  
 SQL\_HANDLE\_STMT 64  
 SQL\_NEED\_DATA 71  
 SQL\_NTS 63  
 SQL\_NULL\_DATA 63  
 SQL\_NULL\_HANDLE 64  
 SQL\_PARAM\_INPUT 67  
 SQL\_ROLLBACK 76  
 SQL\_SUCCESS 63  
 SQL92 55, 88  
 SQLAllocHandle() 64  
 SQLBindCol() 72, 79  
 SQLBindParameter() 67, 69, 79  
 SQLCA 51, 118, 176  
 SQLCCSID 119  
 SQLCloseCursor() 66, 79  
 SQLCODE 106, 118  
     +238 45  
     +239 45  
     +495 170  
     -495 170  
     -514 143  
     -518 143  
     -905 174  
 SQLColAttribute() 79  
 SQLConnect() 79  
 SQLConnectDrive() 66  
 SQLD 24, 119  
 SQLDA 23, 35, 37, 51, 119  
     Double 43  
     Single 43  
     Storage 44  
     Triple 43  
 SQLDABC 24  
 SQLDAID 24, 45  
 SQLDATA 24, 37, 119  
 SQLDescribeCol() 79  
 SQLDescribeParam() 67, 79  
 SQLDriverConnect() 65, 79  
 SQLEndTran() 76  
 SQLERRD.n 119  
 SQLERRMC 118, 176  
 SQLERRP 119  
 sqleseti 219  
 SQLException object 99  
 SQLExecDirect() 70, 79  
 SQLExecute() 79  
 SQLExecution 110  
 SQLExtendedFetch() 73, 79  
 SQLFetch() 79  
 SQLFreeHandle() 79  
 SQLFreeStmt() 66, 79  
 SQLGetCursorName() 75, 79  
 SQLGetData() 79  
 SQLGetDiagRec() 78  
 SQLGetEnvAttr() 64  
 SQLGetSQLCA() 80  
 SQLIND 24, 37, 119  
 SQLJ 9, 89, 110  
 SQLJ translator 111  
 SQLLEN 24, 119  
 SQLLEN.SQLPRECISION 119  
 SQLLEN.SQLSCALE 119  
 SQLLOCATOR 119  
 SQLMoreResults() 73, 79  
 SQLN 24  
 SQLNAME 24, 119  
 SQLNumParams() 67  
 SQLNumResultCols() 79  
 SQLParamData() 71  
 SQLPrepare() 66, 79  
 SQLPutData() 71  
 SQLRowCount() 71  
 SQLRULES 149  
 SQLSetConnectAttr() 83, 219  
 SQLSetConnection() 79  
 SQLSetCursorName() 75, 79  
 SQLSetEnvAttr() 64  
 SQLSTATE 106, 119  
     01005 45  
     01616 170  
     07003 143

- 26501 143
- 57014 174
- 57051 170
- SQLTYPE 24, 119
- SQLVAR 24, 37
- SQLWARN.n 119
- SQLWarning 99
- START RLIMIT 168
- START TRACE command 202
- Statement 101–102
- Statement caching 239
- Statement detail window 198
- Statement handle 54
- Statement invalidation 150
- Statement Object 101
- Statement reuse 148
- Statement string 22
- Statistics Detail window 188
- Storage 44
- Storage usage 154
- Stored procedure invocation 21
- Stored procedures 223
  - Security 227
- SYNCPPOINT 83
- SYSIBM.SYSROUTINES 168
- SYSSCHEMA 85
- System Health window 187
- System Management Facility (SMF) 200
- System overview window 182
- System parameters 181

## T

- TABLETYPE 85
- Thread activity 181
- Thread Detail window 184
- Three-part name 10
- Trace Activation window 203
- Transaction 106
- Translator 9
- Type 1 92
- Type 2 92
- Type 3 92
- Type 4 92

## U

- UCS-2 55
- Unicode 55
- Uniform Resource Locator 90
- UNIX System Services 118
- UNLOAD 18
- Updatable cursor 162
- UQ59527 113
- UQ62212 181
- URL 90
- Used storage 155
- USING BOTH 42–43
- USING clause 32
- Using column labels 42
- USING DESCRIPTOR 39

USING LABELS 42

## V

- VALUES INTO 80
- Varying-list SELECT statements 25
  - 33
  - With parameter markers 39
- Visual Explain 198

## W

- wasNULL() 98
- WHENEVER 80
- WITH HOLD 118
- WITH RETURN 118
- Workstation Online Monitor 180–181

## X

- X/Open 11, 50, 88



## Squeezing the Most Out of Dynamic SQL with DB2 for z/OS and OS/390

(0.5" spine)  
0.475" <-> 0.873"  
250 <-> 459 pages









# Squeezing the Most Out of Dynamic SQL

## with DB2 for z/OS and OS/390

**How to code an application using dynamic SQL**

**Dynamic SQL performance characteristics**

**Dos and don'ts of dynamic SQL**

The time has come to set the record straight. While the use of dynamic SQL in vendor packages continues to rise at a steep rate, the perception of most DB2 professionals remains that dynamic SQL is complex, resource-intensive, and difficult to manage. How true are these perceptions?

This IBM Redbook investigates some of the myths and misconceptions surrounding dynamic SQL. It presents a balanced discussion of issues such as complexity, performance, and control, and provides a “jump-start” to those venturing into this somewhat under-utilized area.

What is dynamic SQL? When is it appropriate? How do I develop an application in COBOL, REXX, Java with JDBC, or C using ODBC containing dynamic SQL? How do I obtain the best performance from it? Should I use dynamic statement caching (and if so, which flavor) for my dynamic SQL statements? How do Enterprise Solution Packages packages exploit dynamic SQL? How do I manage and control it?

In this book, we focus on these and similar questions as we show you how to maximize the benefits that can be obtained by using dynamic SQL in your applications.

**INTERNATIONAL  
TECHNICAL  
SUPPORT  
ORGANIZATION**

**BUILDING TECHNICAL  
INFORMATION BASED ON  
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
[ibm.com/redbooks](http://ibm.com/redbooks)

SG24-6418-00

ISBN 0738425265