Information Management software

IBM

DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond

- Develop and test COBOL, C, REXX, Java and SQL procedures
- Set up, control, and tune the operating environment
- Learn about IBM Data Studio and other tools

Paolo Bruni
Sabine Kaschta
Marcel Kutsch
Glenn McGeoch
Marichu Scanlon
Jan Vandensande

Redbooks

ibm.com/redbooks

International Technical Support Organization

DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond

March 2008

SG24-7604-00

Note: Before using this information and the product it supports, read the information in “Notices” on page xxxv.

First Edition (March 2008)

This edition applies to IBM DB2 Version 9.1 for z/OS (program number 5635-DB2).

Contents

Figures

Tables

Examples

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 illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

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

Redbooks (logo) ®
developerWorks®
iSeries®
i5/OS®
pureXML™
z/Architecture®
z/OS®
zSeries®
z9™
AD/Cycle®
AIX®
C/370™
CICS®
COBOL/370™
Distributed Relational Database Architecture™
DB2 Connect™
DB2 Universal Database™
DB2®
DFS™
DFSMS™
DFSORT™
DRDA®
DYNIX/ptx®
Informix®
IBM®
IMS™
Language Environment®
MQSeries®
MVS™
MVS/ESA™
OMEGAMON®
OS/2®
OS/390®
QMF™
Rational®
Redbooks®
RACF®
REXX™
RMF™
S/390®
System z™
System z9®
System/390®
Tivoli®
VisualAge®
VTAM®
WebSphere®

The following terms are trademarks of other companies:

SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

Java, JDBC, JDK, JNI, JRE, JVM, Solaris, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

ESP, Microsoft, Windows NT, Windows Vista, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

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

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-7604-00
for DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond
as created or updated on February 17, 2011.

March 2008, First Edition

The revisions of this First Edition, first published on March 31, 2008, reflect the changes and additions described below.

March 2009, First Update

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- Added a sentence at 3.3, “Sample application components” on page 24.
- Added a sentence at 9.1.12, “Stored procedure load module in memory” on page 104.
- Added UK33845 for PK57235 in Table 24-12 on page 503.
- Added a note on APAR PK64298 at page 605.
- Added “Some considerations when coding the DB2UDSMD started task” on page 744.

Changed information

- Corrections for GET_SYSTEM_INFO in Table 24-13 on page 504.
- Corrections in Chapter 22, “Multi-threaded stored procedures in the C language” on page 441.

February 2011, Second Update

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- Added a reference to *Data Studio and DB2 for z/OS Stored Procedures,* REDP-4717, in preface and back cover.
- Added considerations on closing cursors as follows::
  - Added text in bullet 9 on page 16.
  - Added bullet 13 on page 130.
  - Added text in Table 10-4 on page 137.
  - Added bullet 8 on page 166.
  - Added text in 19.1.2, “The execution life cycle of a stored procedure” on page 394.

Changed information

- Updated section 13.9.1, “Changing Java stored procedure to enable debugging in Eclipse” on page 209.
- Updated section 14.4, “Migrating to native SQL stored procedures” on page 251.
- Updated table Table 28-1 on page 736.
- Updated “Set up the Session Manager” on page 741.

Preface

This IBM® Redbooks® publication helps you design, install, manage, and tune stored procedures with DB2® 9 for z/OS®. Stored procedures can provide major benefits in the areas of application performance, code re-use, security, and integrity. DB2 has offered ever-improving support for developing and operating stored procedures.

In these days, three years is a generation in the software business; if you have DB2 9 for z/OS, this book replaces the previous *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083; it reflects the changes that have been made to DB2 stored procedures and related tools from V8 to V9.

We show how to develop stored procedures in several languages, including Java™; we explore the functions available for the z/OS platform deployment; and provide recommendations on setting up and tuning the appropriate stored procedure environment.

We talk about the external and native SQL procedures, the debugging options, the special registers, the deployment, and diagnostics.

A chapter is devoted to the increasing number of DB2-supplied stored procedures. They can be used for almost all of a DBA's tasks.

We also devote a part to tools that can be used for accelerating the development process and go into some detail about the stored procedure support provided by the latest IBM product: Data Studio. For recent information on Data Studio, refer to *Data Studio and DB2 for z/OS Stored Procedures,* REDP-4717.

The team that wrote this book

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

Paolo Bruni is a DB2 Information Management Project Leader at the International Technical Support Organization, San Jose Center. He has authored several Redbooks about DB2 for z/OS and related tools, and has conducted workshops and seminars worldwide. During Paolo's many years with IBM, in development, and in the field, his work has been mostly related to database systems.

Sabine Kaschta is a DB2 Specialist working for IBM Global Learning Services in Germany as an education consultant. She has 14 years of experience working with DB2. Before joining IBM in 1998, she worked for a third-party vendor providing second-level support for DB2 utilities. She is experienced in DB2 system programming and client/server implementations in the insurance industry in Germany. She is also a co-author of the IBM Redbooks *DB2 UDB for OS/390 and Continuous Availability*, SG24-54866; *Cross-Platform DB2 Distributed Stored Procedures: Building and Debugging*, SG24-5485-01; *DB2 UDB for z/OS Version 8: Everything You Ever Wanted to Know, ... and More*, SG24-6079; and *DB2 9 for z/OS Technical Overview*, SG24-7330.

Marcel Kutsch is a software engineer at the IBM Boeblingen Laboratory in Germany. He is currently working on the design and development of stored procedures and the administrative task scheduler component of DB2 for z/OS. Prior to joining IBM, Marcel worked for two internships at the IBM Almaden Research Center, focusing on DB2's query optimization for

DB2 on Linux®, UNIX®, and Windows®. Marcel's first assignment at IBM, after graduating from the university in 2005, was on DB2 autonomic features.

Glenn McGeoch is a Senior DB2 Consultant for IBM's DB2 for z/OS Lab Services organization in the United States, working out of San Francisco, CA. He has 30 years of experience in the software industry, with 22 years of experience working with DB2 for z/OS. He holds a degree in Business Administration from the University of Massachusetts and an MBA from Rensselaer Polytechnic Institute. Glenn worked for 19 years as an IBM customer with a focus on CICS® and DB2 application development, and has spent the last 11 years with IBM assisting DB2 customers. His areas of expertise include application design and performance, stored procedures and DB2 migration planning. He has presented to regional DB2 User Groups and to customers on various DB2 topics. Glenn co-authored the previous edition *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083.

Marichu Scanlon is an Advisory Software Engineer for IBM's Application Development Tooling organization, working out of the Silicon Valley Laboratory in San Jose, CA, and is a member of the development team for Developer Workbench and IBM Data Studio. She has over 25 years of experience in the application and software development field. She holds a degree in Electrical Engineering from the University of the Philippines and an MBA from Ateneo University, Philippines. Her areas of expertise include stored procedures and application development tooling in all platforms. She has given presentations and demonstrations on application tooling at IDUG, regional DB2 Users Groups, and Information Management conferences. She has also written several articles in DeveloperWorks on application tooling.

Jan Vandensande is an IT consultant and teacher at SOGETI in Belgium, an IBM business partner. He has over 25 years of experience in the database management field. He holds an MS degree in Electrical Engineering from the University of Louvain, Belgium. Before joining SOGETI, Jan worked as an IMS™ and DB2 system administrator in the financial sector. His areas of expertise include backup and recovery, data sharing, and performance. Jan has previously co-authored the books *DB2 for z/OS and OS/390 Version 7 Performance Topics,* SG24-6129 and *DB2 for z/OS Version 8 Performance Topics*, SG24-6465.

The authors from left to right: Jan Vandensande, Marichu Scanlon, Sabine Kaschta, Marcel Kutsch, Glenn McGeoch, and Paolo Bruni

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

Rich Conway
Emma Jacobs
Bob Haimowitz
Yvonne Lyon
Deanna Polm
Sangam Racherla
International Technical Support Organization

Judy Ruby Brown
ATS Dallas

Debra Eaton
DB2 Migration Team, Chicago

Paul Wirth
IBM Grand Rapids, MI

George M. Young
IBM Lexington, KY

Melissa Biggs
Ben Budiman
Moira Casey
Steve Chen
Clifford Chu

Zeus Courtois
Thanh Dao
Larry England
Marion Farber
Christopher Farrar
Mel Fowler
Gary Hochmuth
Grant Hutchison
Terrie Jacopi
Gopal Krishnan
Gary Lazzotti
Hung P Le
Ellen Livengood
Adrian Lobo
Vikram Manchalal
Bruce McAlister
Claire McFeely
Robert T. Miller
Tom MIller
Todd Munk
Roger Miller
Tom Miller
Barbara Nardi
Brian Payton
Jim Pickel
Akira Shibamiya
Manogari Simanjuntak
Hugh Smith
Marc Terwagne
Yumi Tsuji
Rich Vivenza
Limin Yang
Joseph Yeh
Peter Wansch
Eva Wu
Jay Yothers
Xavier Yuen
Peggy Zagelow
Emily Zhang
Liyan Zhou
Ruiming Zhou
IBM Silicon Valley Lab

Peter Aarnoutse
Jan Panneels
Sogeti Belgium

Michel Castelein
ARCIS Services Belgium

Rick Butler
BMO Canada

Glenn Anderson
IBM USA

Martin Packer
IBM UK

Leif Pedersen
IBM Denmark

Marc Terwagne
IBM Belgium

Willi Jorg
Johannes Schuetzner
IBM Germany

Boris Charpiot
Knut Stolze
Wolfgang Dunz
Benno Staebler
IBM Boeblingen Lab

Debbie Yu
Yves Tolod
Serge Rielau
IBM Toronto Lab

Fang Xing
Yong Hua (Henry) Zeng
China Software Development Lab

Thanks to the authors of the previous edition of this book.
The authors of the first edition, *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083-00, first published in March 2004, were:

Bhaskar Achanti
Suneel Konidala
Glenn McGeoch
Martin Packer
Peggy Rader
Suresh Sane
Bonni Taylor
Peter Wansch

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Part 1

Introduction

In this part we introduce the stored procedures and the contents of the book:

- Chapter 1, “Importance of stored procedures” on page 3 introduces the stored procedures and explains the reasons for their importance.
- Chapter 2, “Stored procedures overview” on page 9 summarizes the main building blocks for stored procedures.
- Chapter 3, “Our case study” on page 23 defines the environment and the case study that were implemented during this project.

1

Importance of stored procedures

Distributed applications require access to databases across a network. Unfortunately, this type of access can result in poor performance when a lot of network interactions and movements of data are involved. A stored procedure runs on the database server with a goal of reducing the network traffic. Since their introduction in DB2 for MVS/ESA™ Version 4, the roles of stored procedures in an enterprise have continued to grow. In this chapter we discuss what stored procedures are, why they are important, and how pervasive they are becoming.

This chapter contains the following:

- What stored procedures are
- Benefits of stored procedures
- Use of stored procedures
- Multi-tiered applications and stored procedures

1.1 What stored procedures are

A stored procedure is a user-written program that can be called by an application with an SQL CALL statement. It is a compiled program that is stored at a DB2 server, and can execute SQL statements.

Stored procedures can be called locally (on the same system where the application runs) and remotely (from a different system). However, stored procedures are particularly useful in a distributed environment since they considerably improve the performance of distributed applications by:

- Reducing the traffic of information across the communication network
- Splitting the application logic and encouraging an even distribution of the computational workload
- Providing an easy way to call a remote program

The advantages provided by stored procedures are clear when comparing them to a standard distributed application where the client may be a workstation or a Java client as shown in Figure 1-1. We see that the client communicates with the server separately for each embedded SQL request.

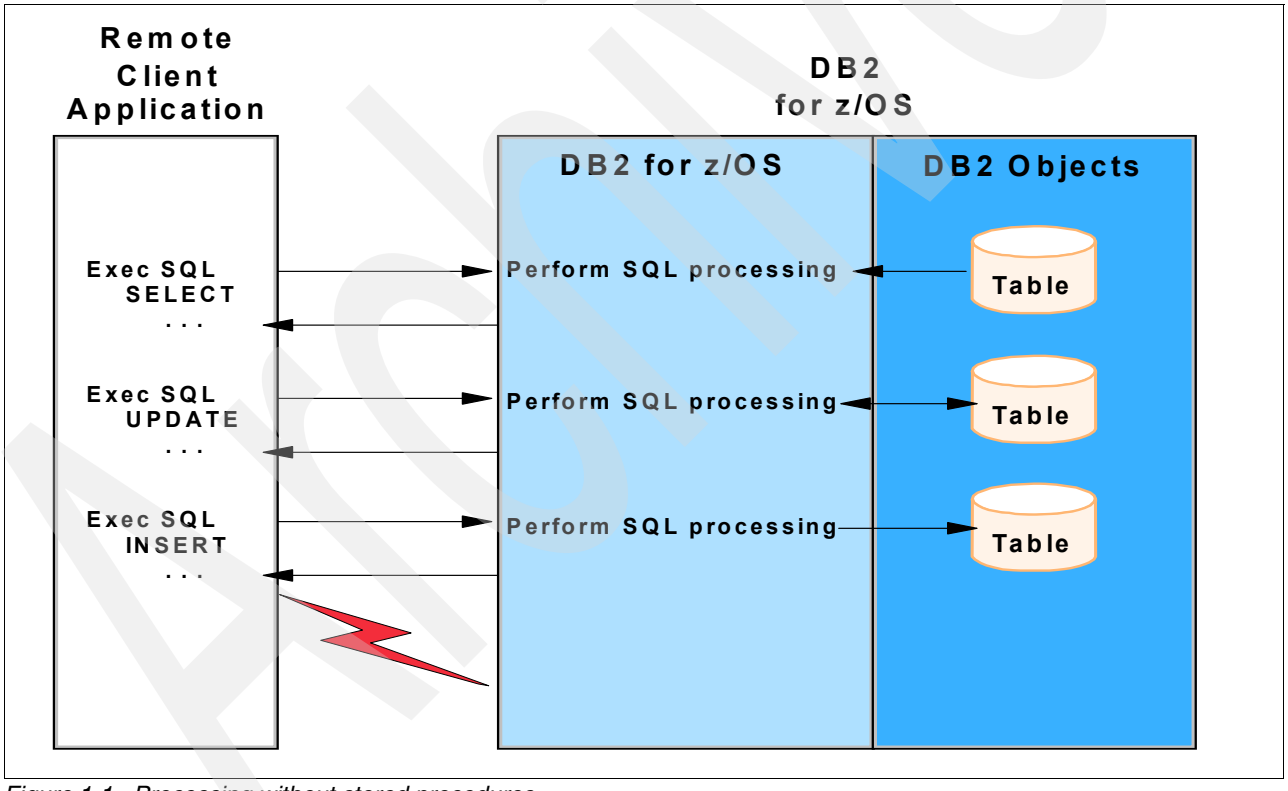

Figure 1-1 Processing without stored procedures

The client communicates with the server with a send and receive operation through the network for each SQL statement embedded in the program. As a consequence, the elapsed time is increased by network transmission time or Java overhead, the remote CPU path length is higher than for a local SQL cost, and DB2 locks are held until commit.

Figure 1-2 shows the stored procedure solution. We have moved the embedded SQL to the server, reducing the network traffic to a single call and return.

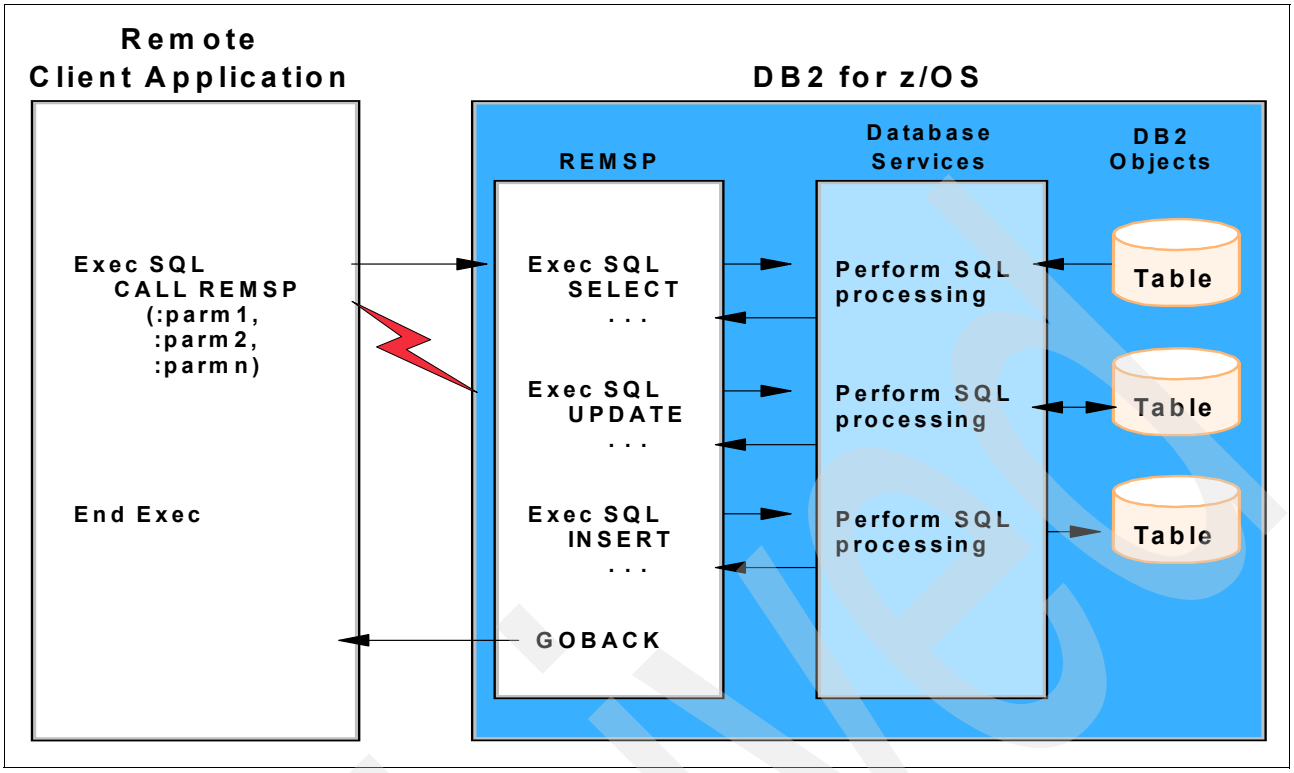

Figure 1-2 Processing with stored procedures

The same SQL previously executed by the client has been stored on the server and is called by the client whenever necessary. The invocation is treated as a regular external call:

- The application waits for the stored procedure to terminate.
- Parameters can be passed back and forth.

1.2 Benefits of stored procedures

Your programming productivity can be improved by using stored procedures when you develop and maintain applications.

- Modularity in application development is encouraged with stored procedures.

  Client developers can focus on their application logic details, while stored procedure programmers develop appropriate DB2 server access.
- When any application calls the stored procedure, it processes data in a consistent way according to the rules defined in the stored procedure.
- If you need to change the rules, you only need to make the change once in the stored procedure, not in every application that calls the stored procedure.
- Reduced network traffic for distributed applications:
  - A typical application requires two trips across the network for *each* SQL statement.
  - Grouping SQL statements into a stored procedure results in two trips across the network for each *group* of statements, resulting in better performance for applications.
  - Improved application security.
  - Sensitive business logic runs on the DB2 server.

  - End users do not need table privileges.
- Access to features that exist only on the server:
  - Stored procedures can have access to commands that run only on the server.
  - They might have the advantages of increased memory and disk space on server machines.
  - They can access any additional software installed on the server.
- Enforcement of business rules:
  - You can use stored procedures to define business rules that are common to several applications.
  - This is another way to define business rules, in addition to using constraints and triggers.
- Application integration solutions:
  - You can use stored procedures to easily access non-DB2 resources.
  - With the use of WebSphere® MQ, you can coordinate access to multiple data and platforms.
- Cost of ownership reduction
  - DRDA® activity is a candidate for zIIP rerouting. A smaller percentage of work is redirected to zIIP for remote non-SQL native procedures, just the CALL, COMMIT and result set processing.
  - Stored procedures written in Java can take advantage of zAAP engines.
  - Native SQL procedures have richer SQL functions and remote native SQL procedures, running as enclaves in a DBM1 address space, are candidates for zIIP rerouting with DB2 V9.

1.3 Use of stored procedures

Stored procedures are becoming quite pervasive. They can be used:

- For distributed applications to:
  - Distribute the logic between a client and a server
  - Perform a sequence of operations at a remote site
  - Combine results of query functions at a remote site
  - Control access to database objects
  - Remove SQL applications from the workstation and prevent workstation users from manipulating the contents of sensitive SQL statements and host variables
  - Dynamically invoke static SQL rather than use the Java Data Base Connectivity (JDBCs) dynamic SQL approach
- To access non-DB2 resources:
  - VSAM files
  - Flat files
  - IMS or CICS transactions
  - DL/I databases
  - MVS/APPC conversations

  - Utilize Recoverable Resource Services (RRS) to coordinate two-phase commit processing of recoverable resources
- When the details of trigger and User Defined Function (UDF) processing go beyond the scope of SQL statements, stored procedures can be called for the application logic.
- To transport messages using MQSeries® functions that:
  - Notify other business processes that an event has taken place
  - Forward information from one process to many other processes
  - Aggregate information from multiple sources to create warehouses and Operational Data Stores (ODSs). See *Building the Operational Data Store on DB2 UDB Using IBM Data Replicator, WebSphere MQ Family, and DB2 Warehouse Manager,* SG24-6513 for more details.

Stored procedures provide the easiest way to perform a remote call and distribute the execution logic of an application. Because stored procedures reside on the data base server and therefore can access programs and features of the server, a myriad of application solutions are available.

Stored procedures *should* be considered for a client/server application when the procedure does at least one of the following:

- Executes two or more SQL statements

  Remote SQL statements can create many network send and receive operations, which results in increased processor costs.
- Contains manipulative and or logical processing that must be consistent across multiple processes, and the goal is to provide coding that is reusable and therefore requires minimal maintenance efforts.
- Accesses host variables or table columns for which you want to guarantee security and integrity.

1.4 Multi-tiered applications and stored procedures

The concept of tiered applications presents a solution to design architects that uses distributed resources dynamically with the ability to insert software and hardware when and where they are needed.

Two-tiered client/server applications might require extensive and expensive deployment strategies for DB2 enablers and business logic.

Multi-tiered (three or more tiers) client/server applications greatly reduce the deployment issues because DB2 enablers and business logic are kept on the lower tiers. The client has the presentation layer of the system while the business logic is on the middle tier and the database access on the server tier.

This flexibility comes with many benefits, including:

- Each tier can be developed, installed, and maintained independently.
  - Code from existing business logic and database access can be separated and reused as deemed necessary.
  - With good planning, you can reuse common software solutions.
- Any number of hardware and software configurations can be deployed to produce a modular packaging of information services.

- A substantial improvement in the maintenance of client/server applications.

Because stored procedures reside and execute on the database server (or the server tier), they can be developed and installed with minimal maintenance activity. Stored procedures support and encourage what is, by far, the most pervasive of current computing trends.

2

Stored procedures overview

Stored procedures are a key database feature for developing robust distributed applications. They can be written in any language supported by the database server including C, C++, COBOL, z/OS Assembler, PL/I, REXX™, SQL Procedures language (both external and native) and Java. Support for invoking a stored procedure and processing its result sets is built into many client applications, as well as ODBC, JDBC™ and SQL for Java (SQLJ) standards.

There are different types of stored procedures, as well as operating environments. Because stored procedures are DB2 objects, they must be defined to the DB2 catalog and then the integrity of the parameters to be passed is protected.

In this chapter we provide a brief functional overview of stored procedures.

This chapter contains the following:

- Stored procedure types
- Example of a stored procedure flow
- DB2 catalog tables
- Behind the scenes of stored procedures

2.1 Stored procedure types

There are two common criteria by which stored procedures are often categorized: by the language in which they are written; and by the type of address space in which they run. In this book we discuss three types of stored procedures:

- External high-level language procedures
- External SQL language procedures
- Native SQL language procedures

The first two types of stored procedures result in an external load module being created and they run in Workload Manager (WLM) address spaces. The third type of procedure does not result in a load module being created and they run in the DBM1 address space. For each of the last two types of stored procedures the source code is written entirely in SQL, with the program logic being part of the stored procedure definition (within the CREATE PROCEDURE statement itself). For external high-level language stored procedures, the stored procedure definition and the program logic are two separate components.

In the next few sections we provide a more detailed description of each type of stored procedure.

2.1.1 External high-level language procedures

An external high-level language stored procedure is written by a developer in one of the programming languages available on the server. The available languages on the z/OS server are: COBOL, PL/I, C, C++, Assembler, REXX, and Java. An external high-level language stored procedure is much like any other SQL application—it can include static or dynamic SQL statements, IFI calls and DB2 commands issued through IFI. With the exception of REXX, you can pre-compile, compile, link and bind the host source program to create the appropriate load modules and packages. Example 2-1 highlights the necessary components of a COBOL stored procedure. From this example you can see that the skeleton is identical to a COBOL subprogram.

Example 2-1 COBOL skeleton of a storage procedure

```
ID DIVISION.
PROGRAM-ID. XFEREMP.
.
.
.
LINKAGE SECTION.
01 ...
PROCEDURE DIVISION USING ...
   EXEC SQL
   ...
   END-EXEC.
   .
   .
   .
   GOBACK.
```

The source code for an external high-level language stored procedure is separate from the definition of the stored procedure. A stored procedure is only bound to a package and not a plan because it utilizes the invoking plan's thread. The stored procedure load module must be placed in a load library that is included in the STEPLIB DD concatenation in the WLM startup JCL (except for Java stored procedures). See Part 3, "Developing stored procedures" on

page 89 for programming details, and Part 2, “Operating environment” on page 37 for setting up the stored procedure environment.

The CREATE PROCEDURE statement is used to inform the system of the name of the load module and what parameters are expected when the procedure is called, as well as other execution and environment options. See Chapter 9, “Defining stored procedures” on page 91 for details.

Example 2-2 shows the information from a CREATE PROCEDURE statement that DB2 needs to locate the load module and to know what source language will be used to create the stored procedure.

Example 2-2 Sample storage procedure CREATE statement

```
CREATE PROCEDURE XFEREMP
   (parameter information)
   EXTERNAL NAME XFEREMP
   LANGUAGE COBOL
   .
   .
   .
```

Whenever possible, stored procedures should be prepared as reentrant programs. Using reentrant code provides the following performance benefits:

- A single copy can be shared by multiple tasks in the WLM SPAS (stored procedures address space). This decreases the amount of virtual storage used for code in the SPAS.
- The stored procedure does not have to be loaded into storage every time it is called.

However, if your stored procedure cannot be reentrant, link-edit it as non-reentrant and non-reusable. The non-reusable attribute prevents multiple tasks from using a single copy of the stored procedure at the same time.

2.1.2 External SQL language procedures

The SQL procedures are stored procedures written in the SQL Procedures language. The SQL Procedures language is based on SQL extensions as defined by the Persistent Stored Modules (SQL/PSM) standard. SQL/PSM (an ISO/ANSI standard for SQL:2003) is a high-level language similar to other RDBMS languages such as Transact SQL (T/SQL) from Sybase, and Procedural Language SQL (PL/SQL) from Oracle®, which extends SQL to procedural support.

The ISO/ANSI SQL:2003 is an open solution for SQL among database management system vendors that support the SQL ISO/ANSI standard. Because this approach is widely used by other RDBMS providers, this support makes it possible to port stored procedures from the other vendors to DB2 and vice versa.

The SQL Procedures language implementation supports constructs that are common to most programming languages. It supports the declaration of local variables, assignment of expression results to variables, statements to control the flow of the procedure, receiving and returning of parameters from and to the invoker, returning result sets and error handling.

Like an external high-level language stored procedure, an external SQL language procedure consists of a stored procedure definition, source code for the stored procedure program and a load module (which will be a C language module for external SQL language procedures). Most of the CREATE PROCEDURE options are the same. In both cases you specify the name of the external load module and your input and output parameters the same way. The

definitions differ in that the source code for the external high-level language procedure is not included in the DDL, while the source code for the external SQL language procedure is included in the DDL.

Example 2-3 shows the statement to CREATE an external SQL language procedure to update employees' salaries.

Example 2-3 CREATE PROCEDURE sample for external SQL language procedures

```
CREATE PROCEDURE UPDATE_SAL
( IN INRATE DECIMAL (7,2), IN INEMPNO CHAR(6))
   LANGUAGE SQL
   UPDATE EMP
      SET SALARY = SALARY * INRATE
         WHERE EMPNO = INEMPNO
```

For the details on creating external SQL language procedures, see Chapter 14, "External SQL procedures" on page 233.

2.1.3 Native SQL language stored procedures

DB2 9 for z/OS introduced a new type of stored procedure, the native SQL language procedure. A native SQL language stored procedure is very similar to an external SQL language stored procedure in that the source code is included within the CREATE PROCEDURE statement. The difference is in the executables (and also in a richer SQL language.)

No external load module is created for native SQL language procedures. The entire executable is contained within DB2. When you create a native SQL stored procedure, the procedural SQL statements are stored in the DB2 catalog and directory, as are the SQL statements that are used for accessing your DB2 data. As a result, when you prepare a native SQL procedure, the entire executable is contained within DB2. This simplifies the deploy process since you don't have to worry about code level management in load libraries and in WLM application environments.

The advantage of this approach is that DB2 can manage these stored procedures directly. The stored procedures run in the DBM1 address space, so there is no need to create a WLM environment to manage the procedures. Since native SQL procedures run under an enclave SRB instead of a TCB, if they are remote, they are also eligible to be run in a System z9™ Integrated Information Processor (zIIP) if one is available.

A zIIP is a specialty engine for System z9 mainframes. z/OS manages and directs work between the general purpose processor (the portion of the mainframe that traditionally has handled the z/OS workload) and the zIIP. Work that runs on the zIIP does not incur software charges based on the service units consumed, therefore it is a very attractive lower-cost alternative to running workloads on a general purpose processor.

Figure 2-1 shows how work is redirected from the z/OS general purpose processor to the zIIP specialty engine. On the left side of the picture you can see that all the DRDA work is executing on the general purpose processor (the CP). On the right side of the picture, with the zIIP specialty engine available, some of the DRDA work is executing on the zIIP, thus reducing the software costs of work running on the CP and also reducing the workload on the CP so you can now run additional work there if needed.

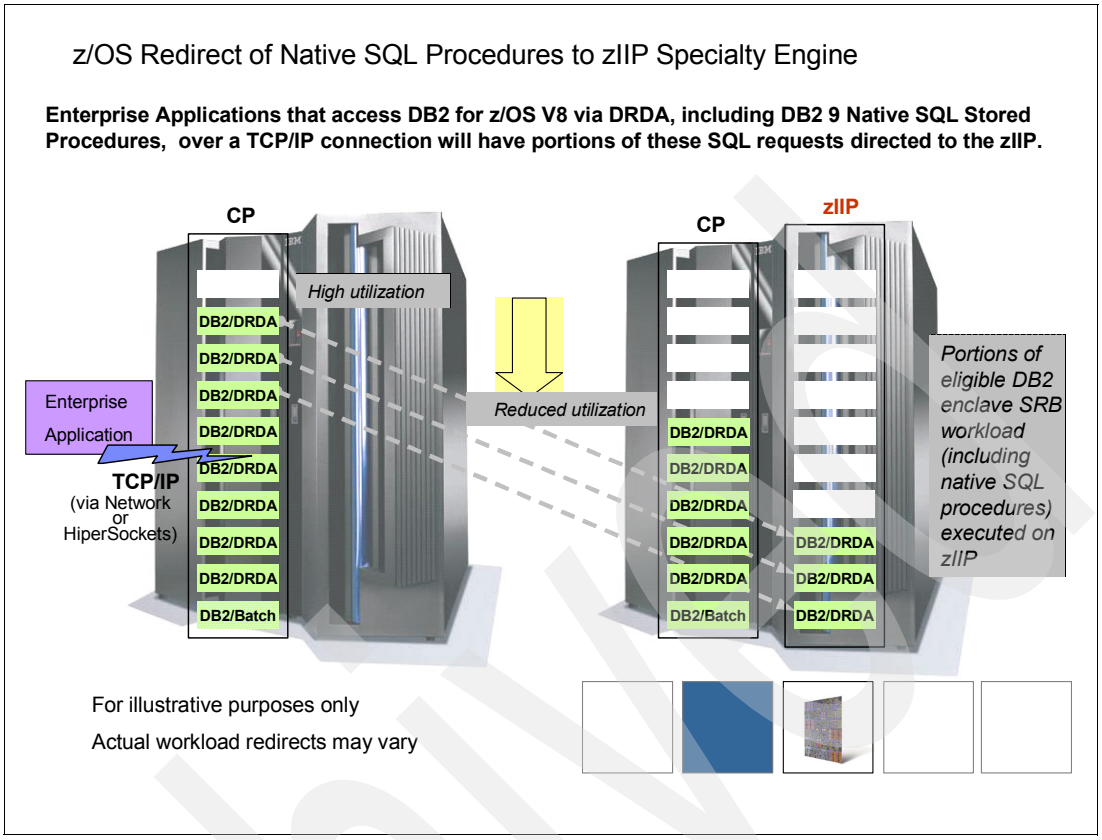

Figure 2-1 z/OS redirect of native SQL procedures to the zIIP specialty engine

For more information on the zIIP specialty engine and what portion of your DB2 workload is eligible to be redirected to a zIIP, see Chapter 4 of *DB2 9 for z/OS Performance Topics*, SG24-7473. For the details on creating native SQL procedures, see Chapter 15, “Native SQL procedures” on page 253.

2.2 Example of a stored procedure flow

In Figure 2-2, we show the statement flow of execution of a simple stored procedure. We have an example of a client calling a stored procedure to assist with the transfer of employees to different departments. We would like to reduce the number of network transmissions that are required if the client application made all of its own database calls. Instead of ten network sending and receiving messages, we have only four. The stored procedure also handles a significant number of table manipulations.

An employee is transferred to another department and, optionally, may also become the new department manager. The stored procedure XREFEMP first inserts into the XFER_EMPPA table any existing project activities that the employee has affiliations with, and removes the corresponding rows from the EMPPROJACT table. If any rows were deleted, a project management process needs to be informed. This process will handle any project activities that are incomplete because of an employee transfer. We chose to use the MQSeries functions of DB2 to notify the process from a performed routine, SEND-MSG-TO-PROJ. The details of using MQSeries functions can be found in *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

As a stored procedure, this function is available to remote client workstation applications, online CICS or IMS transactions, local or remote batch programs, triggers and user-defined functions. All processes that invoke this stored procedure must prepare and pass the appropriate parameters.

Be assured that this stored procedure will do the rest, or report any errors that are encountered.

Throughout this book, we examine the components, services and considerations that must be integrated to ensure success.

Figure 2-2 represents the flow of our sample stored procedure.

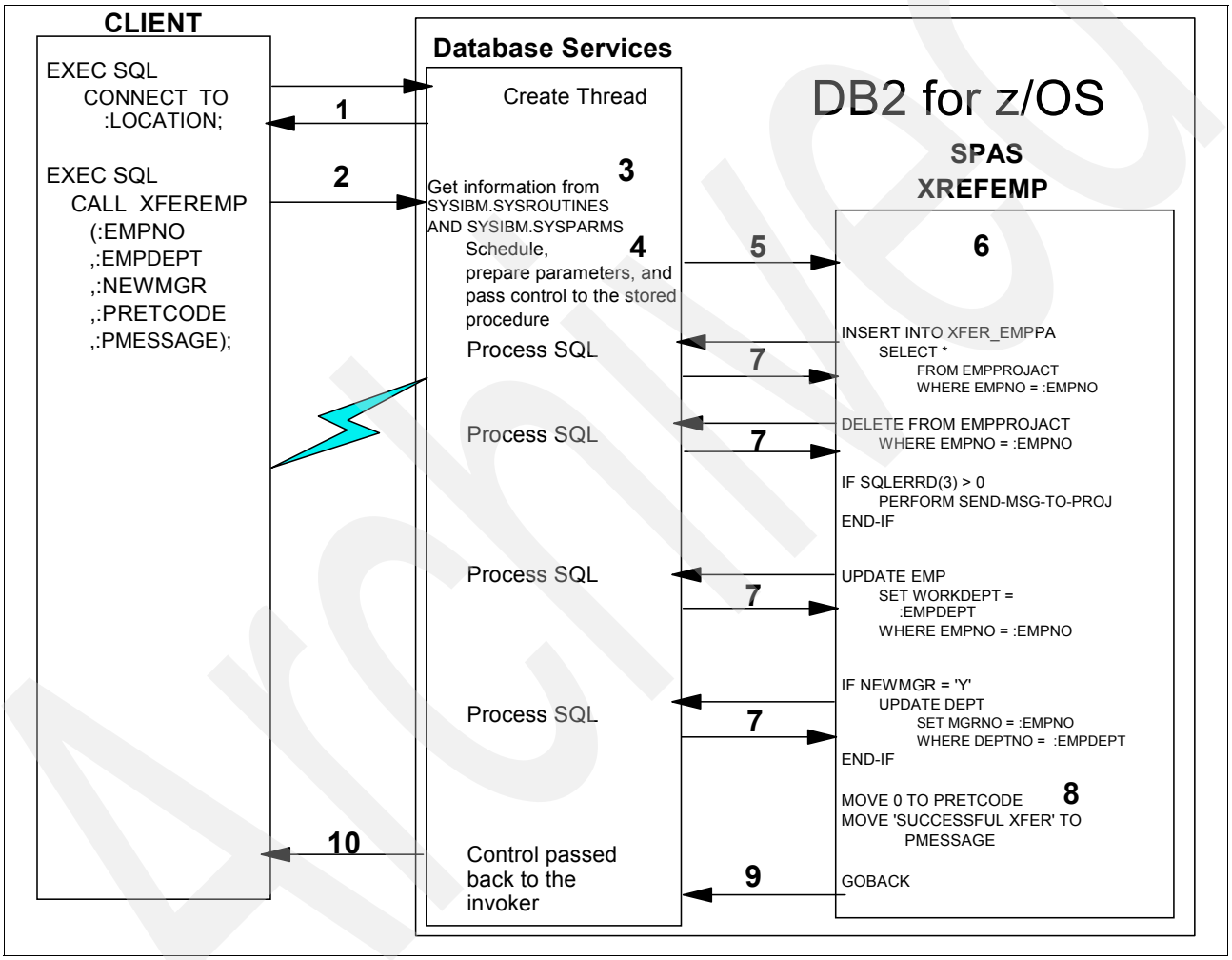

Figure 2-2 Stored procedure that transfers employees - statement flow

1. A thread must be created for each application that needs DB2 services. If the stored procedure is called from a remote client, the thread is created when the client application issues the SQL CONNECT statement. If the application is local, the thread is created when the first SQL statement is executed. After the thread is created, SQL statements can be executed.
2. When a client application issues an SQL CALL statement, the stored procedure name and the I/O parameters are passed to DB2.

3. When DB2 receives the SQL CALL statement, it searches in the SYSIBM.SYSROUTINES catalog table for a row associated with the stored procedure name. From this table, DB2 obtains the load module associated with the stored procedure and the run environment information. It also searches SYSIBM.SYSPARMS to gather the parameter information, such as whether a parameter is input, output or input/output, and the data type of each expected parameter. *Notice that catalog information is cached to avoid I/O*.

   For details on the search of SYSIBM.SYSROUTINES and how DB2 determines which version of the stored procedure to execute, see 18.2, “Versioning of stored procedures” on page 373.

4. With the exception of native SQL language procedures, stored procedures are executed in address spaces that run fenced away from the DB2 code. Multiple Workload Manager (WLM) address spaces may be made available for stored procedures. Starting with DB2 for z/OS V8, all newly created stored procedures must use the WLM-established stored procedure address spaces. Those stored procedures that existed prior to Version 8 and were created to run in the DB2-managed address space will continue to run, but all new stored procedures must run in a WLM-managed stored procedure address space. In DB2 9 for z/OS all stored procedures, except for native SQL procedures, are WLM-managed.

   WLM goal mode is mandatory. You can specify a number of task control blocks (TCBs) in this address space available for stored procedures. Each stored procedure is executed under one TCB. After DB2 has searched the SYSIBM.SYSROUTINES table, an available TCB to be used by the stored procedure is selected, and the stored procedure address space is notified to execute the stored procedure.

5. When the stored procedure address space executes a stored procedure, the thread that was created for the client application is reused for the execution. This has the following implications:
   - CPU cost is low because DB2 does not create a new thread.
   - Accounting is on behalf of the client application.
   - For static SQL, the OWNER of the client program must have execute privilege on the stored procedure package. For dynamic SQL issued by the stored procedure, security is checked against the user of the client program, unless the DYNAMICRULES(BIND) option was specified when binding the package for the stored procedure. No sign-on or connection processing is required.
   - The stored procedure address space uses the LE/370 product libraries to load and execute the stored procedure. Through SYSIBM.SYSROUTINES, you can pass runtime information for LE/370 when the stored procedure is executed.

6. Control is passed to the stored procedure along with the input and output parameters.
   - The stored procedure can issue most SQL statements. It also can have access to non-DB2 resources.
   - The stored procedure can either perform all the database access and return the output to the calling program as output parameters (see the next step) or it can open a cursor to build a result set and let the calling program fetch from the result set.
   - Any processing done by the stored procedure is considered a logical continuation of the client application's unit of work. Thus, locks acquired by the stored procedure are released when the unit of work terminates. If DB2 has been so instructed, through the definition of the stored procedure, it can commit the logical unit of work upon return to the caller.

7. Before terminating, the stored procedure assigns values to any output parameters and returns control to DB2.

8. DB2 copies the output parameters received from the stored procedure to the client application parameter area and returns control to the client application.
9. The calling program receives the output parameters and continues the same unit of work. If the stored procedure returned a result set, then the client application can fetch rows from the result set until there are no more rows to fetch, upon which time the client application should close the cursor. The client application implicitly or explicitly issues the COMMIT statement. DB2 can implicitly commit as soon as the stored procedure returns control to the client application based upon the value of the COMMIT ON RETURN option in the CREATE PROCEDURE statement. See 9.1, “CREATE or ALTER PROCEDURE parameters” on page 92 for details of the CREATE statement and the options available. If the client application and the stored procedures used during this execution update at different sites, the two-phase commit protocol is used.
10. DB2 returns control to the invoking program.

2.3 DB2 catalog tables

Stored procedures are DB2 objects, therefore they must be defined with DDL. There are two catalog tables that are affected by this definition: SYSIBM.SYSROUTINES and SYSIBM.SYSPARMS.

SYSIBM.SYSROUTINES contains one row for each created stored procedure. The information contained in this table is from the CREATE PROCEDURE statement. There are columns in this catalog table describing the runtime environment, language, number of parameters, parameter style, whether or not result sets can be returned, whether DB2 should execute a commit when returning to the caller, and so on.

Example 2-4 is a query that you can use to retrieve information about the runtime environment of a stored procedure.

Example 2-4 Query to retrieve stored procedure runtime information

```
SELECT RTRIM("SCHEMA")||'.'||RTRIM(OWNER)||'.'||RTRIM("NAME")
   , PARM_COUNT,PARAMETER_STYLE,"LANGUAGE", "COLLID"
     , SQL_DATA_ACCESS, "DBINFO"
     , STAYRESIDENT
     , WLM_ENVIRONMENT
     , PROGRAM_TYPE
     , COMMIT_ON_RETURN, RESULT_SETS
     , EXTERNAL_NAME
     , RUNOPTS
 FROM "SYSIBM".SYSROUTINES
 WHERE NAME = 'EMPDTL2C' AND SCHEMA = 'DEVL7083'
```

Example 2-5 is QMF™-developed output of the SYSIBM.SYSROUTINES runtime information query from Example 2-4.

Example 2-5 QMF output for query to retrieve stored procedure runtime information

```
PROCEDURE NAME: DEVL7083.PAOLOR1.EMPDTL2C
**************************************************************
COLLECTION ID: DEVL7083  LANGUAGE: COBOL  EXT. NAME: EMPDTL2C

WLM ENVIRONMENT: DB9AWLM  STAY RESIDENT: N  MAIN/SUB S

RUN OPTIONS: ()
```

```
SQL DATA ACCESS: M   RESULT SETS RETURNED: 0

COMMIT ON RETURN: N

# OF PARAMETERS:  10  PARAMETER STYLE: N   DBINFO: N
```

For details on producing the QMF report, refer to *DB2 QMF Reference Version 9 Release 1,* SC18-9685.

SYSIBM.SYSPARMS contains one row for each parameter defined in stored procedures. The parameter information comes from the CREATE PROCEDURE statement. The columns in this catalog table describe the parameter definitions: name, data types, input (notated with a P), output, input/output (notated with a B), and optionally whether the parameter row is associated with a locator or table.

Figure 2-3 shows the relationship between rows in SYSIBM.SYSPARMS and SYSIBM.SYSROUTINES. It shows the stored procedure's relevant columns, and how they can be merged together to produce helpful info for users that want the parameter information.

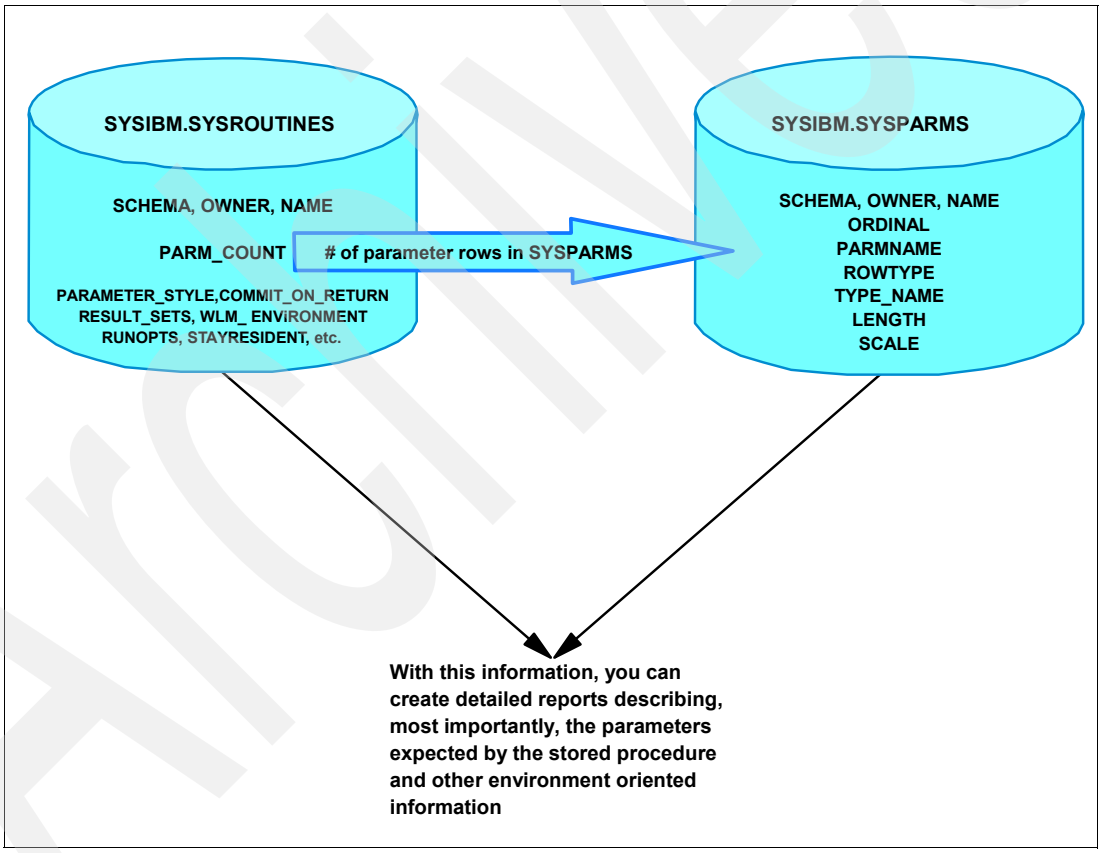

Figure 2-3 Relationship between SYSIBM.SYSROUTINES and SYSIBM.SYSPARMS

A query can then be created to gather information from the catalog about a stored procedure and its expected parameters. Example 2-6 shows such a query.

Example 2-6 Query to retrieve information about expected parameters of a stored procedure

```
SELECT RTRIM("SCHEMA")||'.'||RTRIM(OWNER)||'.'||RTRIM("NAME")
   , ' '||STRIP(DIGITS(PARM_COUNT),L,'0')||' PARMS'||'/'||PARAMETER_STYLE,' '
   ,' ' , "DBINFO"||' ',SQL_DATA_ACCESS||' '
```

```
      , COMMIT_ON_RETURN||' ', DIGITS(RESULT_SETS) RS
  FROM "SYSIBM".SYSROUTINES
  WHERE NAME = 'EMPDTL2C'
    AND SCHEMA = 'DEVL7083'
UNION ALL
  SELECT RTRIM("SCHEMA")||'.'||RTRIM(OWNER)||'.'||RTRIM("NAME")
     , DIGITS(ORDINAL),PARMNAME
     , RTRIM(TYPENAME)||'('||STRIP(DIGITS("LENGTH"),L,'0')||
             CASE TYPENAME
             WHEN 'DECIMAL' THEN ','||STRIP(DIGITS(SCALE),L,'0')||')'
             ELSE ')'  END
     , '  '||ROWTYPE, '  '||"LOCATOR", '  '||"TABLE"
     , ENCODING_SCHEME
  FROM "SYSIBM".SYSPARMS
  WHERE NAME = 'EMPDTL2C'
    AND SCHEMA = 'DEVL7083'
ORDER BY 1,2
```

Example 2-7 shows a QMF-generated report of the SYSIBM.SYSROUTINES and SYSIBM.SYSPARMS query in Example 2-6. The SQL/QMF combination produces a report of procedure information grouped with the parameters.

Example 2-7 QMF generated report from query in Example 2-6

```
*                                                                 D|L C|T
                                                                  A|O O|A
                                                               D|R T|C M|B
                                                               B|O A|A M|L
                                                               I|W C|T I|E
                                                               N|T C|O T|
                                                               F|Y E|R R|
                        PARM STYLE                             O|P S|S E|
PROCEDURE NAME          AND PARM # PARM NAME      PARM DEF      |E S|  T|
------------------------ ---------------------------------------- --- ---
DEVL7083.PAOLOR1.EMPDTL2C  10 PARMS/N                           N   M   N
                        00001      PEMPNO         CHAR(6)       P   N   N
                        00002      PFIRSTNME      VARCHAR(12)   0   N   N
                        00003      PMIDINIT       CHAR(1)       0   N   N
                        00004      PLASTNAME      VARCHAR(15)   0   N   N
                        00005      PWORKDEPT      CHAR(3)       0   N   N
                        00006      PHIREDATE      DATE(4)       0   N   N
                        00007      PSALARY        DECIMAL(9,2)  0   N   N
                        00008      PSQLCODE       INTEGER(4)    0   N   N
                        00009      PSQLSTATE      CHAR(5)       0   N   N
                        00010      PSQLERRMC      VARCHAR(250)  0   N   N

                                                                        *
```

Note: The QMF queries and forms used in this chapter can be downloaded from the ITSO Web site as additional material. Download instructions are in Appendix B, “Additional material” on page 887.

Additional catalog tables are used for Java stored procedures and for SQL language stored procedures, both external and native. Information about Java stored procedures is contained in the following tables:

- SYSIBM.SYSJARCLASS_SOURCE
- SYSIBM.SYSJARCONTENTS
- SYSIBM.SYSJARDATA

- SYSIBM.SYSJAROBJECTS
- SYSIBM.SYSJAVAOPTS
- SYSIBM.JAVAPATHS

Information about external SQL language and native SQL language stored procedures is contained in the following tables:

- SYSIBM.SYSENVIRONMENT
- SYSIBM.SYSROUTINES_OPTS
- SYSIBM.SYSROUTINES_SRC
- SYSIBM.SYSROUTINESTEXT
- SYSIBM.SYSROUTINESAUTH

For more details about creating Java, external SQL language and native SQL language procedures, see Chapter 13, “Java stored procedures” on page 181, Chapter 14, “External SQL procedures” on page 233 and Chapter 15, “Native SQL procedures” on page 253.

2.4 Behind the scenes of stored procedures

In order to better understand the execution environment of stored procedures, let us take a look “behind the scenes” of a stored procedure execution. Figure 2-4 illustrates the relationship between application code, DB2 address spaces, WLM application environments, and so on.

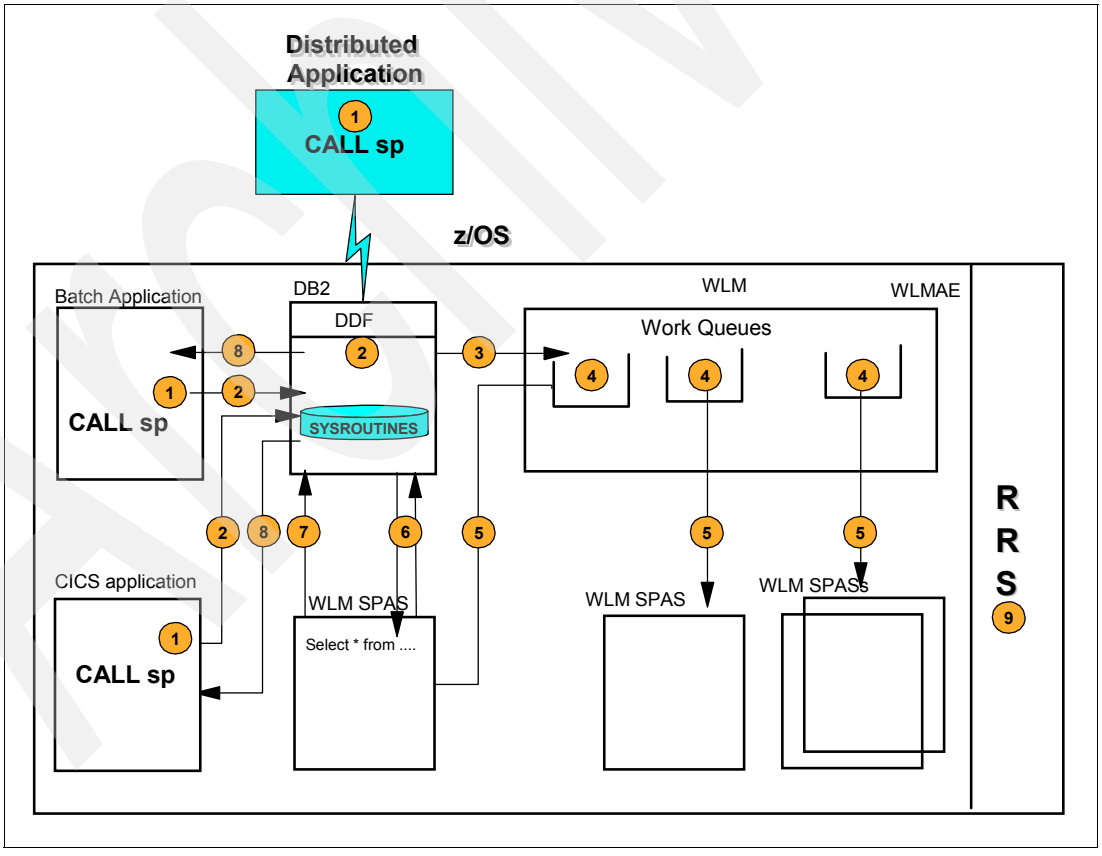

Figure 2-4 The system management of stored procedures

Let us see what happens when you call a stored procedure:

1. The distributed application (or batch, or CICS) issues the SQL CALL statement to invoke a stored procedure.
2. The request is received and handled by the DDF address space, and passed to DB2. For qualified CALLs DB2 uses the three-part name (location, schema, procedure name). For unqualified CALLs (the recommended way), DB2 for z/OS implicitly uses the current server (the location) and SQL path (the schema). If the procedure name is specified as a literal, the SQL path is the value of the PATH bind option that is associated with the calling package or plan. If the procedure name is specified with a host variable, the SQL path is the value of the CURRENT PATH special register.

   DB2 searches the SYSIBM.SYSROUTINES catalog table using the procedure name and, after verifying authorizations and parameter definitions, retrieves the collection ID (COLLID) and WLM application environment (WLM_ENVIRONMENT) names associated with the stored procedure.
3. DB2 sends a request to WLM to schedule the stored procedure in an application environment.
4. WLM places the request in one of its queues. WLM maintains one queue for each combination of application environment and service class. For example, if you have three service classes DDFWKLD (for DDF workload), ONLNWKLD (for CICS workload) and BATCWKLD (for batch workload) for one application environment WLMAE, WLM maintains three separate queues for the same WLMAE. The requests in a queue are processed in an FIFO manner.
5. Once your request has its turn, WLM checks for the availability of a WLM stored procedure address space (SPAS).
   - If no WLM SPAS exists, WLM starts a new one and executes the stored procedure.
   - If WLM SPAS exists and free TCBs are available, it executes the stored procedure.
   - If WLM SPAS exists and there is no availability of free TCBs:
     - If WLM is meeting the performance goal set for the service class, it waits for availability of a TCB in one of the active WLM SPAS. The time DB2 waits depends on the TIMEOUT VALUE (on installation panel DSNTIPX). If the wait time exceeds the TIMOUT VALUE, the request is timed out and the caller receives a -471 SQLCODE with reason code 00E79002.
     - If WLM is *not* meeting the performance goal set for the service class, a new WLM SPAS is started that executes the stored procedure.
   - If WLMAE is in a stopped or quiesced state, it sends a return code back to the calling program.
6. Once the stored procedure is scheduled, DB2 now executes the SQL inside it (if SQLs are present).

   DB2 first uses the following sequence to determine the collection ID:

   a. CURRENT PACKAGE PATH special register in the storage procedure program or in the calling application or even in the CREATE PROCEDURE statement instead of setting COLLID (APAR PK59752)
   b. CURRENT PACKAGESET in the stored procedure program
   c. COLLID collection name in CREATE PROCEDURE
   d. CURRENT PACKAGESET in the calling application
   e. COLLID in the calling application
   f. PKLIST in the calling application

7. When the stored procedure reaches the return statement, it passes control back to DB2 with or without results, depending on the logic.
8. DB2 passes control back to the calling program with or without results, depending on the logic.
9. Since the transactions involving stored procedures can span multiple address spaces, RRS plays the role of coordinator for all the resources between all address spaces involved in the transaction.

Steps 1 to 8 are repeated for each execution of a stored procedure. If a stored procedure calls another stored procedure, all these steps are again repeated.

3

Our case study

This chapter describes the environment, the data, and the sample applications that are used for the case studies discussed throughout the book.

All the files required to reconstruct the sample applications can be downloaded from the IBM Redbooks Web site:

http://www.ibm.com/redbooks

Refer to Appendix B, “Additional material” on page 887 for more details.

This chapter contains the following:

- Overview
- The environment
- Sample application components
- Populating the tables with XML data
- Naming conventions

3.1 Overview

Most of the examples in this book reference two of the sample tables provided with DB2 for z/OS: the DEPT and EMP tables. Sample stored procedure code as well as sample calling programs are provided for two application scenarios: return employee details for a specific employee number that has been provided; and return a result set of employees for a department, with the results sorted by salary. Samples are provided for COBOL, C, REXX, Java, external SQL Procedures language and native SQL Procedures language.

3.2 The environment

The software levels used for our case studies are as follows:

- DB2 9 for z/OS for all examples except where noted
- z/OS Version 1.9
- CICS Transaction Server Version 3.1
- IMS Version 9
- IBM Tivoli® OMEGAMON® XE for DB2 Performance Expert on z/OS Version 4.1
- QMF Version 9.1
- DB2 9 for Linux UNIX and Windows
- IBM Debug Tool for z/OS Version 8.1
- IBM Data Studio Version 1.1
- Java Developer Kit Version 1.5

The hardware configuration for our case studies is as follows:

z9 (2094-S18) processor in an LPAR with 4 GB central storage and six logical processors:

- Two general processors
- Two Integrated Information Processors (zIIP)
- Two Application Assist Processors (zAAP)

3.3 Sample application components

The sample applications chosen for our case studies were intentionally kept simple. There are two basic sets of stored procedures, with variations on each for some of the languages to show specific features. The first set of stored procedures selects six columns from the EMP sample table for a given value of employee number (column EMPNO) supplied by the calling program. Columns with various data types were selected to show how the data types are handled by the various programming languages, both within the stored procedure, and within the calling program. Variations of this stored procedure are included to show how to use different parameter styles.

The second set of stored procedures selects the department name from the DEPT sample table, using a department number value (column DEPTNO) supplied by the calling program, then selects all rows from the EMP sample table where the WORKDEPT column matches the department number supplied. The EMP table rows are returned in a result set to the calling program. Variations are shown for Java stored procedures to show use of a positioned iterator and use of a named iterator.

Some additional samples are provided to show how to do the following:

- Write multi-threaded stored procedures in C
- Write stored procedures that interact with CICS and IMS
- Write applications to call DB2-supplied stored procedures
- Call stored procedures from triggers and user defined functions
- Access XML data

Some of the Java and SQL language stored procedures were imported into IBM Data Studio in our DB2 9 for z/OS system to show you how to create and debug stored procedures using the GUI capabilities of IBM Data Studio. See 27.5.1, “Creating a new native stored procedure using the wizard” on page 689, 27.5.2, “Creating an external SQL stored procedure from the wizard” on page 694 and 27.5.3, “Creating a Java stored procedure from the wizard” on page 696 for details on creating native SQL, external SQL and Java language stored procedures within the IBM Data Studio product. See 28.3, “Debugging SQL procedures on z/OS, Linux, UNIX, and Windows” on page 746 for details on debugging sample SQL stored procedure EMPDTLSS.

We did not code and test any Assembler language stored procedures. If you would like to see a sample Assembler language stored procedure, refer to SDSNSAMP library member DSNTWR, which is the load module for the WLM_REFRESH stored procedure. Instructions to prepare this stored procedure can be found in the *DB2 Version 9.1 for z/OS Installation Guide*, GC18-9846.

Our sample stored procedures were run on DB2 9 for z/OS, except where noted, using copies of the sample tables provided with DB2. Table 3-1 lists the sample tables.

Table 3-1 Sample tables

Object Type	Name	Description
Table	EMP	Sample employee table
	DEPT	Sample department table
	CATALOG	Sample catalog table
	CUSTOMER	Sample customer table. See footnote a.
	EMP_PHOTO_RESUME	Sample employee photo resume table
	PRODUCT	Sample product table
	PURCHASEORDER	Sample purchase order table
	SUPPLIERS	Sample suppliers table. See footnote a.
a.This table contains XML data. Refer to 3.4, “Populating the tables with XML data” on page 34 for details on how to populate this table.		

Table 3-2 on page 26 lists the objects that were used for the COBOL programming examples.

Table 3-2 Objects for COBOL programming examples

Object Type	Name	Description
Stored procedure	EMPDTL1C	COBOL stored procedure that returns employee data for a supplied employee number, using parameter style GENERAL
	EMPDTL2C	COBOL stored procedure that returns employee data for a supplied employee number, using parameter style GENERAL WITH NULLS
	EMPDTL3C	COBOL stored procedure that returns employee data for a supplied employee number, using parameter style DB2SQL with NO DBINFO
	EMPDTL4C	COBOL stored procedure that returns employee data for a supplied employee number, using parameter style DB2SQL with DBINFO
	EMPRSETC	COBOL stored procedure to return a result set of employees for a given department
	EMPAUDTS	COBOL stored procedure invoked by a trigger, accessing transition variables
	EMPAUDTX	COBOL stored procedure invoked by a trigger, accessing transition tables
	EMPODB1C	COBOL stored procedure accessing IMS data via ODBA
	EMPEXC1C	COBOL stored procedure invoking CICS transaction through EXCI
	EMPEXC3C	COBOL stored procedure calling DB2-supplied stored procedure DSNACICS to invoke a CICS transaction
Calling program	CALDTL1C	COBOL program that calls stored procedure EMPDTL1C
	CALDTL2C	COBOL program that calls stored procedure EMPDTL2C
	CALDTL3C	COBOL program that calls stored procedure EMPDTL3C
	CALDTL4C	COBOL program that calls stored procedure EMPDTL4C
	CALRSETC	COBOL program that calls stored procedure EMPRSETC
Called program	EMPEXC2C	COBOL CICS program invoked through EXCI by stored procedure EMPEXC1C
	EMPEXC4C	COBOL CICS program invoked through DSNACICS by stored procedure EMPEXC3C
User-defined function	EMPAUDTU	COBOL user-defined function invoked by a trigger for data validation

Table 3-3 on page 27 lists the objects that were used for the C programming examples.

Table 3-3 Objects for C programming examples

Object Type	Name	Description
Stored procedure	EMPDTL1P	C stored procedure that returns employee data for a supplied employee number, using parameter style GENERAL
	EMPDTL2P	C stored procedure that returns employee data for a supplied employee number, using parameter style GENERAL WITH NULLS
	EMPRSETP	C stored procedure to return a result set of employees for a given department
	RUNSTATP	Multi-threaded C stored procedure that returns a result set, using parameter style GENERAL WITH NULLS
Calling program	CALDTL1P	C program that calls stored procedure EMPDTL1P
	CALDTL2P	C program that calls stored procedure EMPDTL2P
	CALRSETP	C program that calls stored procedure EMPRSETP

Table 3-4 lists the objects that were used for the Java programming examples.

Table 3-4 Objects for Java programming examples

Object Type	Name	Description
Stored procedure	EMPDTLSJ	JDBC stored procedure that returns employee data for a supplied employee number - EmpDtlsJ.java
	EMPDTLSMJ	JDBC stored procedure used to illustrate using Multiple Jars (MJ) for V9 Java stored procedures - EmpDtlsMJ.java
	EmpDtls2	Variation on EMPDTLSJ for CASE A when changing external name
	EmpDtls3	Variation on EMPDTLSJ for CASE B when changing external name
	EMPRSETJ	JDBC stored procedure to return a result set of employees for a given department - EmpRsetJ.java
	EMPDTL1J	SQLJ stored procedure that returns employee data for a supplied employee number - EmpDtl1J.sqlj
	EMPRST1J	SQLJ stored procedure to return a result set of employees for a given department - EmpRst1J.sqlj
	EMPRST2J	SQLJ stored procedure that updates the employee salary for a given department using a result set and positioned iterator EmpRst2J.sqlj & EmpRst2J_UpdByPos.sqlj
	EMPCLOBJ	JDBC stored procedure returning a CLOB - EmpClobJ.java
	EMPPHOTJ	JDBC stored procedure returning a BLOB - EmpPhotJ.java
	EXTRACT _JAR	JDBC stored procedure to extract a jar (BLOB of 100 MB) from DB2. ExtractJarSp.java
	EMPRMTEJ	JDBC stored procedure making a remote Stored Procedure Call. EmpRmteJ.java

Object Type	Name	Description
Calling program	CALDTLSJ	Java application program that calls the EMPDTLSJ stored procedure. CalDtlsJ.java
	EmpClobSpServlet	Java servlet that calls the stored procedure EMPCLOBJ and writes out the output to a Web page. EmpClobSpServlet.java
	EmphotoSpServlet	Java servlet calls the stored procedure EMPPHOTJ and writes out an Image to the Web browser. EmpPhotoSpServlet.java
Java program	ExtractJar	Java application that extracts a jar from DB2. ExtractJar.java
	Getters_staff	The Java class that has the methods called/used in EMPDTLSMJ
Iterator	EmpRst2J_UpdByPos.sqlj	Iterator declaration file for Java stored procedure EMPRST2J

Table 3-5 lists the objects that were used for the REXX programming examples.

Table 3-5 Objects for REXX programming examples

Object Type	Name	Description
Stored procedure	EMPDTLSR	REXX stored procedure that returns employee data for a supplied employee number
	EMPRSETR	REXX stored procedure to return a result set of employees for a given department

Table 3-6 lists the objects that were used for the External SQL language programming examples.

Table 3-6 Objects for External SQL language programming examples

Object Type	Name	Description
Stored procedure	EMPDTLSS	SQL stored procedure that returns employee data for a supplied employee number
	EMPRSETS	SQL stored procedure to return a result set of employees for a given department
	EMPDTLV8	SQL stored procedure to illustrate V8 enhancements
JCL	SQLSPCUS	JCL to promote an SQL language stored procedure from a test environment to a production environment

Table 3-7 lists the objects that were used for the Native SQL language programming examples.

Table 3-7 Objects for Native SQL language programming examples

Object Type	Name	Description
Calling program	Median_RS	Sample java program that calls a specific version of the MEDIAN_RESULT_SET SQL procedure

Object Type	Name	Description
Stored procedure	CALC_SALARY	SQL procedure that calculates the sum of all salaries in a table “staff” with the FOR loop construct
	DIVIDEPR	SQL procedure that implements CURRENT and STACKED DIAGNOSTICS statements
	GOTO	SQL procedure that employs the GOTO construct to branch to a label defined at a higher level
	MEDIAN_RESULT_SET	SQL procedure that calculates the median value of all salaries stored in a table “staff” and returns a result set containing all rows with a higher salary than the calculated median value. Two samples are provided, MEDIANV1 and MEDIANV2, to demonstrate the capability to ALTER an SQL procedure to create a new version.
	NODIFF	SQL procedure that employs the FOR loop construct to check if all values in a column of type integer feature the same value
	REBIND_PACKAGES	SQL procedure that issues multiple REBIND PACKAGE subcommands by calling the procedure ADMIN_COMMAND_DSN inside of a FOR loop construct
	SCOPECUR	SQL procedure that demonstrates scoping of cursor definitions
	SCOPECUR2	SQL procedure that demonstrates scoping of cursor definitions. This sample first prepares an SQL statement and opens the respective cursor later when it is needed.
	SCOPEHND	SQL procedure that demonstrates the scoping of condition handler declarations
	SCOPELAB	SQL procedure that illustrates the scoping of label names
	SCOPELA2	SQL procedure that illustrates the scoping of label names and how to avoid ambiguity problems with referenced column names
	SCOPEVAR	SQL procedure that demonstrates how variable declarations can be scoped
	SCPCNHDL	SQL procedure that demonstrates the usage of compound statements in a condition handler
	TYPES	SQL procedure that creates variables of data type BIGINT, BINARY, VARBINARY and DECFLOAT

Table 3-8 lists the objects that were used for the examples of developing multi-threaded stored procedures in the C language.

Table 3-8 Objects for multi-threaded C language examples

Object Type	Name	Description
Stored procedure	RUNSTATP	Multi-threaded C stored procedure to return a result set of employee for a given department
Calling program	RunstatPDriver	Java application that calls stored procedure RUNSTATP

Table 3-9 lists the objects that were used for the examples of applications that call DB2-supplied stored procedures.

Table 3-9 Objects for DB2-supplied stored procedure examples

Object Type	Name	Description
Calling program	AdminDB2Command	Java application program that calls the DB2-supplied stored procedure ADMIN_COMMAND_DB2, which executes multiple DB2 commands - DISPLAY DDF DETAIL
	AdminDataSet	Java application program that calls the DB2-supplied stored procedures ADMIN_DS_WRITE, ADMIN_DS_BROWSE, ADMIN_DS_RENAME, ADMIN_DS_SEARCH, ADMIN_DS_LIST, and ADMIN_DS_DELETE to manage data sets
	AdminJob	Java application program that calls the DB2-supplied stored procedures ADMIN_JOB_SUBMIT, ADMIN_JOB_QUERY, ADMIN_JOB_FETCH, and ADMIN_JOB_PURGE to illustrate a way to perform remote JCL administration
	AdminUtilityExecution	Java application program that shows how the RUNSTATS process could be automated. It calls the DB2-supplied stored procedure DSNACCOX to determine all table spaces that require RUNSTATS to be run on. In a following program step, the parallel utility scheduler stored procedure ADMIN_UTL_SCHEDULE is invoked to execute RUNSTATS on the recommended table spaces.
	AdminSystemInformation	Java application program that calls the DB2-supplied stored procedures ADMIN_INFO_SSID, ADMIN_INFO_HOST, DSNWZP, and DSNUTILU, to display various system information
	AdminUNIXCommand	Java application program that calls the DB2-supplied stored procedure ADMIN_COMMAND_UNIX to issue a “ls -lat” UNIX System Services command
	AdminWLMRefresh	Java application program that calls the DB2-supplied stored procedure WLM_REFRESH, to refresh a WLM application environment
	AdminDSNSubcommand	Java application program that calls the DB2-supplied stored procedure ADMIN_COMMAND_DSN to issue a REBIND PACKAGE command

Table 3-10 lists the objects that were used for the examples of applications that use the DB2-supplied task scheduler. For each application component we describe to which use case it applies.

Table 3-10 Objects for DB2-supplied task scheduler use cases

Object Type	Use Case#	Name	Description
Calling program	1	AdminSchedule1	Java application program that calls the DB2-supplied stored procedure ADMIN_TASK_ADD to schedule a one time execution of the DSNUTILU stored procedure with static input parameters (for use with use case - 1)
	3	AdminSchedule3	Java application program that calls the DB2-supplied stored procedure ADMIN_TASK_ADD to schedule a non-regularly recurring execution of the WLM_REFRESH stored procedure with dynamic input parameters. Appendix A, “Samples for using DB2-supplied stored procedures” on page 807 shows only the parameter initialization section of the java program.
	4	AdminSchedule4	Java application program that calls the DB2-supplied stored procedure ADMIN_TASK_ADD to schedule stored procedure execution, which is triggered by the stored procedure execution in Use Case 1. The triggered task is another call to the DSNUTILU stored procedure.
	4	AdminScheduleR	Java application program that calls the DB2-supplied UDF table function ADMIN_TASK_LIST to obtain all expired tasks from the scheduler task list. The program then invokes the DB2-supplied stored procedure ADMIN_TASK_REMOVE on every row returned.
Trigger	2	USER.TR_WLM_REFRESH	Example of a before insert trigger that invokes the ADMIN_TASK_ADD stored procedure to schedule a call to the WLM_REFRESH stored procedure. The DDL for this trigger, along with the DDL for the trigger and table below, are included in file SCHEDTRI.JCL.
	2	USER.TR_WLM_REFR_REMOVE	Example of a before delete trigger that invokes the ADMIN_TASK_REMOVE stored procedure to remove a scheduled WLM_REFRESH call from the scheduler task list. The DDL for this trigger, along with the DDL for the trigger above and the table below, are included in file SCHEDTRI.JCL.

Object Type	Use Case#	Name	Description
Table	2	USER.TAB_WLM_REFR	Both triggers depend on this table, and make use of transition variables. The DDL for this table, along with the DDL for the two triggers above, are included in file SCHEDTRI.JCL.
	3	USER.INPUT_PARMS	Table required for the input parameters for the stored procedure call in calling program AdminSchedule3

Table 3-11 lists the objects that were used for the examples to invoke the Common SQL API stored procedures.

Table 3-11 Objects for invoking the Common SQL API stored procedures

Object Type	Name	Description
Java program	SPDriver	Java class that contains the main method and instantiates an object of the class SPWrapper.java
	SPWrapper	Java class that provides the service routines to eventually CALL the Common SQL API stored procedures SYSPROC.GET_SYSTEM_INFO, GET_CONFIG, and GET_MESSAGE
	GetConfDriver	Java application program that invokes the DB2-supplied stored procedure GET_CONFIG with a valid XPATH string, that filters the IP address of the DB2 data server
XML Document	CompleteMode	Sample XML_COMPLETE_MODE document to be used with the Common SQL API example

We attempt to show as many different combinations of languages and environments as possible. The stored procedures in the case studies all are executed on a z/OS platform, but the client applications are executed from either z/OS or from IBM Data Studio. In many cases we did not provide a calling application for all the stored procedures in each language; instead we used IBM Data Studio to test each of the stored procedures. Since many of the stored procedures are identical in function, with the only difference being the source language, a calling application in any source language is identical. For example, COBOL calling program CALRSETC, which calls COBOL stored procedure EMPRSETC, can just as easily call the Java stored procedure EMPRSETJ.

Table 3-12 lists QMF queries and forms that display catalog information about the stored procedures used in our case studies. See 2.3, “DB2 catalog tables” on page 16 for details on the queries and their respective reports. Query QRPARMER and the associated form FRPARMER are not discussed in the book, but variations of query QRPARM70 and form FRPARM70 are discussed.

Table 3-12 QMF objects

Object Name	Description
QRPARMER	QMF query that retrieves information about a stored procedure and associated parameters as well as external name and runtime options
QRPARM70	QMF query that retrieves information about a stored procedure and associated parameters

Object Name	Description
QRTNONLY	QMF query that retrieves stored procedure runtime options
FRPARMER	QMF form associated with QMF query QRPARMER
FRPARM70	QMF form associated with QMF query QRPARM70
FRTNONLY	QMF form associated with QMF query QRTNONLY

Table 3-13 lists the triggers that are used in our examples to show their interaction with stored procedures and user defined functions.

Table 3-13 DB2 triggers

Object Name	Description
EMPTRIG1	Example of a trigger calling a stored procedure passing transition variables
EMPTRIG2	Example of a trigger calling a stored procedure passing transition tables
EMPTRIG3	Example of a trigger invoking a UDF for validation

Table 3-14 lists a REXX exec that we used for configuration management and change management purposes. See 18.4, “Notes on REXX execs” on page 384 for more details.

Table 3-14 REXX execs for configuration management

REXX Name	Description
DDLMOD	REXX exec that modifies the DDL for a CREATE PROCEDURE statement so that it can be promoted to a different DB2 environment, based on the values specified in a configuration file, and generates SYSIN cards which can be used for WLM refresh, DROP stored procedure and SET CURRENT SQLID.

Table 3-15 lists the jobs that we used to set up the IMS environment for our ODBA example. See 23.2.1, “Accessing IMS databases through the ODBA interface” on page 478 for a detailed description of the ODBA setup process.

Table 3-15 Jobs for IMS ODBA setup

Job Name	Description
IMS01 through IMS13	Jobs that coincide with examples about accessing IMS from stored procedures
DB9AODBA	JCL to define WLM application environment for stored procedures that access IMS
WLMDEF	Sample WLM application environment definition for DB9AODBA

Table 3-16 lists the objects that were used in the examples of developing a stored procedure using Data Studio.

Table 3-16 Objects for Data Studio examples

Object Type	Name	Description
DDL	CreateTable.sql	DDL to create the CUSTOMER and PURCHASEORDER tables used in the examples

Object Type	Name	Description
Data file	CUSTOMER	Data for loading CUSTOMER table
	PURCHASEORDER	Data for loading PURCHASEORDER table

Table 3-17 lists sample JCL that can be used for setting up the Session Manager for the Unified Debugger.

Table 3-17 Jobs for Unified Debugger Session Manager setup

Job Name	Description
SESSMGR1	Job to define the Unified Debugger Session Manager Started Task to RACF®
SESSMGR2	Job to create a file in the HFS to hold the Environment settings used when the Unified Debugger Session Manager runs as a Started Task on z/OS
SESSMGR3	Job to create the Started Task JCL for DB2UDSMD. This is used to launch the Unified Debugger Session Manager on z/OS.

3.4 Populating the tables with XML data

The DB2supplied sample tables CUSTOMER and PURCHASEORDER are defined with XML columns. However, the installation job, DSNTIJEX, that created them does not populate them with rows. For our case study, we use IBM Data Studio to load data into these tables. Refer to Chapter 27, “The IBM Data Studio” on page 643 for more details on installing and using Data Studio.

- In the Database Explorer, open the DB9A connection.
- Expand the folders **DB9A** → **Schemas** → **DEVL7083** → **Tables** → **CUSTOMER.**
- Right-click **CUSTOMER**, then click **Data** → **Load.** See Figure 3-1.

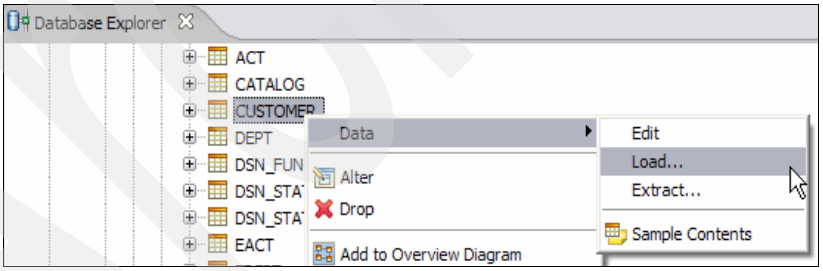

Figure 3-1 Data Studio, Database Explorer, Load data into a table

- In the Load Data dialog, click **Browse**.
- Point the file browser to where you downloaded the additional materials, and where the CUSTOMER.DATA file is. Click **OK**.
- The Load Data dialog is complete. See Figure 3-2.
- Click **Finish**.

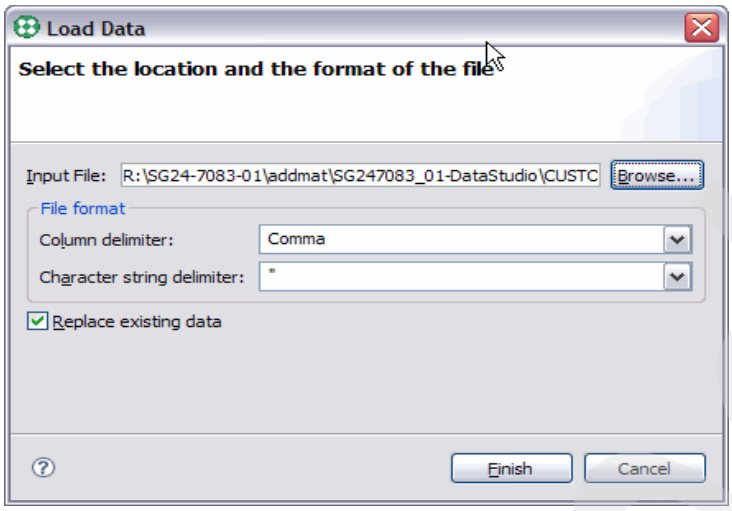

Figure 3-2 Load Data dialog, input file specified

In the Output View, note that 6 rows were inserted into the CUSTOMER table.

3.5 Naming conventions

The sample DDL and source code refer to various table names and stored procedure names. We adopted standard naming conventions for these objects. In addition, we adopted standards for schema, package, and collection names. Chapter 4, “Setting up and managing Workload Manager” on page 39 refers to WLM application environments that are used for the case studies. The WLM application environment naming standard is also described in 3.5.2, “WLM application environment names” on page 35. Many of the naming conventions have continued to use the last four digits of the previous book publication number, 7083.

3.5.1 Table qualifiers, schema names, collection IDs and package owners

Since we show examples of how to migrate a stored procedure from a development environment to a production environment, we developed the following conventions for table qualifiers, schema names, collection IDs, and package owners:

- DEVL7083 is used for all DB2 objects in our development environment.
- PROD7083 is used for all DB2 objects in our production environment.

3.5.2 WLM application environment names

We defined multiple WLM application environments for our case studies. See Chapter 4, “Setting up and managing Workload Manager” on page 39 to understand the criteria we used for defining WLM environments. The naming convention for the WLM application environments is as follows:

```
DB9AWLM for COBOL, C and external SQL procedures
DB9AWLMJ for Java
DB9AREXX for REXX
DB9AEXCI for COBOL procedures that call CICS
DB9AODBA for COBOL procedures that call IMS
```

We chose this naming scheme for ease of development. We do not recommend that you combine stored procedures of different languages in the same application environment. See

“Grouping of stored procedures within application environments” on page 414 for information to help you decide on how to group your application environments.

Part 2

Operating environment

In this part we describe the setup of the operating environment that hosts the stored procedures. The topics discussed here will interest primarily the MVS™ system programmers, but the DBA will need to have at least some general understanding of the terminology and definition criteria.

This part contains the following chapters:

4

Setting up and managing Workload Manager

In this chapter we discuss setting up Workload Manager (WLM) environments using WLM panels. We also give some general recommendations regarding the number of application environments to set up and the number of TCBs that should run in each address space.

We also look at how to diagnose SQLCODE -471/-00E79002, which means that WLM could not get a task to run the stored procedure. WLM service classes where stored procedures run need to be examined, and how to obtain and read the reports. We also look at:

- WLM service classes where stored procedures run need to be examined, and how to obtain and read the reports.
- And provide a little insight into the WLM algorithms for starting another address space.

This chapter contains the following:

- Workload Manager overview
- WLM Application Environment recommendations
- Setting up WLM for DB2 stored procedures

4.1 Workload Manager overview

Workload Manager (WLM) is a component of z/OS. WLM performs workload management functions for the operating system. The purpose of workload management is to balance the available system resources to meet the demands of z/OS subsystems work managers such as CICS, batch, TSO, UNIX System Services, and WebSphere in response to incoming work requests.

DB2 uses WLM to allocate workload requests for DB2 stored procedures. As requests for stored procedures come into DB2, WLM determines whether additional resources such as a new WLM-managed address space are needed in order to process the requests. WLM manages the number of tasks (TCBs) that can run in each address space, and starts additional address spaces as needed. Starting with DB2 for z/OS Version 8, all new external stored procedures must run in WLM-managed address spaces.

4.2 WLM Application Environment recommendations

In this section we describe some recommendations for setting up WLM Application Environments. Additional information on tuning your WLM environment can be found in Chapter 20, “Server address space management” on page 423.

The setting of the WLM environments is dependent upon the routines that are being executed under the same DB2 subsystem.

There are many DB2-supplied stored procedures invoked directly or through various tools. It is very convenient to group them based on their characteristics. This information is provided in Table 24-13 on page 504.

Table 4-1 on page 41 summarizes the possible aggregations into WLM Application Environments for user-defined procedures.

- COBOL, C/C++ and PL/I-related stored procedures

  One or more WLM Application Environments are recommended for executing COBOL, C/C++, and PL/I user stored procedures. The same WLM Application Environment can be used when the same STEPLIB requirements and same runtime LE options apply to all languages. When different STEPLIB data sets or options are required, then additional WLM Application Environments need to be created to support the processing of the different STEPLIB data sets.

  You might also want to aggregate or separate these stored procedures to take into account nesting (see 10.3.1, “Nested stored procedures” on page 130), or performance (see Chapter 20, “Server address space management” on page 423), or operational considerations.

- Debugging COBOL, C/C++ and PL/I-related stored procedures

  When Debug Tool is used to debug COBOL, C/C++, or PL/I stored procedures, the stored procedure is compiled with the TEST option. This option, along with the presence of Debug Tool either in the WLM Application Environment STEPLIB or LINKLST, causes Debug Tool modules to be loaded during execution. It is recommended to separate these language stored procedures into a separate WLM Application Environment to be used when executing in DEBUG mode.

- REXX-related stored procedures

  The TCB setting is required to be 1 for REXX stored procedures.

- Java related stored procedures (resettable or unresettable mode)

  A separate WLM Application Environment is recommended for executing the users' Java stored procedures.

  The max NUMTCB setting for resettable Java stored procedures should be no more than 8, since a Java Virtual Machine (JVM™) is loaded for each NUMTCB value.
- External SQL stored procedures are similar to COBOL, C/C++, and PL/I stored procedures.
- Native SQL stored procedures are executed in the DBM1 address space.

Table 4-1 How many types WLM environments should be defined?

Stored procedure name	NUMTCB	Comments
COBOL, C/C++, PL/I	10-40	
Debugging COBOL, C/C++ and PL/I related stored procedures	10-40	
REXX	1	REXX stored procedures must run in a WLM procedure with NUMTCB = 1. If they execute in a procedure with NUMTCB>1, unpredictable results, such as an 0C4 will occur.
Java (resettable mode)	5-8	A Java Virtual Machine (JVM) is loaded for each NUMTCB when the WLM procedure includes a JAVAENV DD. The JAVAENV DD is required for executing Java stored procedures.
Java (non-resettable mode)	20-40	Each JVM is started in non-resettable mode and is never reset. This allows you to run many more Java stored procedures. See "Non-resettable JVMs" on page 417 for more details.
External SQL stored procedures	10-40	Must have one unauthorized data set. COBOL, C/C++, PL/I stored procedures can share if the JCL is comparable

The NUMTCB value that you choose for each application environment will vary by language. Table 4-1 provides general recommendations for the WLM procedure NUMTCB setting for the different language stored procedures. These recommendations are based on available resources and should be tuned accordingly.

4.3 Setting up WLM for DB2 stored procedures

This section describes the WLM Application Environment panels and some sample JCL procedures for running some of the sample stored procedures used in our case studies. We also discuss WLM setup for some of the DB2-supplied stored procedures used by the Data Studio or Developers Workbench.

Each WLM Application Environment needs to be defined to WLM using the WLM panels. Once into WLM, enter 9 for Application Environment. Create a new definition or copy an existing WLM definition. We made the Application Environment Name and the Procedure Name that run in this environment the same, because the JCL in this procedure is unique to DB9A. When the procedure name is the same as the WLM Application Environment name, it is easy to monitor the active WLM definitions, and know which one to refresh, quiesce or resume by using the same name that's displayed on the SDSF Display Active panel. Example 4-1 shows a sample Application Environment definition panel.

When you want to be able to easily change the NUMTCB value, we recommend that you leave this parameter off the WLM Application Environment definition, and specify it on the procedure that executes in the WLM Application Environment. If the NUMTCB parameter is specified on both the WLM Application Environment and the procedure that executes in the WLM Application Environment, the value in the WLM Application Environment overrides the value in the procedure.

Example 4-1 WLM Application Environment definition for general DB2 stored procedures

```
Application-Environment  Notes  Options  Help
 ------------------------------------------------------------------------------
                    Modify an Application Environment
 Command ===> ____________________________________________________________

 Application Environment Name . : DB9AWLM
 Description  . . . . . . . . . . DB9A General DB2 SPs
 Subsystem Type . . . . . . . . . DB2
 Procedure Name . . . . . . . . . DB9AWLM
 Start Parameters . . . . . . . . DB2SSN=DB9A,APPLENV=DB9AWLM
                                  ____________________________________
                                  ___________________________________

 Starting of server address spaces for a subsystem instance:
 1   1.  Managed by WLM
     2.  Limited to a single address space per system
     3.  Limited to a single address space per sysplex
```

Example 4-2 shows the procedure definition we used for executing many of our DB2 system stored procedures including DSNTJSPP. This WLM environment contains all APF authorized STEPLIB data sets, so that we can run the DB2 system WLM_REFRESH stored procedure here as well, which requires all APF-authorized STEPLIB data sets. We specify the NUMTCB value on the procedure, and not the WLM Application Environment definition, due to ease of maintenance. Changes to JCL procs can be made available by refreshing WLM Application Environment, while changes to WLM Application Environment definitions need re-installing z/OS service policy at an LPAR or sysplex level. If the LE runtime SCEERUN library is not included in your system LINKLIST, you need to uncomment the STEPLIB DD for SCEERUN.

Example 4-2 Our procedure for executing many DB2-supplied stored procedures

```
//*************************************************************
//*    JCL FOR RUNNING THE WLM-ESTABLISHED STORED PROCEDURES
//*    ADDRESS SPACE
//*       RGN     -- THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//*       DB2SSN  -- THE DB2 SUBSYSTEM NAME.
//*       NUMTCB  -- THE NUMBER OF TCBS USED TO PROCESS
//*                  END USER REQUESTS.
//*       APPLENV -- THE MVS WLM APPLICATION ENVIRONMENT
//*                  SUPPORTED BY THIS JCL PROCEDURE.
//*
//*************************************************************
```

```
//DB9AWLM  PROC RGN=OK,APPLENV=XXXXXXXX,DB2SSN=DB9A,NUMTCB=40
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DISP=SHR,DSN=DB9A9.SDSNEXIT
//         DD  DISP=SHR,DSN=DB9A9.SDSNLOAD
```

Example 4-3 is a sample procedure for executing DSNTPSMP and DSNTBIND stored procedures that are used by the Development Center. The JCL needed for DSNTPSMP is included in <hlq>.SDSNSAMP(DSN8WLMP). Since both DSNTPSMP and DSNTBIND are REXX stored procedures, we set NUMTCB equal to 1.

Example 4-3 Our procedure for executing DSNTPSMP and DSNTBIND

```
//DB9AWLMR PROC DB2SSN=DB9A,NUMTCB=1,APPLENV=DB9AWLMR
//*
//NUMTCB@1 SET NUMTCB=                          <== NULL NUMTCB SYMBOL
//*
//DSNTPSMP EXEC PGM=DSNX9WLM,TIME=1440,
//             PARM='&DB2SSN,1,&APPLENV',       <== USE 1, NOT NUMTCB
//             REGION=0M,DYNAMNBR=5             <== ALLOW FOR DYN ALLOCS
//* INCLUDE SDSNEXIT TO USE SECONDARY AUTHIDS (DSN3@ATH DSN3@SGN EXITS)
//STEPLIB  DD  DISP=SHR,DSN=DB9A9.SDSNEXIT
//         DD  DISP=SHR,DSN=DB9A9.SDSNLOAD
//         DD  DISP=SHR,DSN=CBC.SCCNCMP      <== C COMPILER
//         DD  DISP=SHR,DSN=CEE.SCEERUN      <== LE RUNTIME
//SYSEXEC  DD  DISP=SHR,                       <== LOCATION OF DSNTPSMP
//             DSN=DB9A9.SDSNCLST
//SYSTSPRT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSABEND DD  DUMMY
//DSNTRACE DD  SYSOUT=*
//**** DATA SETS REQUIRED BY THE SQL PROCEDURES PROCESSOR
//SQLDBRM  DD  DISP=SHR,                       <== DBRM LIBRARY
//             DSN=DB9AU.DBRMLIB.DATA
//SQLCSRC  DD  DISP=SHR,                       <== GENERATED C SOURCE
//             DSN=DB9AU.SRCLIB.DATA
//SQLLMOD  DD  DISP=SHR,                       <== APPLICATION LOADLIB
//             DSN=DB9AU.RUNLIB.LOAD
//SQLLIBC  DD  DISP=SHR,                       <== C HEADER FILES
//             DSN=CEE.SCEEH.H
//         DD  DISP=SHR,
//             DSN=CEE.SCEEH.SYS.H
//         DD  DISP=SHR,                       <== DEBUG HEADER FILE
//             DSN=DB9A9.SDSNC.H
//SQLLIBL  DD  DISP=SHR,                       <== LINKEDIT INCLUDES
//             DSN=CEE.SCEELKED
//         DD  DISP=SHR,
//             DSN=DB9A9.SDSNLOAD
//SYSMSGS  DD  DISP=SHR,                       <== PRELINKER MSG FILE
//             DSN=CEE.SCEEMSGP(EDCPMSGE)
//*
//**** DSNTPSMP CONFIGURATION FILE - CFGTPSMP (OPTIONAL)
//*             A SITE PROVIDED SEQUENTIAL DATASET OR MEMBER, USED TO
//*             DEFINE CUSTOMIZED OPERATION OF DSNTPSMP IN THIS APPLENV.
//*CFGTPSMP DD  DISP=SHR,DSN=
//*
//**** WORKFILES REQUIRED BY THE SQL PROCEDURES PROCESSOR
//SQLSRC   DD  UNIT=SYSALLDA,SPACE=(23440,(20,20)),
```

```
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLPRINT DD  UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLTERM  DD  UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLOUT   DD  UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLCPRT  DD  UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLUT1   DD  UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLUT2   DD  UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLCIN   DD  UNIT=SYSALLDA,SPACE=(32000,(20,20))
//SQLLIN   DD  UNIT=SYSALLDA,SPACE=(3200,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLDUMMY DD  DUMMY
//SYSMOD   DD  UNIT=SYSALLDA,SPACE=(23440,(20,20)),   <= PRELINKER
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
```

Example 4-4 is a sample procedure for executing the user external SQL, COBOL, or C/C++ stored procedures. This user procedure for external SQL stored procedures needs one unauthorized data set included in STEPLIB.

Example 4-4 Sample user procedure for SQL, COBOL, C/C++ stored procedures

```
//*************************************************************
//*    JCL FOR RUNNING THE WLM-ESTABLISHED STORED PROCEDURES
//*    ADDRESS SPACE
//*       RGN     -- THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//*       DB2SSN  -- THE DB2 SUBSYSTEM NAME.
//*       NUMTCB  -- THE NUMBER OF TCBS USED TO PROCESS
//*                  END USER REQUESTS.
//*       APPLENV -- THE MVS WLM APPLICATION ENVIRONMENT
//*                  SUPPORTED BY THIS JCL PROCEDURE.
//*
//*************************************************************
//DB9AWLM  PROC RGN=0K,APPLENV=XXXXXXXX,DB2SSN=DB9A,NUMTCB=40
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=DB9AU.RUNLIB.LOAD
//         DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DISP=SHR,DSN=DB9A9.SDSNEXIT
//         DD  DISP=SHR,DSN=DB9A9.SDSNLOAD
```

Example 4-5 is a sample procedure for executing user Java stored procedures. Only the WLM Application Environment that executes Java stored procedures should include a //JAVAENV DD. The presence of this DD causes a JVM to be loaded, one for each NUMTCB. We set the NUMTCB to 1 for our test environment, so the refresh to the WLM environment went quickly while we were developing our code and making changes. We set NUMTCB to 5 in our production user procedure for Java stored procedures.

This user procedure for Java stored procedures needs one unauthorized data set included in STEPLIB.

Example 4-5 Sample procedure for user Java stored procedures

```
//*************************************************************
//*    JCL FOR RUNNING THE WLM-ESTABLISHED STORED PROCEDURES
```

```
//*    ADDRESS SPACE
//*       RGN     -- THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//*       DB2SSN  -- THE DB2 SUBSYSTEM NAME.
//*       NUMTCB  -- THE NUMBER OF TCBS USED TO PROCESS
//*                  END USER REQUESTS.
//*       APPLENV -- THE MVS WLM APPLICATION ENVIRONMENT
//*                  SUPPORTED BY THIS JCL PROCEDURE.
//*
//*************************************************************
//DB9AWLM  PROC RGN=0K,APPLENV=WLMENVJ,DB2SSN=DB9A,NUMTCB=5
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=DB9AU.RUNLIB.LOAD
//         DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DISP=SHR,DSN=DB9A9.SDSNEXIT
//         DD  DISP=SHR,DSN=DB9A9.SDSNLOAD
//         DD  DISP=SHR,DSN=DB9A9.SDSNLOD2
//JAVAENV DD DISP=SHR,DSN=DB9AU.JSPENV
//JSPDEBUG DD SYSOUT=*
```

5

Language Environment setup

In this chapter we provide an overview of Language Environment® for z/OS Version 1, Release 9, including the runtime options that impact stored procedure development.

This chapter contains the following:

- Language Environment concepts
- Language Environment runtime options
- Viewing RUNOPTS settings
- Language and compiler release level restrictions

5.1 Language Environment concepts

Language Environment (LE) is a component of the z/OS operating system. LE establishes a common runtime environment for different programming languages. It combines essential runtime services, such as routines for runtime message handling, condition handling, and storage management. All of these services are available through a set of interfaces that are consistent across programming languages. You may either call these interfaces yourself, or use language-specific services that call the interfaces. With Language Environment, you can use one runtime environment for your applications, regardless of the application's programming language or system resource needs.

DB2 uses LE to provide a runtime environment for stored procedures written in high-level languages such as COBOL, PL/I, and C. Stored procedures written in each of these languages can execute in the same stored procedure address space, though you may wish to separate them for various resource management reasons, as discussed in Chapter 4, "Setting up and managing Workload Manager" on page 39. Since multiple languages can share the same LE runtime library, you do not have to specify the language-specific libraries in the JCL procedure of each stored procedure address space.

A LE runtime library is required for WLM-managed stored procedure address spaces, and it must be the only runtime library available. You must not reference any other language runtime libraries within the system link list or within the joblib or steplib for the stored procedure started task. If other language runtime libraries are defined in the system link list, then you must use joblib or steplib overrides to exclude them. Old OS/VS COBOL, COBOL II, PL/I, and C runtime libraries (which are no longer supported by IBM) are not thread-safe and may cause logically inconsistent behavior if used in multi-threaded environments such as WLM-managed stored procedure address spaces. Depending upon the compile and link options that are used when your programs are prepared, some of those old runtime routines may be linkedited into your program load modules, and can cause inconsistencies at execution time.

LE performs several functions for DB2. It hides the differences between programming languages, provides the ability to make a stored procedure resident in the stored procedure address spaces, and supports a large number of runtime options, including those options needed to use tools to debug your stored procedures.

5.2 Language Environment runtime options

Each high level language (COBOL, PL/I, C) has a number of runtime options used to control the execution environment for applications written in each high level language. The default values for each of the LE runtime options is documented in *z/OS V1R9.0 Language Environment Customization*, SA22-7564-09. Note that you may already be overriding the z/OS default values for your LE runtime options with installation specific values for each language, especially if you are still supporting applications that run in AMODE 24.

DB2 stored procedures can use many of your existing values for LE runtime options, either the defaults provided with z/OS, or those values you overrode when you set up Language Environment. There are some options that you may wish to override for the purposes of improved debugging capabilities and better management of storage below the 16 MB line. Default values for LE runtime options can be overridden by specifying new values for the options on the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement. The following runtime options are those that are most frequently overridden for DB2 stored procedure.

5.2.1 MSGFILE

The MSGFILE runtime option specifies the ddname of the file that contains runtime diagnostics for the stored procedure. The format of the RUN OPTIONS parameter to specify a MSGFILE is as follows:

```
RUN OPTIONS 'MSGFILE(ddname,,,,ENQ | NOENQ)'
```

The purpose of the MSGFILE option is to provide a destination file for diagnostic messages. If a stored procedure also contains host language statements that display informational messages, then those messages will also be written to the MSGFILE data set. Care should be taken when managing diagnostics messages to avoid a situation where many stored procedures write to the same MSGFILE data set, otherwise, you could experience a JES spool serialization error, resulting in an S02A abend with reason C. There are two alternatives for managing message files to avoid serialization errors:

- Specify a different MSGFILE ddname for each common set of applications running in an application environment. For example, you could define message files using ddnames of SYSOUT1, SYSOUT2, etc., and direct messages from one set of applications to SYSOUT1, messages from a second set of applications to SYSOUT2, and so forth. Ideally, the MSGFILE data set should be used for diagnostics only, and you should not experience contention when your stored procedures are behaving well. If you do need to manage many diagnostic messages from many stored procedures, then maintaining multiple message files may be a challenge. In that situation you may prefer the second alternative.
- An alternative to maintaining multiple MSGFILE data sets is to maintain one MSGFILE data set and specify the ENQ sub-option in your MSGFILE runtime option on your stored procedure DDL. Specifying the ENQ sub-option resolves the JES spool serialization error without requiring you to maintain multiple MSGFILE data sets. See “Chapter 6” of *z/OS V1R9.0 Language Environment Customization*, SA22-7564-09 for details on when the ENQ option should be used.

5.2.2 RPTOPTS

The RPTOPTS runtime option generates a report of the runtime options in effect while the application was running. The report is directed to the ddname specified in the MSGFILE runtime option. This information can be useful when debugging a stored procedure, because the resulting report will display the default options specified in the LE environment as well as any options that have been overridden at the stored procedure level. The format of the RUN OPTIONS parameter to specify RPTOPTS is as follows:

```
RUN OPTIONS 'RPTOPTS(ON)'
```

Specify RPTOPTS only when you need to understand what runtime options are in effect while testing a stored procedure. You should ensure that RPTOPTS is set to OFF when running in production as the report generation process increases the time it takes to run the stored procedure.

5.2.3 TEST and NOTEST

If you wish to run your stored procedure through a debugging tool, such as the IBM Data Studio, you need to specify the LE runtime option TEST. The default for this option in LE is NOTEST, which means that no debugging information is generated while the procedure is running. To run your stored procedure through the distributed debugger available with the DB2 Development Center, you need to specify the IP address and listening port for your test environment. In our test case we had an IP address of 9.112.68.25 and a listening port of

8000. The RUN OPTIONS parameters we used to set up our stored procedures for testing with the distributed debugger were as follows:

```
RUN OPTIONS 'TEST(,,,VADTCPIP&9.1.39.26%8000:*)'
```

The information other than the IP address and the listening port is fixed, and should not be changed. See Chapter 28, “Tools for debugging DB2 stored procedures” on page 735 for more details on debugging stored procedures with DB2 Development Center.

You can also debug stored procedures that run on z/OS by using the 3270 MVS MFI VTAM® option. This feature allows you to run a debugging session on the mainframe for your z/OS stored procedures. The TEST options for MFI debugging are different than for Development Center debugging. See Chapter 28, “Tools for debugging DB2 stored procedures” on page 735 for more details.

5.2.4 Options to limit storage required by LE at execution time

There are a number of LE runtime options that you can specify to minimize storage usage below the 16 MB line. They are documented in “Chapter 25, Using stored procedures for client/server processing” of *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841. We repeat them here for your reference:

- HEAP(,,ANY) to allocate program heap storage above the 16 MB line
- STACK(,,ANY,) to allocate program stack storage above the 16 MB line
- STORAGE(,,,4K) to reduce reserve storage area below the line to 4 KB
- BELOWHEAP(4K,,) to reduce the heap storage below the line to 4 KB
- LIBSTACK(4K,,) to reduce the library stack below the line to 4 KB
- ALL31(ON) to indicate all programs contained in the stored procedure run with AMODE(31) and RMODE(ANY).

5.3 Viewing RUNOPTS settings

The RUN OPTIONS for a DB2 stored procedure are set when a CREATE PROCEDURE or ALTER PROCEDURE statement is executed. The options specified in the DDL are stored in the system catalog in table SYSIBM.SYSROUTINES in the column RUNOPTS. An empty string in the RUNOPTS column means that the z/OS default values or installation-supplied values for that language are used for the LE runtime options for the procedure. Figure 5-1 shows the information in the RUNOPTS column for some of our sample stored procedures.

```
SCHEMA    NAME      RUNOPTS
--------  --------  ------------------------------------------------
DEVL7083  EMPDTLSC
DEVL7083  EMPDTLSJ
DEVL7083  EMPDTLSR  TRAP(OFF),RPTOPTS(OFF)
DEVL7083  EMPDTLSS  TRAP(OFF),RPTOPTS(OFF)
DEVL7083  EMPRSETC  TRAP(OFF),RPTOPTS(OFF)
DEVL7083  EMPRSETJ
DEVL7083  EMPRSETR  TRAP(OFF),RPTOPTS(OFF)
DEVL7083  EMPRSETS  TRAP(OFF),RPTOPTS(OFF)
DEVL7083  PRGTYPE1  MSGFILE(SYSOUT,,,,ENQ),TEST(,,,MFI%SC63DT12:*)
DEVL7083  PRGTYPE2  MSGFILE(SYSDB,,,,ENQ)
```

Figure 5-1 runtime options shown in SYSIBM.SYSROUTINES

Note that stored procedures EMPDTLSC, EMPDTLSJ, and EMPRSETJ will use the installation default values for the LE runtime options, while the remainder of the stored procedures will use the defaults plus the additional options shown in the report.

5.4 Language and compiler release level restrictions

If your stored procedures are executing existing production programs or calling existing production sub-programs, be aware that older language compilers have restrictions that may not exist with newer compilers. For example, modules last compiled with the OS/VS COBOL compiler are serially reusable, but not reentrant. Similarly, modules last compiled with COBOL II without the RENT option are also serially reusable, but not reentrant. See the appropriate language's migration guide regarding restrictions or limitations imposed by each language and compiler release level.

If your stored procedures will be executing Assembler language code, they must be LE compliant, meaning they must comply with LE rules, but need not be LE conforming, meaning they need not take advantage of LE features. LE is aware of resources allocated through LE services, and will free them at the appropriate times. LE is not aware of resources allocated through non-LE services, and cannot free them at the appropriate times. For example, an Assembler routine that does a GETMAIN must always do its own FREEMAIN. However, memory acquired through LE's callable memory allocation routine is automatically freed.

If your stored procedures will be executing third party software, it may be difficult to determine which routines are multi-thread safe and which are not. You may have to add your own serialization method around those routines.

6

RRSAF

In this chapter we provide a brief overview of the Resource Recovery Services Attach Facility (RRSAF), a description of how it is used by DB2 for WLM stored procedures, and a list of the steps required to implement RRSAF.

This chapter contains the following:

- RRSAF overview
- RRSAF and DB2 stored procedures
- Implementing RRSAF

6.1 RRSAF overview

The vast majority of a company's computer resources are so critical to that company's business that the integrity of these resources must be guaranteed. If changes to the data in the resources are corrupted by a hardware or software failure, human error, or a catastrophe, the computer must be able to restore the data. These critical resources are called *protected resources* or, sometimes, *recoverable resources*. The data in DB2 tables is one type of resource that falls into this category.

Resource recovery is the protection of these critical resources. Resource recovery consists of the protocols and program interfaces that allow an application program, such as a DB2 stored procedure, to make consistent changes to multiple protected resources of different types, such as DB2 data and IMS data. z/OS, when requested, can coordinate changes to one or more protected resources, which can be accessed through different resource managers and reside on different systems. z/OS ensures that all changes are made or no changes are made. In other words, all data is committed or all data is rolled back. Resources that z/OS can protect include:

- A hierarchical database, such as IMS
- A relational database, such as DB2
- A product-specific resource

There are three types of programs that work together to protect resources within a z/OS environment:

- **Application program** - The application program accesses protected resources and requests changes to the resources. For our purposes the application is a DB2 stored procedure.
- **Resource Manager** - A resource manager controls and manages access to a resource. A resource manager is an authorized program that provides an application programming interface (API) that allows the application program to read and change a protected resource. The resource manager, through exit routines that get control in response to events, takes actions that commit or back out changes to a resource it manages. Often an application changes more than one protected resource, so that more than one resource manager is involved. A resource manager may be an IBM product, such as DB2 or IMS, part of an IBM product, or a product from another vendor.
- **Syncpoint manager** - Resource Recovery Services (RRS) is the syncpoint manager program. It uses a two-phase commit protocol to coordinate changes to protected resources, so that all changes are made or no changes are made. During its processing, RRS drives exit routines for each resource manager. For example, if a DB2 application issues a commit, RRS drives the commit exit routine for each resource manager involved. If the DB2 application is executing under CICS, then RRS drives the commit exit routine for both DB2 and CICS.

RRSAF works in conjunction with the application program, the resource manager and the syncpoint manager to ensure that updates to DB2 resources and other protected resources are synchronized across a unit of work. Either all work is committed, or all work is backed out.

6.2 RRSAF and DB2 stored procedures

Resource Recovery Services (RRS) is important to DB2 because it coordinates two-phase commit processing of recoverable resources in a z/OS system. RRSAF is required for stored procedures that run in a WLM-established address space. You can write RRSAF applications in any of the high level languages supported by Language Environment.

Preparing an application to run in RRSAF is similar to preparing it to run in other environments such as CICS, IMS, or TSO, except that WLM-established stored procedures have to be linkedited with the DSNRLI language interface module. You can prepare an RRSAF application by executing program preparation JCL in batch, or by using the DB2 program preparation panels. For more details on preparing a DB2 program to run in RRSAF see, “Chapter 32” of *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

There are no special coding techniques that you need to follow to invoke RRSAF within a stored procedure once you have prepared your stored procedure using the DSNRLI language interface module. DSNRLI takes care of executing the appropriate exit routines to manage the two-phase commit processing. DB2 Version 7 was the last version of DB2 to support definition of DB2-established stored procedure address spaces. With DB2 Version 9, all new stored procedures must be defined in a WLM-established address space; therefore, all stored procedures need to be linkedited with DSNRLI.

6.3 Implementing RRSAF

To use WLM-established stored procedure address spaces you have to implement Resource Recovery Services (RRS). This section shows the steps required for implementing RRS and the jobs that were used for the RRS environment for our test cases.

DB2 requires that RRS be active, because WLM-established stored procedure address spaces use the RRS attachment facility (RRSAF), not the call attachment facility (CAF) which was used for DB2-established stored procedure address space. You cannot use the CAF in WLM-established stored procedure address spaces.

For documentation purposes the following message is included in the MSTR address space upon restart of your DB2 subsystem:

```
DSN3029I  -DB9A DSN3RRSR RRS ATTACH PROCESSING IS AVAILABLE
```

You cannot explicitly code any call to DSNRLI for WLM-established address spaces.

RRS is an MVS system logger application that records events related to protected resources. RRS records these events in five log streams. In a sysplex environment, these log streams are shared by the systems of the sysplex. Before you can start RRS, you must:

1. Define RRS’s log streams. The log streams can be placed on disk or in the Coupling Facility. In our test case, the log streams were placed in the Coupling Facility. To do this, you must:
   - Add definitions for the structure in the CFRM policy.
   - Define the log streams.
   - Activate the new definitions.
2. Establish the priority for the RRS address space.
3. Set up the RRS procedure in SYS1.PROCLIB.
4. Define the RRS subsystem to MVS.

6.3.1 RRS log streams

To set up your log streams, refer to “Preparing to Use System Logger Applications,” in *z/OS V1R9.0 MVS Setting Up a Sysplex*, SA22-7625-14, and “Understanding RRS Logging Requirements” in *z/OS V1R9.0 MVS Programming: Resource Recovery*, SA22-7616-07.

The five log stream names used by RRS are (where *gname* can be your sysplex name or any name in a non-sysplex environment):

- Main unit-of-recovery log state stream:

  ATR.*gname*.MAIN.UR´

  The state of active URs. RRS periodically moves this information into the RRS delayed UR state log when UR completion is delayed.

- Delayed unit-of-recovery log state stream:

  ATR.*gname*.DELAYED.UR

  The state of active URs, when UR completion is delayed.

- Resource manager data log stream:

  ATR.*gname*.RM.DATA

  Information about the resource managers using RRS services.

- Restart log stream:

  ATR.*gname*.RESTART

  Information about incomplete URs needed during restart. This information enables a functioning RRS instance to take over incomplete work left over from an RRS instance that failed.

- Archive log stream (this log is recommended but optional):

  ATR.*gname*.ARCHIVE

  Information about completed URs.

To define the RRS log streams, use IXCMIAPU, which is a utility program provided in the SYS1.MIGLIB system library.

Defining the RRS log streams to use the Coupling Facility

If the RRS log streams use the Coupling Facility, you have to update the CFRM policy to add the RRS structures. Example 6-1 shows the JCL we used to update the CFRM policy.

Example 6-1 Job to update CFRM policy

```
//DEFCFRM1 JOB MSGCLASS=X,TIME=10,MSGLEVEL=(1,1),NOTIFY=&SYSUID
//STEP1 EXEC PGM=IXCMIAPU
//SYSPRINT DD SYSOUT=*
//SYSABEND DD SYSOUT=*
//SYSIN DD *
    DATA TYPE(CFRM) REPORT(YES)
    DEFINE POLICY NAME(CFRM18) REPLACE(YES)
    CF NAME(CF01)
        TYPE(009672)
        MFG(IBM)
        PLANT(02)
        SEQUENCE(000000040104)
        PARTITION(1)
        CPCID(00)
        DUMPSPACE(2048)
    CF NAME(CF02)
        TYPE(009672)
        MFG(IBM)
        PLANT(02)
.................
.................
```

```
.................
STRUCTURE NAME(RRS_ARCHIVE_1)
               INITSIZE(8000)
               SIZE(16000)
               PREFLIST(CF1,CF2)
               REBUILDPERCENT(5)
        STRUCTURE NAME(RRS_RMDATA_1)
               INITSIZE(8000)
               SIZE(16000)
               PREFLIST(CF1,CF2)
               REBUILDPERCENT(5)
        STRUCTURE NAME(RRS_MAINUR_1)
               INITSIZE(8000)
               SIZE(16000)
               PREFLIST(CF1,CF2)
               REBUILDPERCENT(5)
        STRUCTURE NAME(RRS_DELAYEDUR_1)
               INITSIZE(8000)
               SIZE(16000)
               PREFLIST(CF1,CF2)
               REBUILDPERCENT(5)
        STRUCTURE NAME(RRS_RESTART_1)
               INITSIZE(8000)
               SIZE(16000)
               PREFLIST(CF1,CF2)
               REBUILDPERCENT(5)
```

Note that:

- *gname* can be any name of your choice. In our test case we used SANDBOX. When you start RRS, you must specify for the gname parameter of the JCL procedure the same gname specified when you created your log streams. If you do not specify the name when starting RRS, the default is the sysplex name.
- *vsamls* is an SMS class defined for linear VSAM files. You can set up a new SMS class, or use an existing SMS class for VSAM linear data sets.

  To verify the data classes already defined in SMS, you can invoke the SMS ISPF application, choose **option 4**, and list all defined SMS classes.

  The log stream (LS) VSAM data sets will be allocated at the time the RRS log streams are defined. Each data set is prefixed with IXGLOGR and suffixed with A0000000. They are named as follows:

  ```
  IXGLOGR.ATR.gname.ARCHIVE.A0000000
  IXGLOGR.ATR.gname.ARCHIVE.A0000000.DATA
  ```

  The staging (STG) VSAM data sets are allocated at RRS startup. When RRS is canceled, it deletes the STG data sets. Each data set is prefixed with IXGLOGR, and suffixed with the Sysplex name. They are named as follows:

  ```
  IXGLOGR.ATR.gname.ARCHIVE.Sysplexn
  IXGLOGR.ATR.gname.ARCHIVE.Sysplexn.DATA
  ```

You can map each log stream to a single structure or you can map log streams of like data types to the same structure. Example 6-2 shows the JCL to map each RRS log stream to a structure.

Example 6-2 Job for mapping RRS log stream to a structure

```
//STEP1 EXEC PGM=IXCMIAPU
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
```

```
DATA TYPE(LOGR) REPORT(YES)

  DEFINE STRUCTURE NAME(RRS_ARCHIVE_1) LOGSNUM(1)
         MAXBUFSIZE(64000) AVGBUFSIZE(262)

  DEFINE STRUCTURE NAME(RRS_RMDATA_1) LOGSNUM(1)
         MAXBUFSIZE(1024) AVGBUFSIZE(252)

  DEFINE STRUCTURE NAME(RRS_MAINUR_1) LOGSNUM(1)
         MAXBUFSIZE(64000) AVGBUFSIZE(158)

  DEFINE STRUCTURE NAME(RRS_DELAYEDUR_1) LOGSNUM(1)
         MAXBUFSIZE(64000) AVGBUFSIZE(158)

  DEFINE STRUCTURE NAME(RRS_RESTART_1) LOGSNUM(1)
         MAXBUFSIZE(64000) AVGBUFSIZE(158)

  DEFINE LOGSTREAM
  NAME(ATR.SANDBOX.ARCHIVE) STRUCTNAME(RRS_ARCHIVE_1)
  LS_DATACLAS(SHARE33)
  HLQ(LOGR) MODEL(NO) LS_SIZE(1024)
  LOWOFFLOAD(0) HIGHOFFLOAD(80) STG_DUPLEX(NO)
  RETPD(15) AUTODELETE(YES)

  DEFINE LOGSTREAM
  NAME(ATR.SANDBOX.RM.DATA) STRUCTNAME(RRS_RMDATA_1)
  LS_DATACLAS(SHARE33)
  HLQ(LOGR) MODEL(NO) LS_SIZE(1024)
  LOWOFFLOAD(0) HIGHOFFLOAD(80) STG_DUPLEX(NO)
  RETPD(15) AUTODELETE(YES)

  DEFINE LOGSTREAM
  NAME(ATR.SANDBOX.MAIN.UR) STRUCTNAME(RRS_MAINUR_1)
  LS_DATACLAS(SHARE33)
  HLQ(LOGR) MODEL(NO) LS_SIZE(1024)
  LOWOFFLOAD(0) HIGHOFFLOAD(80) STG_DUPLEX(NO)
  RETPD(15) AUTODELETE(YES)

  DEFINE LOGSTREAM
  NAME(ATR.SANDBOX.DELAYED.UR) STRUCTNAME(RRS_DELAYEDUR_1)
  LS_DATACLAS(SHARE33)
  HLQ(LOGR) MODEL(NO) LS_SIZE(1024)
  LOWOFFLOAD(0) HIGHOFFLOAD(80) STG_DUPLEX(NO)
  RETPD(15) AUTODELETE(YES)

  DEFINE LOGSTREAM
  NAME(ATR.SANDBOX.RESTART) STRUCTNAME(RRS_RESTART_1)
  LS_DATACLAS(SHARE33)
  HLQ(LOGR) MODEL(NO) LS_SIZE(1024)
  LOWOFFLOAD(0) HIGHOFFLOAD(80) STG_DUPLEX(NO)
  RETPD(15) AUTODELETE(YES)
  /*
```

If you need to delete the log streams and the structures from the Coupling Facility, you can use the JCL in Example 6-3 as a model for your JCL.

Example 6-3 Job for deleting log streams and structures

```
  //STEP1 EXEC PGM=IXCMIAPU
  //SYSPRINT DD SYSOUT=*
```

```
  //SYSIN DD *
DATA TYPE(LOGR) REPORT(YES)
    DELETE LOGSTREAM
    NAME(ATR.SANDBOX.ARCHIVE)
    DELETE LOGSTREAM
    NAME(ATR.SANDBOX.RM.DATA)
    DELETE LOGSTREAM
    NAME(ATR.SANDBOX.MAIN.UR)
    DELETE LOGSTREAM
    NAME(ATR.SANDBOX.DELAYED.UR)
    DELETE LOGSTREAM
    NAME(ATR.SANDBOX.RESTART)
  /*
```

6.3.2 Activating the CFRM policy to support RRS

If your log streams use the Coupling Facility, you have to activate the updated CFRM policy. To change the CFRM policy, you have to compile and linkedit the policy. Then you have to activate this new CFRM policy in your sysplex. You can activate the updated CFRM policy with the following operator command:

```
SETXCF START,POLICY,TYPE=CFRM,POLNAME=polname
```

6.3.3 Making the RRS JCL procedure available

You have to move the ATRRRS procedure from the SYS1.SAMPLIB to your SYS1.PROCLIB as member RRS. If you use a different name for the procedure, the first four characters of the procedure name must match the subsystem name as registered in the IEFSSNxx member of SYS1.PARMLIB. A sample of the JCL procedure we used to start RRS is shown in Example 6-4.

Example 6-4 Procedure for starting RRS

```
//RRS      PROC GNAME='',CTMEM=''
//RRS      EXEC PGM=ATRIMIKE,REGION=0M,TIME=NOLIMIT,
//           PARM='GNAME=&GNAME CTMEM=&CTMEM'
//
```

The GNAME must match the gname specified when defining the log streams.

6.3.4 Adding RRS subsystem name

To activate the RRS application, you have to define RRS as a subsystem to MVS. To add the RRS subsystem name to MVS, you must find out with your system programmer the active IEFSSNxx member of SYS1.PARMLIB. Edit the member to include the following entry:

```
SUBSYS SUBNAME(RRS)         /* RESOURCE RECOVERY SERVICES */
```

The subsystem name can be RRS or any other name of your choice. Note that the first characters (up to four) of the JCL procedure name to start RRS must match the subsystem name.

6.3.5 Starting and stopping RRS

Once you have set address space priority, provided the statement in IEFSSNxx, and know that the system logger is active, you can start RRS with the following operator command:

```
START RRS
```

You can stop RRS with the following operator command:

```
SETRRS CANCEL
or
SETRRS SHUTDOWN
```

If SETRRS CANCEL or SETRRS SHUTDOWN does not stop RRS, you can use the FORCE RRS,ARM command. In this command, RRS is the subsystem name your installation assigned to RRS in PARMLIB member IEFSSNxx.

Here are the messages you receive when you issue this command:

```
SETRRS CANCEL
...
ATR101I CANCEL REQUEST WAS RECEIVED FOR RRS.
ATR143I RRS HAS BEEN DEREGISTERED FROM ARM.
...
ASA2960I RRS SUBSYSTEM FUNCTIONS DISABLED. COMPONENT ID=SCRRS
ATR167I RRS RESMGR PROCESSING COMPLETED.
```

6.3.6 RRS error samples

In this section, we examine some errors that you may encounter and the possible causes.

- If RRS cannot find one of the log streams, you get the following when starting RRS:

```
     IEF403I RRS - STARTED - TIME=20.49.38
     ATR221I RRS IS JOINING RRS GROUP gname ON SYSTEM SC53
     ATR130I RRS LOGSTREAM CONNECT HAS FAILED FOR 496
     MANDATORY LOGSTREAM ATR.gname.RM.DATA.
     RC=00000008, RSN=0000080B
IEA989I SLIP TRAP ID=X13E MATCHED. JOBNAME=RRS , ASID=0068.
IXG231I IXGCONN REQUEST=CONNECT TO LOG STREAM ATR.gname.RM.DATA
DID 495
NOT SUCCEED FOR JOB RRS. RETURN CODE: 00000008 REASON CODE: 0000080B
DIAG1: 00000008 DIAG2: 0000F801 DIAG3: 05030004 DIAG4: 05020010
ASA2013I RRS INITIALIZATION FAILED. COMPONENT ID=SCRRS
```

  Action: Verify that the define log stream job ran correctly.

- Starting sample procedure member ATRRRS with the MVS subsystem name of RRS, you receive the following error message:

```
S ATRRRS,SUB=MSTR
.................
.................
IEF695I START ATRRRS WITH JOBNAME ATRRRS IS ASSIGNED TO USER STC
    , GROUP SYS1
IEF403I ATRRRS - STARTED - TIME=14.19.57
ASA2016I ATRR IS NOT A VALID SUBSYSTEM. COMPONENT ID=SCRRS
ASA2013I ATRR INITIALIZATION FAILED. COMPONENT ID=SCRRS
```

  Action: Rename procedure member name from ATRRRS to RRS (or a name that matches your subsystem name) and restart RRS.

6.4 DB2 restart and recovery with RRS

On restart DB2, like any other DBMS, performs extensive restart and recovery processing. The DB2 for z/OS Administration Guide discusses DB2 restart and recovery processing in detail. Here, we concentrate on how DB2 interacts with RRS on restart and recovery, and we

look at scenarios that may force manual intervention to resolve in-doubt URs with RRS. For detailed information, see *Systems Programmer's Guide to Resource Recovery Services (RRS),* SG24-6980.

The normal restart and recovery phases that DB2 goes through are:

- Phase 1: Log initialization

  DB2 identifies the last LOG RBA used before termination so that it can start logging at the next RBA.

- Phase 2: Current status rebuild.

  DB2 determines what URs are outstanding and the status of each UR (in-flight, in-commit, in-abort or in-doubt). DB2 also recovers information about coordinator and participants for all outstanding URs.

- Phase 3: Forward Log Recovery

  Having determined the outstanding URs in Phase 2, DB2 makes all the database changes for committed work as well as for in-flight, in-doubt and in-abort URs. For in-flight, in-doubt and in-abort URs, DB2 locks the changed data to make it unavailable.

- Phase 4: Backward Log Recovery

  In this phase, DB2 reverses out changes by in-flight or in-abort URs and releases locks for those URs.

During Phase 2 of DB2 restart processing, DB2 retrieves any outstanding URs from RRS.

6.4.1 DB2 restart if RRS is unavailable

If DB2 restarts and RRS is not available, then it cannot resolve any URs that were in-doubt where RRS was the syncpoint manager and DB2 was a participant. For any URs in which DB2 was coordinator (DB2 WLM-managed stored procedures, for example) or in which the state was in-flight, in-commit, in-abort or in postponed abort, DB2 does not need to retrieve UR data from RRS. It can process those URs by backing out any in-flight or in-abort URs.

Any in-doubt URs may result in retained locks for affected data. If there are in-doubt URs, DB2 will issue messages DSN3010I or DSN3011I to indicate that RRS cannot be contacted to resolve these in-doubt URs.

DB2 will resync with RRS once RRS is restarted on the z/OS image. At that stage it can resolve any in-doubt URs according to the final state that RRS supplies.

Note that when DB2 starts and RRS is unavailable, any DB2 facilities that require RRS will also be unavailable. That means any attempt to use RRSAF or WLM managed stored procedures will result in an error. For example, an attempt to start a WLM stored procedure results in the error message shown in Example 6-5.

Example 6-5 WLM stored procedure sample startup messages

```
DSNX982I  DSNX9WLS ATTEMPT TO PERFORM RRS ATTACH FUNCTION  963
SPAS_INIT_SP       FAILED WITH RRS RC = 00000008 RSN = 00F30052 SSN =
DB9A     PROC= DB9AWLM  ASID = 0097 WLM_ENV = DB9AWLM
```

Once RRS restart processing is complete, a message is displays in the DB2 MSTR log; see Example 6-6 on page 62.

Example 6-6 DB2 message after RRS start/restart

```
DSN3029I  -DB9A DSN3RRRS RRS ATTACH PROCESSING IS AVAILABLE
```

6.4.2 Navigate the RRS ISPF panels

We show how to relate the information in the RRS ISPF application with information available in DB2.

Select **option 2** on the RRS option menu; see Example 6-7.

Example 6-7 RRS option menu

```
                                    RRS
 Option ===> 2

 Select an option and press ENTER:

 1 Browse an RRS log stream
 2 Display/Update RRS related Resource Manager information
 3 Display/Update RRS Unit of Recovery information
 4 Display/Update RRS related Work Manager information
 5 Display/Update RRS UR selection criteria profiles
 6 Display RRS-related system information
```

We look at the status of the DB2 resource managers by looking at the RRS ISPF resource manager list panel; see Example 6-8.

Example 6-8 RRS resource manager list

```
RRS Resource Manager List          Row 1 to 10 of 1
 Command ===>                                          Scroll ===> PAGE

 Commands: v-View Details u-View URs r-Remove Interest d-Delete RM
           n-Unregister RM

 S   RM Name                        State            System   Logging Group
     DSN.RRSATF.IBM.DB7G            Reset            SC63     SANDBOX
     DSN.RRSATF.IBM.DB8A            Run              SC63     SANDBOX
     DSN.RRSATF.IBM.DB9A            Run              SC63     SANDBOX
     DSN.RRSATF.IBM.DB9B            Reset            SC63     SANDBOX
     DSN.RRSATF.IBM.D8F1            Run              SC63     SANDBOX
     DSN.RRSPAS.IBM.DB7G            Reset            SC63     SANDBOX
     DSN.RRSPAS.IBM.DB8A            Run              SC63     SANDBOX
     DSN.RRSPAS.IBM.DB9A            Run              SC63     SANDBOX
     DSN.RRSPAS.IBM.DB9B            Reset            SC63     SANDBOX
     DSN.RRSPAS.IBM.D8F1            Run              SC63     SANDBOX
```

We are interested in resource manager `DSN.RRSATF.IBM.DB9A`. The RRS state is `Run` which means that DB9A has registered to RRS on system SC63 and has completed restart processing. We ran batch job PAOLOR15 that executes program RRSDTL1C that uses the RRSAF attachment and calls stored procedure EMPDTL1C

Type `u` next to `DSN.RRSATF.IBM.DB9A` to enter the UR detail view; see Example 6-9 on page 63.

Example 6-9 UR detail view

```
RRS Unit of Recovery Details
Command ===>

Commands r-Remove Interest v-View URI Details

UR identifier : C180BF5D7E0860000000001601040000
Create time : 2007/11/15 18:16:08.672508  GMT  Comments :
UR state : InFlight      UR type : Unpr
System : SC63      Logging Group : SANDBOX
SURID : N/A
Work Manager Name : SC63.PAOLOR15.0038
   Display Work IDs             Display IDs formatted
   Luwid  . : Not Present
   Eid  . . : Not Present
   Xid  . . : Not Present
Expressions of Interest:
S   RM Name                           Type  Role
    DSN.RRSATF.IBM.DB9A               Unpr  Participant
```

We can relate the RRS URid to a DB2 thread by means of the DB2 command DISPLAY THREAD(*) RRSURID(*); see Example 6-10.

Example 6-10 DISPLAY THREAD(*) RRSURID(*) output

```
DSNV401I  -DB9A DISPLAY THREAD REPORT FOLLOWS -
DSNV402I  -DB9A ACTIVE THREADS -
NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
RRSAF    SP *     5              PAOLOR1  RRSDTL1C 0038    20
 V480-DB2 IS PARTICIPANT FOR RRS URID=C180BF5D7E0860000000001601040000
 V429 CALLING PROCEDURE=DEVL7083.EMPDTL1C,
      PROC=DB9AWLM , ASID=0092, WLM_ENV=DB9AWLM
DISPLAY ACTIVE REPORT COMPLETE
DSN9022I  -DB9A DSNVDT '-DISPLAY THREAD' NORMAL COMPLETION
```

Notice the value for RRS URID.

7

Security and authorization

Stored procedures are DB2 objects that are maintained by DB2 just like other objects such as tables, views, packages, and plans. Like any other DB2 object, access to stored procedures is controlled by the privileges that have been granted to the authorization IDs that are requesting access.

In this chapter we describe the privileges required for creating and executing DB2 stored procedures, along with the security administration tasks to set up those privileges. For details on security requirements to use IBM Data Studio, see “IBM Data Studio authorization setup” on page 651.

This chapter contains the following:

- Workload Manager security requirements
- Privileges required to create stored procedures
- Privileges required to execute stored procedures
- Network trusted context and roles
- Additional stored procedure security considerations

See the Security section of *DB2 Version 9.1 for z/OS Administration Guide,* SC18-9840 for more information on these topics.

7.1 Workload Manager security requirements

DB2 stored procedures that run on z/OS can run in either the single DB2-established stored procedure address space that became available with Version 4 of DB2, or in any of a number of WLM-managed address spaces that you may define. Since DB2 Version 8 no longer supports the creation of *new* stored procedures in the DB2-established address space, we focus our attention in this chapter only on the security requirements for stored procedures that run in WLM-managed address spaces.

There are two external levels of security that have to be set up for WLM stored procedures:

- Each authid that attempts to create a stored procedure needs the authority to create stored procedures in the WLM environment where the procedure will be run.
- Application developers or DBAs that need to refresh WLM application environments must be permitted access to the DB2-supplied stored procedure WLM_REFRESH to refresh the address spaces.

7.1.1 Controlling access to WLM

This is an optional step, which we did not implement, for controlling which address spaces can be WLM-established server address spaces that run stored procedures. Otherwise, any address space can connect to WLM and run stored procedures.

- You need to use the server resource class and define a class named SERVER with the command:

```
RDEFINE SERVER (DB2.DB9A.WLMENV)
```

- You then authorize the profile that you want to associate with the server:

```
RDEFINE SERVER (DB2.DB9A.DB9AWLM)
```

- Activate the resource class:

```
SETROPTS RACLIST(SERVER)REFRESH
```

- Permit read access to the user IDs associated with the stored procedure address space:

```
PERMIT DB2.DB9A.DB9AWLM CLASS(SERVER) ID(SYSDSP) ACCESS(READ)
```

7.1.2 Controlling creation of stored procedures in WLM environments

When you define a WLM environment, you need to issue some RACF commands to prevent all users from creating stored procedures in that environment. Otherwise, you would not be able to prevent application developers from creating stored procedures in production application environments. The RACF command we used to protect WLM application environment DB9AWLM on subsystem DB9A is:

```
RDEFINE DSNR (DB9A.WLMENV.DB9AWLM) UACC(NONE)
```

Issuing this command ensures that universal access is NONE on the application environment. To allow individual developers or groups access to the application environment we issue the following command, which permits users in RACF group DEVL7083 to create stored procedures in address space DB9AWLM:

```
PERMIT DB9A.WLMENV.DB9AWLM CLASS(DSNR) ID(DEVL7083) ACCESS(READ)
```

In case of data sharing, the first node can be the group ID of the data sharing group.

7.1.3 Permitting access to WLM REFRESH command

When you prepare a new version of a stored procedure in a WLM application environment, you need to refresh the application environment to activate the new version of the program. You do this by issuing a VARY REFRESH command, which can be done on a z/OS command line, or by executing the DB2-supplied WLM_REFRESH stored procedure. We issued the following command to refresh application environment DB9AWLM, which contained the majority of the COBOL stored procedures in our test cases:

```
/V WLM,APPLENV=DB9AWLM,REFRESH
```

Alternatively, we could have executed DB2-supplied stored procedure WLM_REFRESH, which executes the REFRESH command for us. Since most developers do not have the authority to issue operator commands, we recommend that you use the WLM_REFRESH procedure. There are two steps needed to permit developers to use the WLM_REFRESH stored procedure.

First you must permit access to the WLM_REFRESH RACF resource profile for each application environment. The RACF RDEFINE command to permit RACF group DEVL7083 access to the WLM_REFRESH resource profile for application environment DB9AWLM on subsystem DB9A is shown in Example 7-1.

Example 7-1 Permit access to WLM_REFRESH resource profile

```
RDEFINE DSNR (DB9A.WLM_REFRESH.DB9AWLM)
PE DB9A.WLM_REFRESH.DB9AWLM +
   CLASS(DSNR) ID(DEVL7083) ACCESS(READ)
END
```

After issuing the above RDEFINE command for each environment for which you need to refresh, you then need to grant EXECUTE authority on the WLM_REFRESH stored procedure to the authids or groups who will be refreshing the environment. You only need to grant EXECUTE authority once since you supply the application environment name as a variable when you execute WLM_REFRESH. A sample GRANT statement for WLM_REFRESH is as follows:

```
GRANT EXECUTE ON PROCEDURE SYSPROC.WLM_REFRESH TO DEVL7083;
```

The DB2-supplied installation verification job DSNTEJ6W contains the steps to create the resource profile for refreshing WLM, and to prepare the WLM_REFRESH stored procedure. See Chapter 13, “Verifying installation with the sample applications” in *DB2 Version 9.1 for z/OS Installation Guide,* SC18-9846 for details on job DSNTEJ6W. See A.2, “Refresh a WLM environment with AdminWLMRefresh” on page 817 in this book for more details on WLM_REFRESH.

7.2 Privileges required to create stored procedures

Stored procedures are typically created by a DBA or application programmer, depending on how roles and responsibilities are defined for your organization. The DBA or application programmer requires certain DB2 privileges in order to create stored procedures. In this section we describe the statements used to grant those privileges, along with errors you may receive while attempting to create a procedure when you do not have the appropriate authorization. In our case study we created authid PAOLORW with no DB2 privileges in order to demonstrate the SQL statements required for PAOLORW to create stored procedures, and to document the error messages received when those privileges do not exist.

7.2.1 CREATEIN privilege on the schema

When a stored procedure is created, it is implicitly or explicitly qualified by a schema. A schema is a collection of named objects such as stored procedures, triggers, and user-defined functions. When a stored procedure is created, it is given a three-part name. The first part is the location, or DB2 subsystem, where the stored procedure is defined. The location can be implicitly or explicitly specified. If the location is left blank, it defaults to the subsystem on which the CREATE PROCEDURE statement is issued.

The second part of the name is the schema name, which also can be implicitly or explicitly specified. If the schema is left blank, it defaults to the value in CURRENT SCHEMA (assuming dynamic SQL) for the person issuing the CREATE PROCEDURE statement.

Most stored procedures are created into one or more common schemas that are defined at an application level. For our case study, we used schema DEVL7083 for all stored procedures created in our development environment, while we used schema PROD7083 for our production environment. The following SQL statement was issued during our case study to allow authid PAOLORW to create stored procedures in our development environment:

```
GRANT CREATEIN ON SCHEMA DEVL7083 TO PAOLORW
```

Since you may have many application developers or DBAs creating stored procedures into the same schema, you may wish to grant the CREATEIN privilege on the schema to a secondary authid that represents a group of users who create stored procedures. Each application developer could then issue a SET CURRENT SQLID statement to the secondary authid prior to creating the stored procedure in the desired schema.

Users who attempt to create a stored procedure by issuing the CREATE PROCEDURE statement without having the CREATEIN privilege on the schema receive an SQLCODE of -552 with an SQLSTATE of 42502. Here is the error message we received when authid PAOLORW attempted to create stored procedure EMPDTLSC without having been granted CREATEIN on schema DEVL7083:

```
DSNT408I SQLCODE = -552, ERROR:  PAOLORW DOES NOT HAVE THE PRIVILEGE TO PERFORM
         OPERATION CREATE PROCEDURE
DSNT418I SQLSTATE   = 42502 SQLSTATE RETURN CODE
```

The CREATEIN privilege is always required to create a stored procedure in a given schema. Note, however, that there are some considerations in how the qualifier is determined when a CREATE PROCEDURE statement is unqualified. See 7.5.7, “Resolution of unqualified stored procedure names at create time” on page 79 for details on the behaviors for handling unqualified stored procedure names in DB2 for z/OS V8 and DB2 9 for z/OS.

7.2.2 BINDADD privilege for stored procedures that contain SQL

If the stored procedure being created contains SQL statements, a package will be created and stored in the DB2 catalog. The BINDADD system privilege is required to create new packages in a DB2 subsystem. The SQL to grant BINDADD privilege to authid PAOLORW is as follows:

```
GRANT BINDADD TO PAOLORW
```

Since the BINDADD privilege is a system-level privilege, the GRANT statement only needs to be issued once per authid for a subsystem. Rather than grant BINDADD to every individual who can create stored procedures, you can grant the privilege to a new authid, which is used by a group of users who create stored procedures. Each application developer in the group could then issue a SET CURRENT SQLID statement to the new authid prior to creating the stored procedure in the desired schema.

Users who attempt to create a stored procedure by issuing the CREATE PROCEDURE statement without having the BINDADD privilege on the system where the stored procedure will reside receive an SQLCODE of -567 with an SQLSTATE of 42501. Here is the error message we received when authid PAOLORW attempted to bind the package for stored procedure EMPDTLSC without having been granted BINDADD on the subsystem:

```
BIND AUTHORIZATION ERROR USING PAOLORW AUTHORITY
PACKAGE = EMPDTLSC PRIVILEGE = BINDADD
```

7.3 Privileges required to execute stored procedures

Once a stored procedure has been created, it will most likely be executed by a number of DB2 users. Two types of authorizations are required:

- Authorization to execute the CALL statement

  The privileges required to execute the CALL depend on several factors, including the way the CALL is invoked. The CALL can be dynamic or static:

  - The dynamic invocation of a stored procedure is whenever the CALL is executed with a host variable, as shown here:

    CALL :*host-variable*

  - The static invocation of a stored procedure is whenever the CALL is executed with a procedure name, as shown here:

    CALL *procedure-name*

- Authorization to execute the stored procedure package and any dependent packages

  The privilege required to EXECUTE the package is independent of the type of CALL.

In places where we mention “authid” in this chapter, we are referring to the value of CURRENT SQLID. This is usually the authid of the process, meaning the authid being passed from the client in the case of a distributed call, or the authid associated with a CICS transaction or batch job if the stored procedure is called from CICS or batch. The authid may be changed if the calling application issues a SET CURRENT SQLID statement prior to issuing the CALL to the stored procedure.

With DB2 9, the GRANT EXECUTE statement allows for the privilege on a stored procedure to be granted to a ROLE. ROLE privileges or executing stored procedures are considered in addition to the value of CURRENT SQLID for batch, local and remote applications. Trusted contexts cannot be defined for CICS and IMS.

7.3.1 Privileges to execute a stored procedure called dynamically

For static SQL programs that use the syntax CALL host variable (ODBC applications use this form of the CALL statement), the authorization ID of the plan/package (which contains the CALL statement or an assigned role) [1] must have one of the following:

- The EXECUTE privilege on the stored procedure
- Ownership of the stored procedure
- SYSADM authority

In our example in 7.2, “Privileges required to create stored procedures” on page 67, we granted user ID PAOLORW the privileges required to create stored procedure EMPDTLSC in

[1] The DYNAMICRULES behavior for the plan or package that contains the CALL statement determines both the authorization ID and the privilege set that is held by that authorization ID.

schema DEVL7083. After creating the procedure, PAOLORW must then grant EXECUTE privilege on the procedure to the authids that will execute the procedure from a distributed client, such as a Microsoft® Windows application, which invokes the stored procedure dynamically.

To test what happens when a client authid does *not* have EXECUTE authority on a procedure that is called by the client application, we developed a stub client application on Windows that issues a CALL to stored procedure EMPDTLSC. We attempted to call the stored procedure while using authid PAOLORW, which this time did not have EXECUTE authority on the stored procedure. Figure 7-1 shows the error message that was returned to the Windows client.

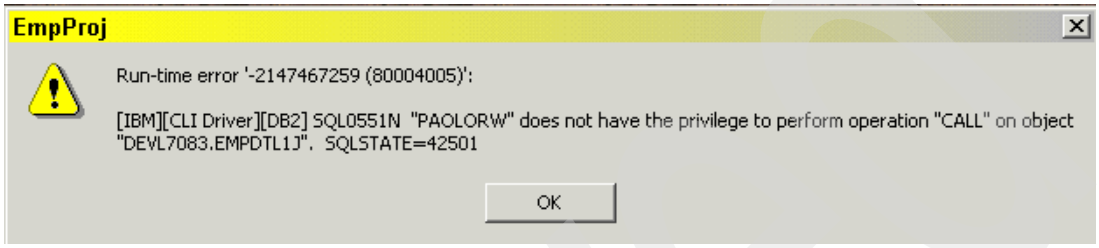

Figure 7-1 Sample error message on Windows client when EXECUTE privilege does not exist

Subsequently, we issued the following SQL statement to grant the EXECUTE privilege on the stored procedure to the client authid:

```
GRANT EXECUTE ON PROCEDURE DEVL7083.EMPDTLSC TO PAOLORW
```

We ran the stub client application again and were able to successfully execute stored procedure EMPDTLSC.

For details, refer to *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

7.3.2 Privileges to execute a stored procedure called statically

For static SQL programs that use the syntax CALL procedure, the owner (an authid or a role) of the plan or package that contains the CALL statement must have one of the following:

- The EXECUTE privilege on the stored procedure
- Ownership of the stored procedure
- SYSADM authority

It does not matter whether the current authid at execution time has the EXECUTE privilege on the stored procedure. As long as the authid has EXECUTE authority on the plan or package of the calling application, it will be able to execute any CALL statements within the calling application. This privilege is checked at the time the plan or package for the calling application is bound, unless VALIDATE(RUN) is used.

In our test case we used authid PAOLORW, which for this test scenario had been granted no access to any DB2 packages or plans and had been granted no privileges to create stored procedures, so could therefore not be the owner of a stored procedure. We ran a batch job, using an authid of PAOLORW, that executed program CALDTLSC, which is a COBOL program that calls stored procedure EMPDTLSC. Since PAOLORW had no privileges on CALDTLSC, we received the error messages shown in Example 7-2.

Example 7-2 Sample error messages on z/OS caller when EXECUTE privilege does not exist

```
PLAN CALDTLSC NOT AUTHORIZED FOR SUBSYSTEM DB9A AND AUTH ID PAOLORW

DSNT408I SQLCODE = -924, ERROR:  DB2 CONNECTION INTERNAL ERROR, 0001, 0100,
```

```
          00F30016
DSNT418I SQLSTATE   = 58006 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNAET03 SQL PROCEDURE DETECTING ERROR
```

We then issued the following SQL statement to grant the EXECUTE privilege on the package for the calling application to authid PAOLORW:

```
GRANT EXECUTE ON PLAN DEVL7083.CALDTLSC TO PAOLORW
```

We ran the batch job again and were able to successfully execute stored procedure EMPDTLSC.

7.3.3 Authorization to execute the stored procedure packages

DB2 checks the following authorization IDs in the order in which they are listed for the required authorization to execute the stored procedure package (in each case an owner is either an authid or a role):

- The owner (the definer) of the stored procedure.
- The owner of the plan that contains the statement that invokes the stored procedure package if the application is local or if the application is distributed and DB2 for z/OS is both the requester and the server.
- The owner of the package that contains the statement that invokes the stored procedure package if the application is distributed and DB2 for z/OS is the server but not the requestor, or the application uses Recoverable Resources Management Services attachment facility (RRSAF).
- The owner of the package that contains the statement that invokes the package if the application is distributed and DB2 for z/OS is the server but not the requester.
- The authorization ID as determined by the value of the DYNAMICRULES bind option for the plan or package that contains the CALL statement if the CALL statement is in the form of CALL host variable. Prior to DB2 9 for z/OS, when DYNAMICRULES(RUN) was specified, then DB2 checked the CURRENT SQLID, the primary authid (if different) and its secondary authids. DB2 9 for z/OS will also include the user's role in effect.

The privilege required to run the stored procedure package and any packages that are used under the stored procedure is any of the following:

- The EXECUTE privilege on the package
- Ownership of the package
- PACKADM authority for the package's collection
- SYSADM authority

A PKLIST entry is not required for the stored procedure package.

In case of stored procedures invoking triggers and UDF, additional authorizations are required. See *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

7.4 Network trusted context and roles

One of the challenges of designing applications that run in a three-tiered architecture is managing user IDs. End-user authentication often happens at the middleware level in a three-tiered architecture. After the middleware layer has performed the authentication, a generic user ID and password associated with the middleware layer is often what is passed in the request to DB2 for z/OS. The security exposure in this type of architecture is that there is

one generic user ID shared by all users of the application and there is no easy way to prevent use of that user ID outside of the normal context of the application. Network trusted context is a DB2 9 for z/OS enhancement that allows you to manage access to DB2 for z/OS resources in the context of a trusted network connection. Details on the network trusted context enhancement in DB2 9 for z/OS can be found in *Securing DB2 and Implementing MLS on z/OS*, SG24-6480, which was recently updated to include DB2 9 for z/OS security features. A trusted connection can be established for a local or a remote application. In this book we only discuss network trusted context as it applies to calls to remote DB2 stored procedures.

A trusted context is an independent database entity that is based on a system authorization ID (SYSTEM AUTHID) and connection trust attributes. For a remote stored procedure CALL the SYSTEM AUTHID is derived from the system user ID provided by an external entity, for example a middleware server, when initiating the connection. The connection trust attributes are specified in the CREATE TRUSTED CONTEXT statement. For a remote stored procedure CALL the attributes considered are:

- ADDRESS - IP address or domain name. The protocol is restricted to TCP/IP only.
- SERVAUTH - A resource in the RACF SERVAUTH class.
- ENCRYPTION - Minimum level of encryption for the connection:
  - NONE - No encryption. This is the default.
  - LOW - DRDA data stream encryption.
  - HIGH - Secure Sockets Layer (SSL) encryption.

For the case of a remote stored procedure CALL from a middleware server, the ADDRESS attribute would be the IP address of the server. An example of a server used for remote stored procedure CALLs in a three-tiered architecture is an IBM WebSphere Application Server. Let's look at an example of how a network trusted context could be used to control access to one of the stored procedures in our case study.

COBOL stored procedure DEVL7083.EMPDTL1C resides on our DB2 9 for z/OS system DB9A. In our original case study we called EMPDTL1C from COBOL program CALDTL1C running on DB2 for z/OS. For the testing in the above mentioned security redbook an IBM WebSphere Application Server (WAS) was established. We modified our case study to call stored procedure EMPDTL1C on a DB2 9 for z/OS subsystem from the Database Explorer view in Data Studio. We executed the call from one IP address for which a trusted connection was defined and then from another IP address for which no trusted connection was defined. We used IP addresses for Windows workstations to perform the test.

First we ran a test attempting to execute DEVL7083.EMPDTL1C using authid PAOLORW when no EXECUTE privileges had been granted on the stored procedure. We received an SQLCODE of -551 as expected. The test results are shown in Example 7-3. This proves that authid PAOLORW does not have the privilege to execute the stored procedure.

Example 7-3 Sample results when no privileges exist on a stored procedure

```
DEVL7083.EMPDTL1C - Exception occurred while running:
A database manager error occurred.SQLCODE: -551, SQLSTATE: 42501 - PAOLORW DOES NOT HAVE
THE PRIVILEGE TO PERFORM OPERATION CALL ON OBJECT DEVL7083.EMPDTL1C. SQLCODE=-551,
SQLSTATE=42501, DRIVER=3.50.152
```

Now we perform the steps necessary to define the trusted context.

1. The first step is to create the role. A role is a database entity that groups together one or more privileges and can be assigned to users via a trusted context. A role can be used in conjunction with a trusted context and stored procedures to identify one or more authids who can execute a stored procedure. We define a role called SP_CALLER, as shown in Example 7-4.

Example 7-4 DDL to create a role

```
---------+---------+---------+---------+---------+---------+
CREATE ROLE SP_CALLER;
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+
```

2. The second step is to grant the EXECUTE privilege on the stored procedure to the role. In our test case we granted execute on stored procedure EMPDTL1C to the role we just created, SP_CALLER, as shown in Example 7-5.

Example 7-5 DDL to grant EXECUTE privilege on a stored procedure to a role

```
---------+---------+---------+---------+---------+---------+---------+
GRANT EXECUTE ON PROCEDURE DEVL7083.EMPDTL1C TO ROLE SP_CALLER;
---------+---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+---------+
```

3. The third and final step is to define the trusted context. In our test case we created a trusted context called TRUSTED_EMPDTL1C using the authid of PAOLORW, a default role of SP_CALLER, and a specific IP address. The DDL we executed is shown in Example 7-6.

Example 7-6 DDL to create a trusted context using an existing role

```
---------+---------+---------+---------+---------+---------+
   CREATE TRUSTED CONTEXT TRUSTED_EMPDTL1C
    BASED UPON CONNECTION USING SYSTEM AUTHID PAOLORW
    ATTRIBUTES (ADDRESS '9.30.28.113')
    DEFAULT ROLE SP_CALLER
    ENABLE;
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+
```

At this point we are ready to run our trusted context test. Based on the above definition, authid PAOLORW should only be able to execute stored procedure DEVL7083.EMPDTL1C if it is invoked from an IP address of 9.30.28.113. The results of our test from a workstation with this IP address is shown in Example 7-7.

Example 7-7 Results of a stored procedure call from an IP address defined in trusted context

```
DEVL7083.EMPDTL1C - Run started.
Data returned in result sets is limited to the first 50 rows.
Data returned in result set columns is limited to the first 100 bytes or characters.
DEVL7083.EMPDTL1C - Calling the stored procedure.
DEVL7083.EMPDTL1C - Run completed.
```

In order to determine whether the trusted context is really working, we needed to run a test from a workstation with a different IP address. We ran the same test, without changing any of the privileges granted, on a different workstation that had a different IP address. Our results are shown in Example 7-8.

Example 7-8 Results of stored procedure call from IP address not defined in trusted context

```
DEVL7083.EMPDTL1C - Run started.
Data returned in result sets is limited to the first 50 rows.
Data returned in result set columns is limited to the first 100 bytes or characters.
```

```
DEVL7083.EMPDTL1C - Calling the stored procedure.
DEVL7083.EMPDTL1C - Exception occurred while running:
A database manager error occurred.SQLCODE: -551, SQLSTATE: 42501 - PAOLORW DOES NOT HAVE
THE PRIVILEGE TO PERFORM OPERATION CALL ON OBJECT DEVL7083.EMPDTL1C. SQLCODE=-551,
SQLSTATE=42501, DRIVER=3.50.152
DEVL7083.EMPDTL1C - Roll back completed successfully.
DEVL7083.EMPDTL1C - Run failed.
```

You can see that the trusted context definition prevented user PAOLORW from executing stored procedure EMPDTL1C from any workstation other than the one associated with the IP address for that user.

When you define a trusted context you will most likely use a domain name instead of an IP address for the ADDRESS attribute of the trusted context. We used IP addresses in our tests to be able to more easily distinguish between the two workstations executing the stored procedure.

Note: Trusted contexts cannot be defined for CICS or IMS. Therefore, if your stored procedure can be executed from CICS or IMS as well as from a DRDA connection, you should not use a trusted context to control access to the stored procedure.

Here are some design points to keep in mind when considering trusted context and roles for securing your stored procedures:

- A stored procedure can be created by a role within a trusted connection.
- A stored procedure can be owned by a role.
- You can restrict access to a stored procedure by only allowing it to be exercised via a role within a trusted connection.
- You can control who can create stored procedures by defining a role and a trusted connection.
- The role privileges are in addition to the privileges held by the allowed user of the trusted context.

For more details on the network trusted context enhancement in DB2 9 for z/OS, see Chapter 8, “Network trusted contexts and roles” in *Securing DB2 and Implementing MLS on z/OS*, SG24-6480.

7.5 Additional stored procedure security considerations

In this section we discuss some additional considerations with regard to stored procedure security. Topics discussed are:

- Privileges required when owner and binder are different
- Interaction with external security products
- Privileges for usage of distinct types
- Privileges for usage of jar files
- Dynamic SQL statements in stored procedures
- Limiting the types of SQL that can be executed
- Resolution of unqualified stored procedure names at create time
- Authorization caching

7.5.1 Privileges required when owner and binder are different

In some cases the owner of the stored procedure may be different than the authid used to bind the stored procedure package. This may occur when a DBA is responsible for creating the procedure (issuing the CREATE PROCEDURE statement) and an application developer is responsible for program preparation, including binding the stored procedure package. In that case granting the EXECUTE privilege on the stored procedure to the authid who will be executing the procedure will not be sufficient. The owner of the stored procedure package will need to grant the EXECUTE privilege on the stored procedure package to the executing authid.

7.5.2 Interaction with external security products

The SECURITY parameter of the CREATE PROCEDURE statement specifies how the stored procedure interacts with external security products, such as RACF, to control access to non-SQL resources. If a stored procedure does not require an external security product to protect access to non-SQL resources, then you should specify SECURITY DB2, which is the default. SECURITY DB2 causes access to external resources to be performed using the authid of the stored procedure address space. If the stored procedure does require an external security product to access non-SQL resources, such as a VSAM file, you can specify either SECURITY USER or SECURITY DEFINER to control access to the resource. SECURITY USER will cause the external security product to use the authid of the user who invoked the stored procedure. SECURITY DEFINER will cause the external security product to use the authid of the owner of the stored procedure.

7.5.3 Privileges for usage of distinct types

A distinct type is a data type that is specific for a customer environment. It is based on one of the built-in data types. For example, you could define a distinct type of US_DOLLARS that is based on a data type of DECIMAL(9,2). The main reason to use a distinct type is to ensure that only functions, procedures, comparisons and assignments that are defined for that type can be used for columns defined with that data type.

DB2 generates two functions associated with the distinct type: one to cast between the distinct type and its' source data type; and one to cast between the source data type and the distinct type. When the distinct type is created the owner of the type implicitly has the USAGE privilege on the type and the functions associated with the type.

Stored procedures can pass parameters that have a distinct type as a data type. The creator of the stored procedure must have the USAGE privilege on a distinct data type if that type is to be used as a parameter in the stored procedure. No additional USAGE privilege is required to any authid that is granted the EXECUTE privilege on the procedure. For example, if a distinct type of US_DOLLARS was created by authid PAOLORX, and authid PAOLORW wanted to create a stored procedure that passed a parameter with a data type of US_DOLLARS, then PAOLORX would have to issue the following SQL statement to allow PAOLRW to create the procedure:

```
GRANT USAGE ON DISTINCT TYPE US_DOLLARS TO PAOLORW
```

7.5.4 Privileges for usage of jar files

Stored procedures with language type of Java can specify a Java archive (jar) file in the EXTERNAL NAME clause. If a jar file is specified, it must exist at the time the procedure is created. In addition, the authid used to create the stored procedure must have the USAGE privilege on the jar. For example, if authid PAOLORW wishes to create Java stored procedure EMPDTL1J that specifies an external name of 'DEVL7083.EmpJar:EmpDtl1J.GetEmpDtls',

where ‘DEVL7083.EmpJar’ is the jar name, EmpDtl1J is the class name and GetEmpDtls is the method name, the ownerid or schema name that was used when the INSTALL_JAR stored procedure was executed must issue the following SQL statement to allow PAOLORW to create the Java stored procedure:

```
GRANT USAGE ON JAR DEVL7083.EmpJar TO PAOLORW
```

Note that the jar name is case sensitive. Make sure that you set caps off prior to issuing the GRANT statement.

See 13.6, “Preparing Java stored procedures” on page 195 for more details on preparing jar files and using the DB2-supplied INSTALL_JAR stored procedure.

7.5.5 Dynamic SQL statements in stored procedures

We know that stored procedures can be called by dynamic SQL programs, and can execute their work using static SQL within the procedures, and so derive the security strengths of the static SQL model. You can grant execute privilege on the procedure, rather than access privileges on the tables that are accessed in the procedures. An ODBC or JDBC application can issue a dynamic CALL statement, and invoke a static stored procedure to run under the authority of the package owner for that stored procedure.

Stored procedures with dynamic SQL are also good for security reasons, but they require a bit more effort to plan the security configuration.

All of the security topics we have discussed so far are applicable to stored procedures that are created with and contain static SQL. Stored procedures that contain dynamic SQL are influenced by the DYNAMICRULES option in effect at the time that the stored procedure package is bound.

The DYNAMICRULES option, in combination with the runtime environment, determines what values apply at runtime for dynamic SQL attributes such as authid for authorization checking, and qualifier for unqualified objects, as well as some other attributes. The set of attribute values is called the dynamic SQL statement behavior. The four dynamic SQL statement behaviors are:

- Run behavior
- Bind behavior
- Define behavior
- Invoke behavior

Each behavior represents a different set of attribute values that impact how authorizations are handled for dynamic SQL. The authorization processing for dynamic SQL in a stored procedure is impacted by the value of the DYNAMICRULES parameter when binding the stored procedure package. There are six possible options for the DYNAMICRULES parameter:

- BIND
- RUN
- DEFINEBIND
- DEFINERUN
- INVOKEBIND
- INVOKERUN

If you bind the package for the stored procedure with DYNAMICRULES(BIND) then the dynamic SQL in the stored procedure will also be authorized against the package owner for the dynamic SQL program.

For each of the other values, the authorization for dynamic SQL in a stored procedure is checked against an auth ID other than the package owner.

Table 7-1 shows how DYNAMICRULES and the runtime 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 7-1 How is runtime 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

The DYNAMICRULES option along with the runtime environment of a package (whether the 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 7-2 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, secondary authorization IDs and role in effect are checked	Owner of user-defined function or stored procedure	Authorization ID of invoker
Default qualifier for unqualified objects	Bind OWNER or QUALIFIER value	CURRENT SCHEMA, which defaults to CURRENT SQLID if not explicitly set	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

Dynamic SQL attribute	Bind behavior	Run behavior	Define behavior	Invoke behavior
Can execute GRANT, REVOKE, ALTER, DROP, RENAME	No	Yes	No	No

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. Figure 7-2 shows how complex the choices can be when invoking a stored procedure.

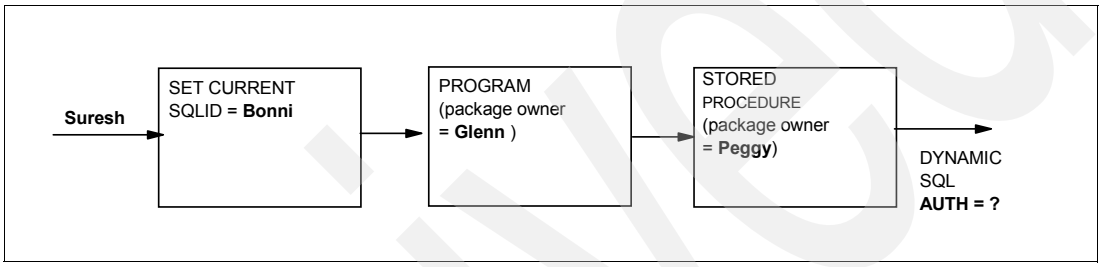

Figure 7-2 Security implications of dynamic SQL in a stored procedure

Consider a user, Suresh, using a current SQLID of Bonni, who executes a package bound by Glenn. This package calls a stored procedure created and bound by Peggy. Whose authority is checked at runtime? 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)
- Bonni (current SQLID)
- Glenn (owner of package) or
- Peggy (owner of stored procedure)

Due to the variety of options available, it is difficult to make a general recommendation that applies to all situations. See Chapter 9 “Controlling access to DB2 objects” in *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413 for more details on the DYNAMICRULES option of the BIND command to help you determine which value is appropriate for your application.

7.5.6 Limiting the types of SQL that can be executed

One of the options on the CREATE PROCEDURE statement is MODIFIES SQL DATA. The valid values are:

- MODIFIES SQL DATA
- READS SQL DATA
- CONTAINS SQL DATA
- NO SQL

This option can be used to control the types of SQL statements that may be executed within the stored procedure. For example, a stored procedure created with the option READS SQL DATA cannot include an INSERT, UPDATE, or DELETE statement. For complete details on

which types of SQL statements are allowed for each option value, refer to the description of the CREATE PROCEDURE (external) statement in *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

7.5.7 Resolution of unqualified stored procedure names at create time

Prior to DB2 for z/OS V8 you did not have the capability to explicitly specify a schema name; instead DB2 would always use the value of special register CURRENT SQLID to resolve unqualified database object references. If CURRENT SQLID was explicitly set via the SET CURRENT SQLID statement, then DB2 would use this value as the qualifier. If CURRENT SQLID was not explicitly set then DB2 would use the authorization ID of the process, such as the TSO user ID or the user ID associated with a batch job.

If you defined a stored procedure in DB2 V7 or DB2 V8, but did not qualify the stored procedure name on the CREATE PROCEDURE statement, the stored procedure would be created under the schema of the CURRENT SQLID. An example of creating an unqualified stored procedure in DB2 for z/OS V7 or V8 is shown in Example 7-9. The GRANT statement shows that the procedure does exist under the schema that was set via the SET CURRENT SQLID statement.

Example 7-9 DB2 V7 and V8 CREATE PROCEDURE with no qualifier

```
---------+---------+---------+---------+---------+---------+
SET CURRENT SQLID = 'PAOLOR4';
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+
CREATE PROCEDURE EMPDTL1C
(
 IN  PEMPNO        CHAR(6)
,OUT PLASTNAME     VARCHAR(15)
)
RESULT SETS 0
EXTERNAL NAME EMPDTL1C
LANGUAGE COBOL
PARAMETER STYLE GENERAL
WLM ENVIRONMENT DB9AWLM
STAY RESIDENT YES
COLLID DEVL7083
PROGRAM TYPE SUB
COMMIT ON RETURN NO ;
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+
GRANT EXECUTE ON PROCEDURE PAOLOR4.EMPDTL1C TO PUBLIC ;
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
```

DB2 for z/OS V8 New Function Mode introduced the new special register CURRENT SCHEMA, which specifies the schema name used to qualify unqualified database object references in SQL statements. You could set CURRENT SCHEMA to a different value than CURRENT SQLID, but you could only use the CURRENT SCHEMA value to reference existing database objects. You could not create any database objects using a different schema than the CURRENT SQLID. If you attempted to do this you would receive an SQLCODE of -20283, as shown in Example 7-10.

Example 7-10 DB2 V8 CREATE PROCEDURE with SET SCHEMA and no qualifier

```
---------+---------+---------+---------+---------+---------+---------+---------+
SET CURRENT SQLID = 'PAOLOR4';
---------+---------+---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+---------+---------+
SET SCHEMA = 'DEVL7083';
---------+---------+---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+---------+---------+
CREATE PROCEDURE EMPDTL1C
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
)
RESULT SETS 0
EXTERNAL NAME EMPDTL1C
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB9AWLM
STAY RESIDENT YES
COLLID DEVL7083
PROGRAM TYPE SUB
COMMIT ON RETURN NO ;
---------+---------+---------+---------+---------+---------+---------+---------+
DSNT408I SQLCODE = -20283, ERROR:  A DYNAMIC CREATE STATEMENT CANNOT BE
         PROCESSED WHEN THE VALUE OF CURRENT SCHEMA DIFFERS FROM CURRENT SQLID
DSNT418I SQLSTATE   = 429BN SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNXODDL SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = 2  0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'00000002'  X'00000000'  X'00000000'  X'FFFFFFFF'
         X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
---------+---------+---------+---------+---------+---------+---------+---------+
DSNE618I ROLLBACK PERFORMED, SQLCODE IS 0
```

DB2 9 for z/OS has been enhanced to look at the CURRENT SCHEMA value and to use it, if supplied, to resolve unqualified stored procedure names on a CREATE statement. If the CURRENT SCHEMA is not supplied, then the value of the CURRENT SQLID special register will be used, whether it was set implicitly or explicitly. Example 7-11 shows the successful SQLCODE received for the CREATE PROCEDURE statement using the CURRENT SCHEMA value. The GRANT statement shows that the procedure does exist under the schema that was set via the SET SCHEMA statement.

Example 7-11 DB2 V9 CREATE PROCEDURE with SET SCHEMA and no qualifier

```
---------+---------+---------+---------+---------+---------+
SET CURRENT SQLID = 'PAOLOR4';
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+
SET SCHEMA = 'DEVL7083';
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+
```

```
CREATE PROCEDURE EMPDTL1C
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
)
RESULT SETS 0
EXTERNAL NAME EMPDTL1C
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB9AWLM
STAY RESIDENT YES
COLLID DEVL7083
PROGRAM TYPE SUB
COMMIT ON RETURN NO ;
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+
GRANT EXECUTE ON PROCEDURE DEVL7083.EMPDTL1C TO PUBLIC ;
---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
```

This DB2 9 for z/OS enhancement allows you to create stored procedures with a schema that is different from the CURRENT SQLID. DB2 will use the value of the CURRENT SCHEMA special register if supplied. If no CURRENT SCHEMA value is supplied, then DB2 will use the value of the CURRENT SQLID special register and the CURRENT SCHEMA special register will also contain that value. Note that the developer still needs to have the CREATEIN privilege on the schema in which they are creating the stored procedure.

APAR PK49647 provides this functionality within the stored procedure application development tools for external SQL Language procedures. For details on this behavior in the development tools, see 27.7.5, “Behavior when setting the Current Schema project property” on page 717.

7.5.8 Authorization caching

When DB2 for z/OS receives a request to access a resource, such as a request to execute a stored procedure, DB2 must determine whether the user making the request has the privilege to execute the stored procedure. Depending on how your security scheme is defined, DB2 has to perform one or more of the following operations:

- Search for the necessary privilege in an authorization cache for PUBLIC
- Search for the necessary privilege in an authorization cache for a specific auth ID or role
- Search for the necessary privilege in the DB2 catalog for PUBLIC, a specific auth ID or role

If DB2 can determine whether an auth ID has a privilege by searching an authorization cache, then it can perform that operation much faster than if it has to go to the DB2 catalog to make that determination. Privileges for stored procedures and user-defined functions are stored in the routine authorization cache. The size of the cache is determined by DB2 system parameter CACHERAC.

Many customers use the default value for CACHERAC, which was 32 KB in DB2 V7 and is 100 KB in DB2 V8 and V9. If you run a large number of stored procedures concurrently, then the default value is almost certainly too small. If your cache is too small, then entries in the cache are overwritten and then have to be re-read from the DB2 catalog. See the topic “Authorization caching” on page 416 in 19.2.5, “Overview of performance knobs” on page 411 for details on monitoring the effectiveness of your cache so you can determine the appropriate size.

8

Operational issues

In this chapter we discuss the operational aspects of what actions you must take after you change a stored procedure definition, its parameters, or its program logic. We also discuss what steps you can take to prevent stored procedures from looping or hanging, and how to terminate them if you need to. We discuss how you can control what happens to subsequent executions of the stored procedure after it fails.

This chapter contains the following:

- Refreshing the stored procedure environment
- Handling error conditions in the application environment
- Preventing hanging or looping stored procedures
- Terminating hanging or looping stored procedures
- Handling application failures

8.1 Refreshing the stored procedure environment

After a stored procedure has become operational, various things can change that require you to refresh the environment. Some examples of when this may be necessary are:

- The stored procedure logic changes or the parameters change and you create a new load module that must replace the existing cached load module. In this case, you must refresh the Language Environment.
- You want to make a change to the startup JCL for the stored procedure address space. In this case, you must restart the stored procedure address space.

Update the environment by using either the REFRESH option or the QUIESCE option followed by the RESUME option. REFRESH is the preferred method (instead of QUIESCE and RESUME) for all changes (such as changed load modules) that do not involve a change in the startup JCL, since it causes minimal interruption in processing of requests.

After a REFRESH, all *new address spaces* will use the new JCL, but any address space currently running will continue with the old JCL unless you do the QUIESCE/RESUME. So, in a busy and stable environment the old JCL may remain active for a while, unless the REFRESH/QUIESCE is issued. The REFRESH and QUIESCE options should be used as follows:

- Use the REFRESH option of the VARY z/OS command to refresh a WLM environment. Refreshing the WLM environment starts a new instance of each address space that is active for this WLM environment. Existing address spaces stop when the current requests that are executing in those address spaces complete. The following example shows the refreshing of the environment DB2GDEC1:

  ```
  /VARY WLM,APPLENV=DB2GDEC1,REFRESH
  ```

  When you execute this command, it is an application environment that is refreshed, not just a stored procedure. Therefore, all stored procedures that are associated with the application environment DB2GDEC1 are refreshed, and new invocations of each stored procedure will automatically execute in the new instance of the address space.

  You can also call the DB2-supplied stored procedure WLM_REFRESH for this purpose. See Appendix A.2, “Refresh a WLM environment with AdminWLMRefresh” on page 817 for details.

- Use the QUIESCE option of the VARY z/OS command to stop all stored procedure address spaces that are associated with the WLM application environment. The address spaces stop when the current requests that are executing in those address spaces complete. The following example shows the quiesce of the environment DB2GDEC1:

  ```
  /VARY WLM,APPLENV=DB2GDEC1,QUIESCE
  ```

  When you execute this command, you affect all stored procedures that are associated with the application environment DB2GDEC1.

  You follow this with the RESUME option of the VARY z/OS command to start all stored procedure address spaces that are associated with the WLM application environment. In general, you use this option for changes to the startup JCL only and new address spaces start when the JCL changes are complete. The following example shows the restart of environment DB2GDEC1:

  ```
  /VARY WLM,APPLENV=DB2GDEC1,RESUME
  ```

See *z/OS V1R8.0 MVS Planning: Workload Management,* SA22-7602-13 for more information about the command VARY WLM.

There are very few instances where you need to STOP or START an individual stored procedure. If the stored procedure abends a specified number of times (see 8.5, “Handling application failures” on page 87 for details), DB2 places it in a STOPABN status and you must issue a START command as shown below to make it operational again:

```
-START PROCEDURE(DEVL7083.EMPDTLSC)
```

This results in:

```
DSNX946I  -DB2G DSNX9ST2 START PROCEDURE SUCCESSFUL FOR
DEVL7083.EMPDTLSC
DSN9022I  -DB2G DSNX9COM '-START PROC' NORMAL COMPLETION
```

Note that for native SQL procedures you can only STOP or START the current version of a procedure. There is no way to stop or start a specific version. For more details on native SQL procedures see Chapter 15, “Native SQL procedures” on page 253.

If you want to temporarily queue requests while you quickly change something, you can find useful the command STOP PROC ACTION(QUEUE).

STOP PROC ACTION(REJECT) can be used in case you believe too many requests are queuing up or a flood of stored procedure invocations is causing too many WLM stored procedure address space instances to start. In either of these cases you should be open a problem to IBM.

8.2 Handling error conditions in the application environment

In the previous section we described situations where you want to refresh the stored procedure environment for a planned change. There are also instances when WLM experiences errors that affect the application environments. WLM stops the creation of new address spaces when one of the following conditions exists:

- JCL errors in the procedure associated with the application environment.
- Coding errors in the stored procedure that cause five unexpected terminations of the address space.
- Five operator cancellations of the stored procedures address space within 10 minutes.
- Failure of the address space to connect to WLM.

The application environment first enters the STOPPING state, then the STOPPED state after all systems in the sysplex have accepted the action. In STOPPED state, no new address space are created. An existing address space continues to be operational and can execute new stored procedure requests.

When the application environment is in STOPPED state, you can make changes to libraries, the JCL procedure, or any other changes needed to repair the condition that caused WLM to stop address space creation. After you solve the problem, use the RESUME option of the VARY WLM command.

Additionally, when WLM stops an application environment, it sends the following message to the console and system log.

```
IWM032I Internal stop for xxxxx completed
```

Where `xxxxx` is the application environment name.

The message on the console will not be highlighted. So the effect may not be known immediately as long as you have active address spaces serving the requests. The clients will

start receiving SQLCODE -471 when WLM tries to start a new address space and fails due to the application environment being in “stopped state.”

For sensitive applications, pro-active monitoring should be in place to track IWM032I messages and alert concerned persons and groups. Note that IWM032I messages appear even in response to the VARY commands. The key word to look for in the message is `IWM032I Internal stop` to distinguish between the VARY command and WLM stopping the AE.

8.3 Preventing hanging or looping stored procedures

You can control the total amount of processor time, in CPU service units, for a single execution of a stored procedure by specifying it through the ASUTIME parameter. ASUTIME NO LIMIT means that there is no limit on the service units (this is the default). ASUTIME LIMIT n (where *n* is a positive integer in the range of 1 to 2 147 483 647) means that DB2 cancels the stored procedure if the stored procedure uses more service units than the specified limit.

If the execution profile of the stored procedure is predictable (that is, it does a fixed amount of work), you can set this limit quite easily. For example, for a stored procedure that generally consumes less than 1 CPU second, you can set this limit to a small reasonable number (say 30 CPU seconds). This prevents any run-away queries without causing any accidental cancel of a stored procedure that should have been allowed to run.

If the execution profile of the stored procedure is unpredictable (that is, the work it does varies and can be impacted by the data), ASUTIME is a little harder to set. However, we recommend that a reasonable upper limit be set for all stored procedures, and that no stored procedure should be allowed to run with the NO LIMIT option (which again, is the dangerous default).

Since accidental looping is more likely in the development environments, you may consider placing a smaller limit in the development environment and a higher limit in a production environment.

8.4 Terminating hanging or looping stored procedures

When a stored procedure hangs or appears to be in an endless loop, the following steps terminate it in a controlled manner with minimal impact to other applications. You should proceed to the next step only if the previous step does not succeed in terminating the stored procedure after waiting for a reasonable time (around 10 seconds):

1. Cancel the thread. This terminates the stored procedure if it is issuing SQL calls. For a stored procedure hanging outside DB2, this step does not achieve anything except to flag it for termination, which will take effect at the next SQL call, if it makes one.
2. If the stored procedure is called from a local application, cancel the invoking job.
3. Refresh the WLM environment where the stored procedure is running. This starts a new address space instance for all new work, and allows all work currently executing (except the problem stored procedure) to complete. The problem stored procedure should be the only active thread in the old stored procedure address space instance. You can refresh the environment by issuing the /VARY WLM command or by executing the DB2-supplied stored procedure WLM_REFRESH as discussed in 8.1, “Refreshing the stored procedure environment” on page 84.
4. Cancel the started task for the WLM stored procedure address space in which the problem stored procedure is executing. It should be the only one left when the refresh was issued.

8.5 Handling application failures

When a stored procedure abends continually, it can have a negative impact on the other stored procedures that execute in the same application environment. DB2 provides you with two options to manage these stored procedures, one at the DB2 subsystem level and one at the individual stored procedure level.

At the subsystem level you can specify the maximum number of failures of a stored procedure before it is placed in a stopped status. This is controlled by the zparm STORMXAB on installation panel DSNTIPX.

There are options on the CREATE PROCEDURE statement that allow you to control this behavior at the individual stored procedure level. The possible choices are discussed below:

- STOP AFTER SYSTEM DEFAULT FAILURES

  This specifies that the stored procedure should be placed in a stopped status after the number of failures reaches the value of MAX ABEND COUNT (STORMXAB) on installation panel DSNTIPX. This is the default and the only behavior allowed in V7.

- STOP AFTER n FAILURES

  This specifies that the stored procedure should be placed in a stopped status after n failures. The value of n can be an integer between 1 and 32767.

- CONTINUE AFTER FAILURE

  This specifies that the stored procedure should not be placed in a stopped status after any number of failures.

The option you should choose for this parameter is based on various factors including the following:

- Frequency of execution
- Monitoring and early detection of failures
- Criticality of the stored procedure
- Most likely cause of failure (program logic, resources, data)

You may also want to configure the test environment different from the production environment. For example, a large number of failures can be tolerated in test to eliminate frequent DBA intervention, but a lower limit can be specified in a production environment.

The ability to set this limit at the stored procedure level gives you complete control over the environment, and can eliminate repeated resource-intensive failures of a stored procedure for the same reason until the problem is resolved.

Part 3

Developing stored procedures

In this part we describe how to define and code stored procedures. Application programmers and DBAs will be interested in the topics discussed in this part. We provide examples of the CREATE PROCEDURE statements and examples of programming some stored procedures in COBOL, C, REXX, Java and SQL procedures language (external and native). We also look at debugging and code management considerations.

This part contains the following chapters:

9

Defining stored procedures

In order for a stored procedure to run, you must prepare the environment for it and define it to DB2. You define the stored procedure using the CREATE PROCEDURE statement. Some of the parameters can be modified by using the ALTER PROCEDURE statement. In this chapter we discuss in detail the most important parameters that can be specified. We discuss the possible options for each parameter, their impact on the operation of the stored procedure, and recommendations for each parameter.

Note that when a stored procedure is called by a trigger (see Chapter 26, “Using triggers and UDFs” on page 629 for details), the stored procedure must be defined first—that is, CREATE PROCEDURE must be issued before CREATE TRIGGER. Similarly, an attempt to drop the stored procedure used by a trigger will result in an error; instead, you must drop the trigger first. For an external stored procedure, the DB2 package does not need to exist until the trigger is executed.

The list of options discussed here is not exhaustive. See *DB2 Version 9.1 SQL Reference,* SC18-9854 for details.

This chapter contains the following:

- CREATE or ALTER PROCEDURE parameters
- Examples of stored procedure definition
- Summary of recommendations

9.1 CREATE or ALTER PROCEDURE parameters

The CREATE or ALTER PROCEDURE statements are the DDL for the procedure object. With these statements, we inform DB2 and define in the catalog the characteristics of the stored procedure. For an introduction, refer to Chapter 2, “Stored procedures overview” on page 9. In this section we discuss the most important parameters of the CREATE and ALTER PROCEDURE statements. For each parameter, we discuss meaning, choices, and recommendations.

First, we look at some stored procedures-related subsystem default values, which are defined at install time.

9.1.1 The installation panels

Figure 9-1 shows the DB2 Routine Parameters install panel DSNTIPX for DB2 V9. The entries on this panel are used to generate the sample JCL used to start the stored procedures address space where the stored procedures or user-defined functions will run. The values shown are the default values. Notice that, like most DB2 system parameters, options 3 through 7 are online DSNZPARMs, and can be modified after the installation without stopping DB2.

```
DSNTIPX         MIGRATE DB2 - ROUTINE PARAMETERS
===>

Scrolling backward may change fields marked with asterisks
Enter data below:

* 1 WLM PROC NAME    ===> DB8RWLM    WLM-established stored procedure JCL PROC
  2 NUMBER OF TCBS   ===> 8          Number of concurrent TCBs (1-100)
  3 MAX ABEND COUNT  ===> 0          Allowable ABENDs for a procedure (0-255)
  4 TIMEOUT VALUE    ===> 180        Seconds to wait before SQL CALL or
                                     function invocation fails (5-1800,NOLIMIT)
  5 WLM ENVIRONMENT  ===>            Default WLM env name
  6 MAX OPEN CURSORS ===> 500        Maximum open cursors per thread
  7 MAX STORED PROCS ===> 2000       Maximum active stored procs per thread




PRESS:  ENTER to continue   RETURN to exit   HELP for more information
```

Figure 9-1 The DSNTIPX panel

- 1 WLM PROC NAME - It specifies a name for the stored procedures JCL procedure that is generated during installation. This procedure is used for a WLM-established stored procedures' address space. If this field has a blank, the JCL procedure is still generated. The default procedure will be named by appending the string WLM to the DB2 subsystem name.
- 2 NUMBER OF TCBS - It specifies how many SQL CALL statements or invocations of user-defined functions can be processed *concurrently* in one address space. This value is limited by the USS MAXPROCUSER (maximum number of processes for the user) value. As of V8, the value specified in NUMTCB is sent to WLM as a maximum task limit.

- 3 MAX ABEND COUNT - The DSNZPARM parameter is STORMXAB. It specifies the number of times a stored procedure or an invocation of a user-defined function is allowed to terminate abnormally, after which SQL CALL statements for the stored procedure or user-defined function are rejected. The default of 0 (recommended for production) means that the first abend of a stored procedure causes SQL CALLs to that procedure to be rejected. This parameter is subsystem wide, which means that you have to treat all stored procedures and UDFs equally. However, as of DB2 V8, you can specify a value for each stored procedure or UDF.
- 4 TIMEOUT VALUE - The DSNZPARM parameter is STORTIME. It specifies the number of seconds DB2 waits for an SQL CALL to be assigned to one TCB in a DB2 stored procedures address space. If the time interval expires, the SQL statement fails.

  Recommendation: Do not select the NOLIMIT value. If the stored procedure address space is down for some reason or the user-defined function does not complete, your SQL request hangs until the request is satisfied or the thread is canceled.

- 5 WLM ENVIRONMENT - The DSNZPARM parameter is WLMENV. It specifies the name of the WLM_ENVIRONMENT to use for a stored procedure when a value is not given for the WLM_ENVIRONMENT option on the CREATE FUNCTION or CREATE PROCEDURE statements. Specify a default WLM environment even if you do not plan to use external stored procedures. You need a WLM environment for debugging native SQL procedures using the DB2 Unified Debugger.
- 6 MAX OPEN CURSORS - The DSNZPARM parameter is MAX_NUM_CUR. It specifies the maximum number of cursors, including allocated cursors, open per thread. If an application attempts to open a thread after the maximum is reached, the statement will fail. This option is applicable as of DB2 V8.
- 7 MAX STORED PROCS - The DSNZPARM parameter is MAX_ST_PROC. It specifies the maximum number of stored procedures per thread. If an application attempts to call a stored procedure after the maximum is reached, the statement will fail. This count is cleared at commit time. This option is applicable as of DB2 V8.

Figure 9-2 on page 94 shows the DB2 Protection install panel DSNTIPP for DB2 V9. Specify how much storage to allocate for the caching of routine authorization information for all routines on this DB2 member. Routines include stored procedures, CAST functions and user-defined functions. For details on setting the routine authorization cache, see 7.5.8, “Authorization caching” on page 81.

```
DSNTIPP         INSTALL DB2 - PROTECTION
===>

Enter data below:

 1  ARCHIVE LOG RACF  ===> NO         RACF protect archive log data sets
 2  USE PROTECTION    ===> YES        DB2 authorization enabled.  YES or NO
 3  SYSTEM ADMIN 1    ===> SYSADM     Authid of system administrator
 4  SYSTEM ADMIN 2    ===> SYSADM     Authid of system administrator
 5  SYSTEM OPERATOR 1 ===> SYSOPR     Authid of system operator
 6  SYSTEM OPERATOR 2 ===> SYSOPR     Authid of system operator
 7  UNKNOWN AUTHID    ===> IBMUSER    Authid of default (unknown) user
 8  RESOURCE AUTHID   ===> SYSIBM     Authid of Resource Limit Table creator
 9  BIND NEW PACKAGE  ===> BINDADD    Authority required: BINDADD or BIND
10  PLAN AUTH CACHE   ===> 3072       Size in bytes per plan (0 - 4096)
11  PACKAGE AUTH CACHE===> 100K       Global - size in bytes (0-5M)
12  ROUTINE AUTH CACHE===> 100K       Global - size in bytes (0-5M)
13  DBADM CREATE AUTH ===> NO         DBA can create views/aliases for others
14  AUTH EXIT LIMIT   ===> 10         Access control exit shutdown threshold
```

Figure 9-2 The DSNTIPP panel

9.1.2 The CREATE (or ALTER) PROCEDURE statement

The CREATE PROCEDURE statement is used to define a stored procedure. In Chapter 2, “Stored procedures overview” on page 9 we introduce the objects involved in creating a stored procedure, and their relationships. In this section we show parameters for the two types of procedures, external and SQL, some of which are discussed later in this chapter. For details on the statement, refer to *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

Figure 9-3 shows the general structure of the statement. The parameter declaration contains the definition of the input and output parameters passed to the calling program and the data definition. These definitions will have to match the ones in the calling program, as we show in the examples in the various languages.

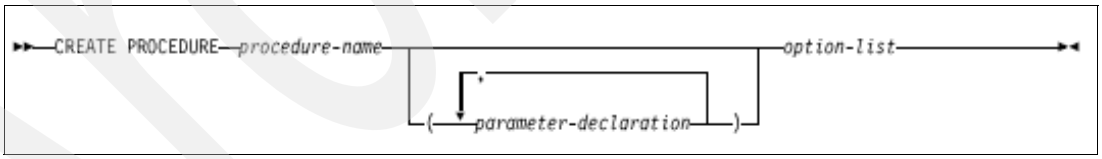

Figure 9-3 The CREATE PROCEDURE statement structure

The option lists contain the definition of the characteristics of the stored procedure.

CREATE PROCEDURE (EXTERNAL) option list

Figure 9-4 shows the CREATE PROCEDURE option list for an external stored procedure. This list is for DB2 V9.

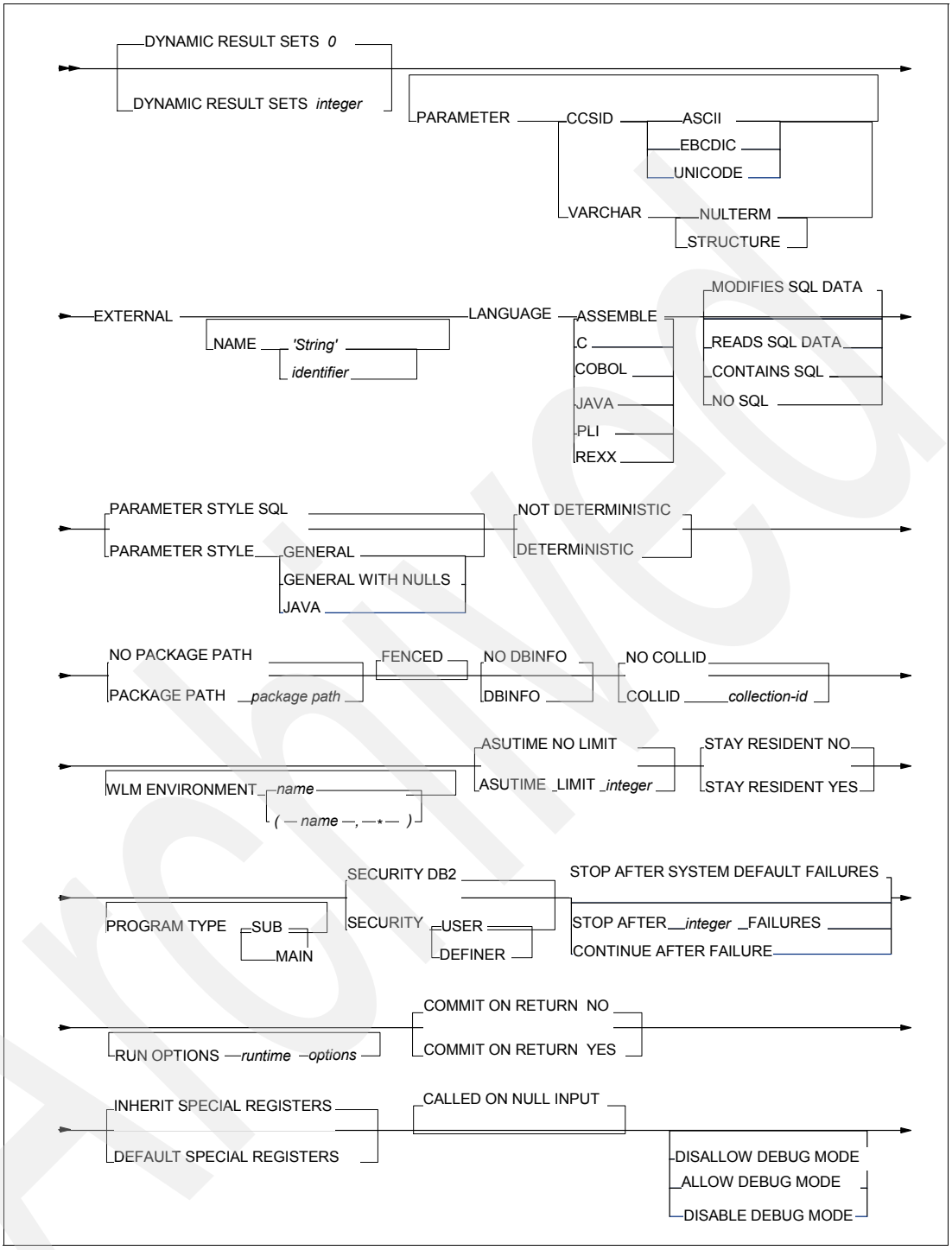

Figure 9-4 The option list for CREATE and ALTER PROCEDURE EXTERNAL

CREATE PROCEDURE (SQL- external) option list

Figure 9-5 on page 96 shows the CREATE PROCEDURE option list for an external SQL stored procedure. This list is also for DB2 V9.

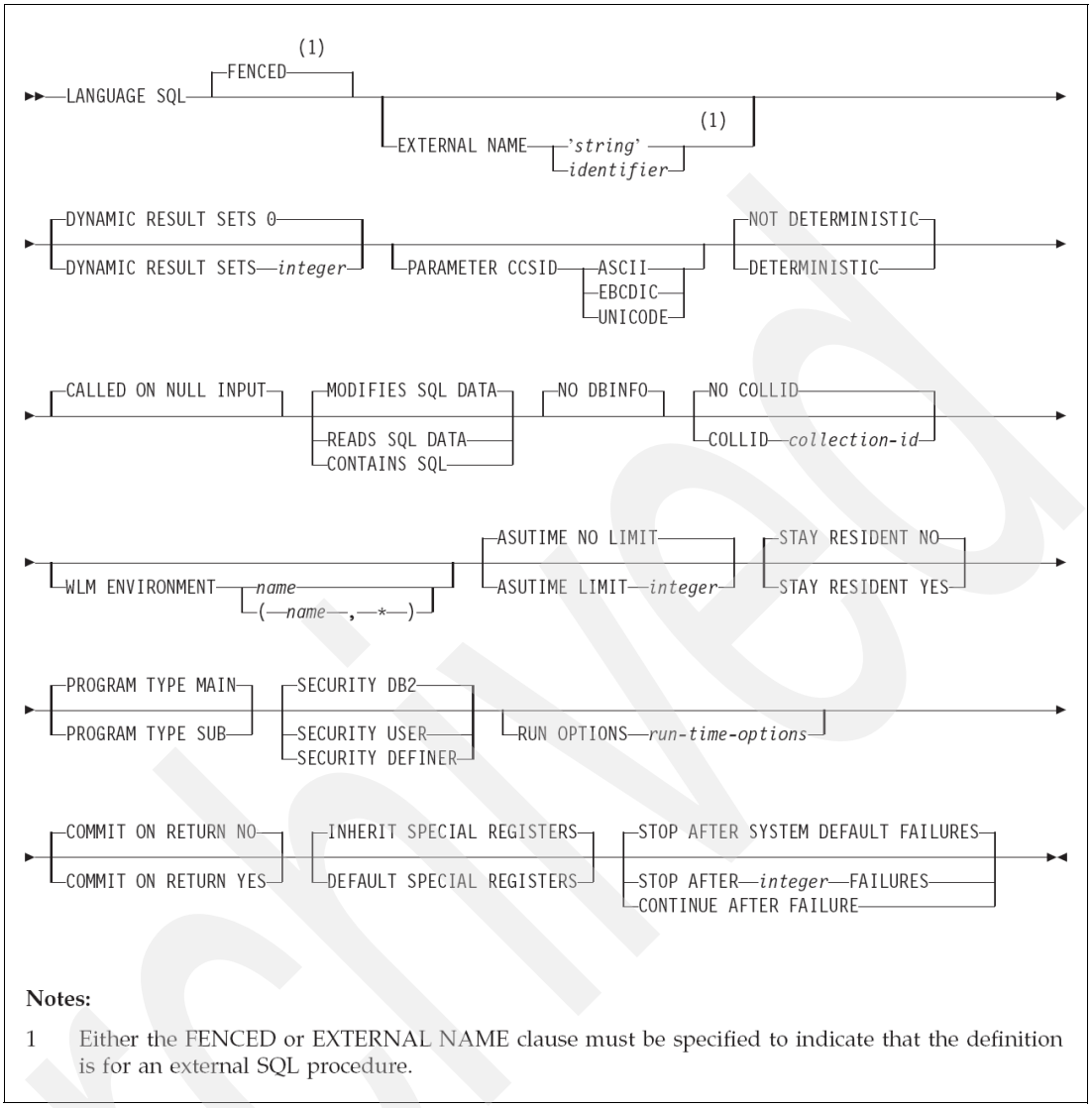

Figure 9-5 The option list for CREATE and ALTER PROCEDURE SQL - external

CREATE PROCEDURE (SQL - native) option list

Figure 9-6 on page 97 shows the CREATE PROCEDURE option list for a native SQL stored procedure which is new in DB2 V9.

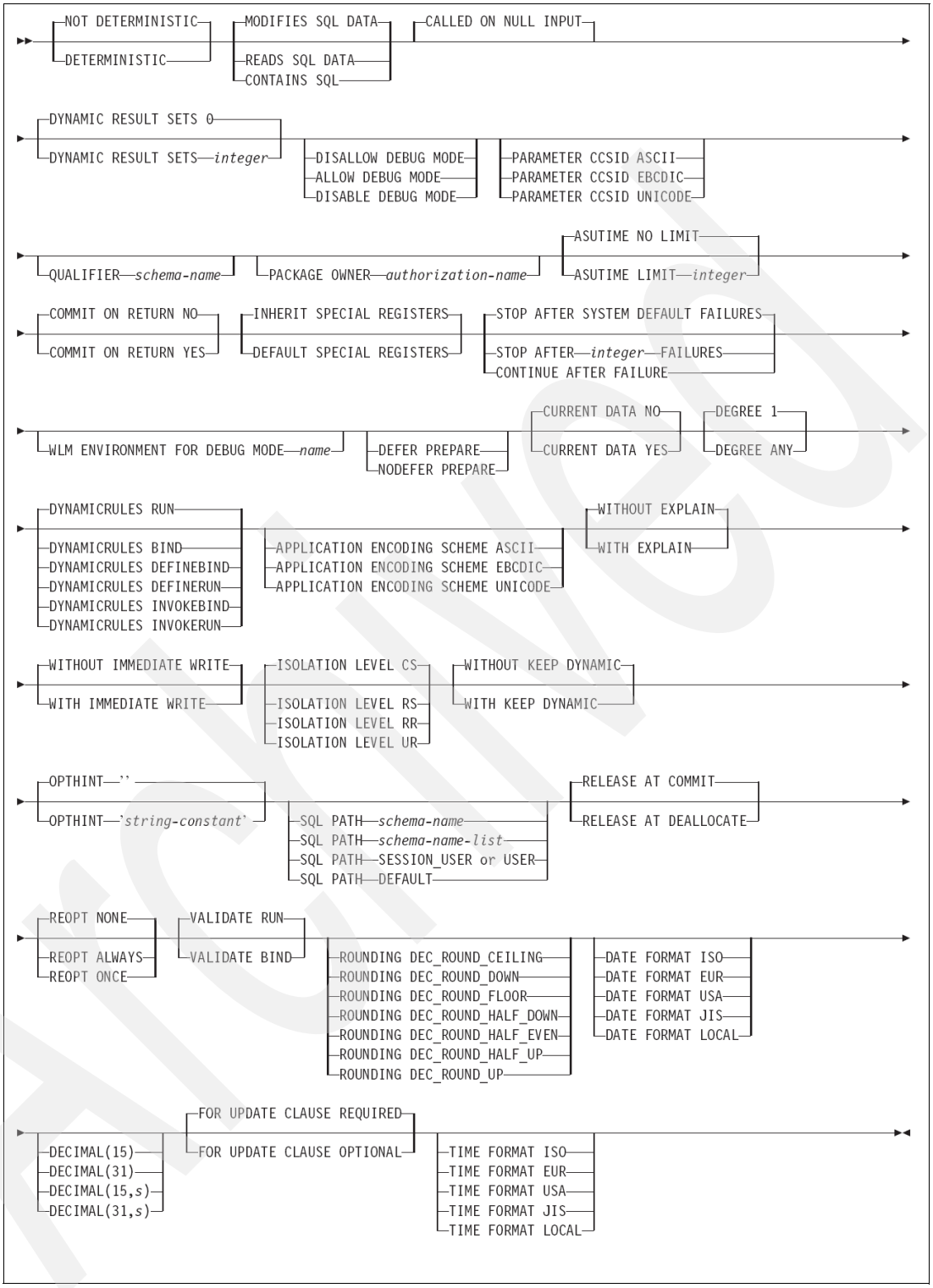

Figure 9-6 The option list for CREATE and ALTER PROCEDURE SQL - native

In comparison to what we have for external SQL procedures, most of the parameters are basically the same as those used in BIND and REBIND. See 15.2.2, “CREATE PROCEDURE syntax” on page 265 for a discussion of these parameters.

9.1.3 Number of returned result sets

A stored procedure will in general have input and output parameters. In addition, if a table containing multiple rows (known as a result set) is to be returned by the stored procedure, this must be specified in the definition.

The maximum number of result sets that can be returned by the stored procedure is controlled by the DYNAMIC RESULT SETS parameter. The possible choices are any number between 0 and 32767, with 0 being the default. Specifying a number other than 0 does not mean that the stored procedure must return that many result sets, it simply specifies the upper boundary.

Recommendation: There are no known implications in specifying a number larger than necessary, and we recommend using a reasonably large number to eliminate the need for maintenance (through ALTER PROCEDURE) at some later time.

9.1.4 Programming languages support

The application programming language in which the stored procedure is written is specified by the LANGUAGE parameter. Specify the appropriate language option such as ASSEMBLE, C, COBOL, Java, PLI, or REXX. Certain restrictions apply for Java and REXX stored procedures. Note the dependency between this and other parameters such as PARAMETER STYLE and PROGRAM TYPE.

When LANGUAGE Java is specified, the EXTERNAL NAME clause must be specified with a valid external Java routine name and PARAMETER STYLE must be Java. The procedure must be a public static method of the specified Java class. DBINFO, PROGRAM TYPE MAIN, and RUN OPTIONS are not permissible parameters.

When LANGUAGE REXX is specified, PARAMETER STYLE of SQL is not permissible (must specify GENERAL or GENERAL WITH NULLS).

9.1.5 Types of SQL supported

You indicate this by specifying one of four choices, namely: NO SQL, MODIFIES SQL DATA, READS SQL DATA or CONTAINS SQL:

NO SQL The stored procedure cannot execute any SQL statements. For a Java procedure whose EXTERNAL NAME specifies a jar, this option is not allowed.

MODIFIES SQL DATA The stored procedure can execute any allowable SQL statement including INSERT, UPDATE and DELETE. This is the default.

READS SQL DATA The stored procedure can execute any allowable SQL statement except those that modify data such as INSERT, UPDATE, and DELETE.

CONTAINS SQL The stored procedure cannot execute any SQL statement that reads or modifies SQL data. For example, statements such as SELECT, INSERT, UPDATE, and DELETE are not permitted but CALL, COMMIT, and SET are permitted.

See “Appendix C” of *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854 for details on which statements are allowed depending on the value of this parameter.

There are no known performance implications in using the most general form—MODIFIES SQL DATA. This eliminates the need to change the parameters later if the functionality of the stored procedure changes. You may want to specify a more restrictive parameter to ensure, for example, that updates do not happen within the scope of a stored procedure.

9.1.6 Passing parameters

You indicate the linkage convention used to pass parameters to the stored procedure by specifying the PARAMETER STYLE. This determines whether or not any parameters are passed to the stored procedure in addition to those specified on the CALL statement. The possible choices are discussed below:

SQL In this case, in addition to the parameters on the CALL statement, the following arguments are also passed to the stored procedure:

- A null indicator for each parameter on the CALL statement
- When a null indicator is set to -1, the parameter corresponding to that null indicator is not passed to or from the stored procedure, thus saving network traffic in a distributed environment.
- The SQLSTATE to be returned to DB2
- The qualified name of the stored procedure
- The specific name of the stored procedure
- The SQL diagnostic string to be returned to DB2
- If DBINFO is specified, an additional parameter, the DBINFO structure, is also passed.

This is the default. It cannot be used for REXX or Java.

GENERAL Only the parameters on the CALL statement are passed to the stored procedure. NULLs are not allowed as values for any INPUT or INOUT parameter.

GENERAL WITH NULLS

In addition to the parameters on the CALL statement, another argument is also passed to the stored procedure. The additional argument contains an array of null indicators, one for each of the parameters on the CALL statement, that enables the stored procedure to accept or receive null parameter values.

When a null indicator is set to -1, the parameter corresponding to that null indicator is not passed to or from the stored procedure, thus saving network traffic in a distributed environment.

For SQL stored procedures, this is the only option.

JAVA The stored procedure uses a convention for passing parameters that conforms to the Java and SQLJ specifications This option can only be specified for Java and, for Java, it is the only option.

For REXX stored procedures, GENERAL and GENERAL WITH NULLS are the only valid values, so do not use the default value of SQL for REXX stored procedures.

Figure 9-7 shows the structure of the parameter list of the external procedure when PARAMETER STYLE GENERAL is used. As a general rule the PARAMETER STYLE has no impact on the calling application.

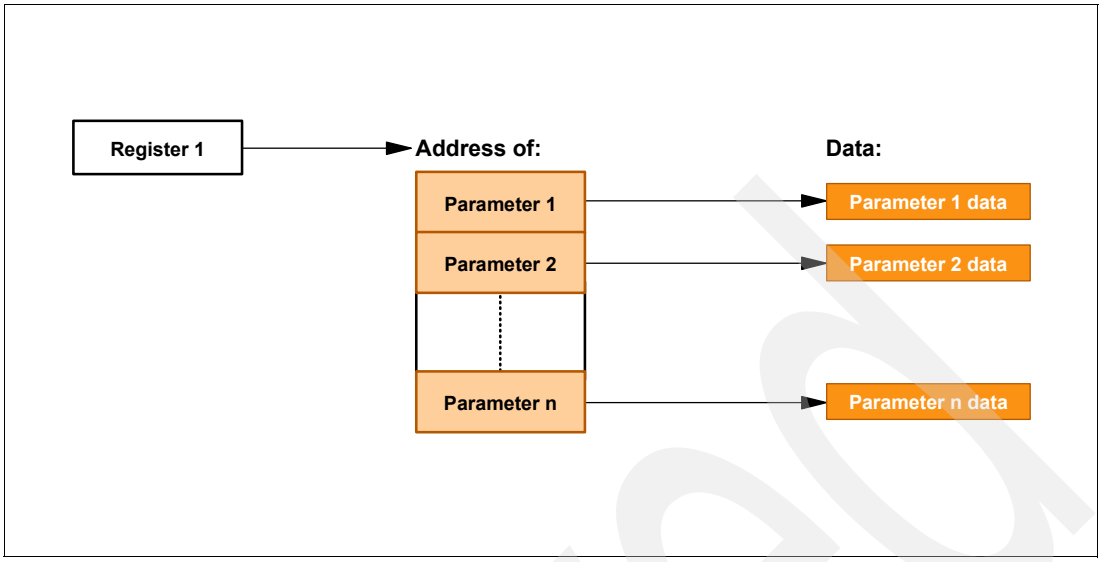

Figure 9-7 Parameter convention GENERAL for a stored procedure

Figure 9-8 shows the structure of the parameter list when PARAMETER STYLE GENERAL WITH NULLS is used.

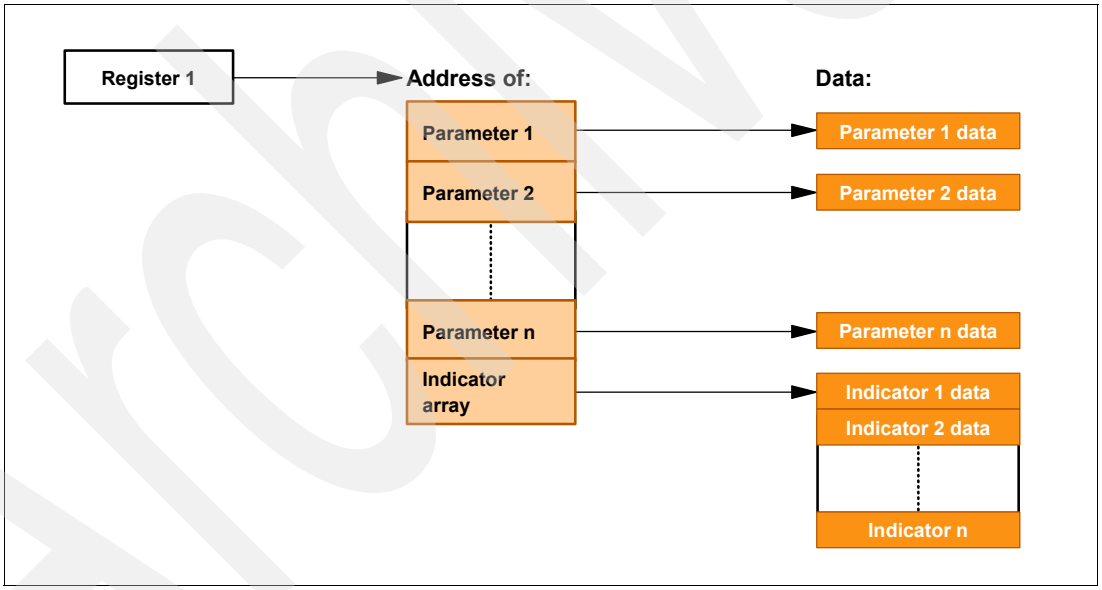

Figure 9-8 Parameter convention GENERAL WITH NULLS for a stored procedure

Figure 9-9 on page 101 shows the structure of the parameter list when PARAMETER STYLE SQL is used.

Important: Remember that when using parameter style SQL, an array of indicator variables is not supported; you must specify an elementary item for each indicator variable.

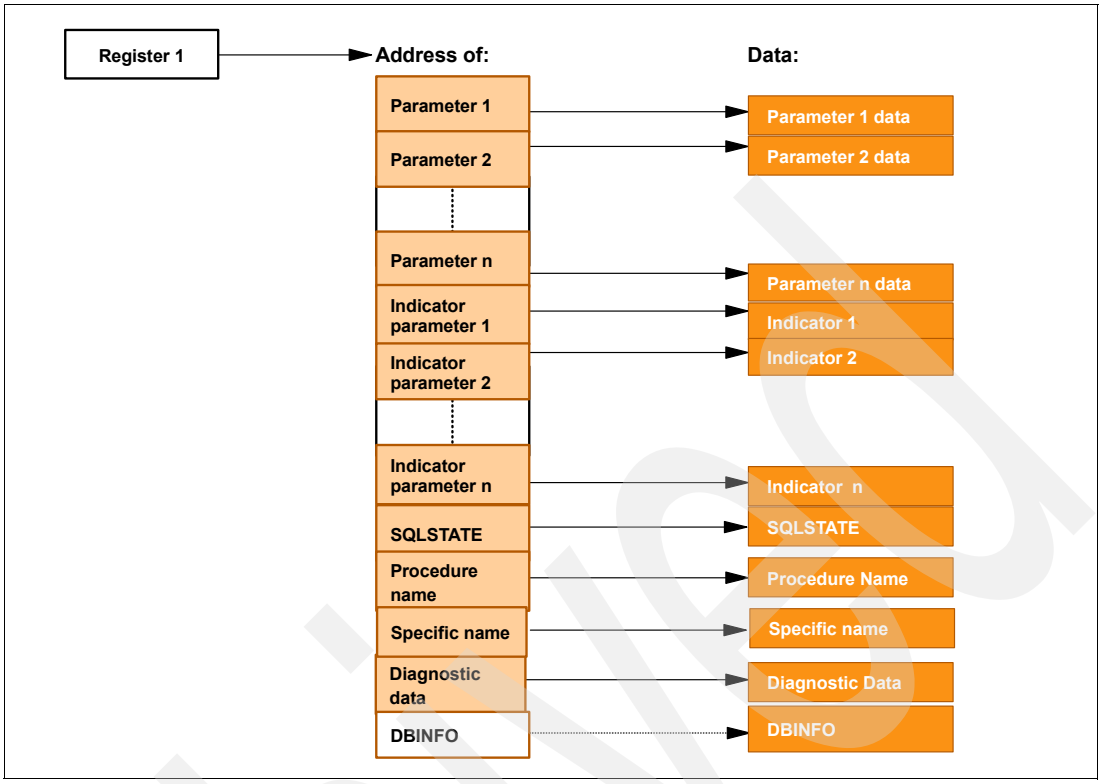

Figure 9-9 Parameter convention SQL for a stored procedure

Figure 9-10 shows the structure of the parameter list when PARAMETER STYLE JAVA is used. The list of ResultSet parameters is optional.

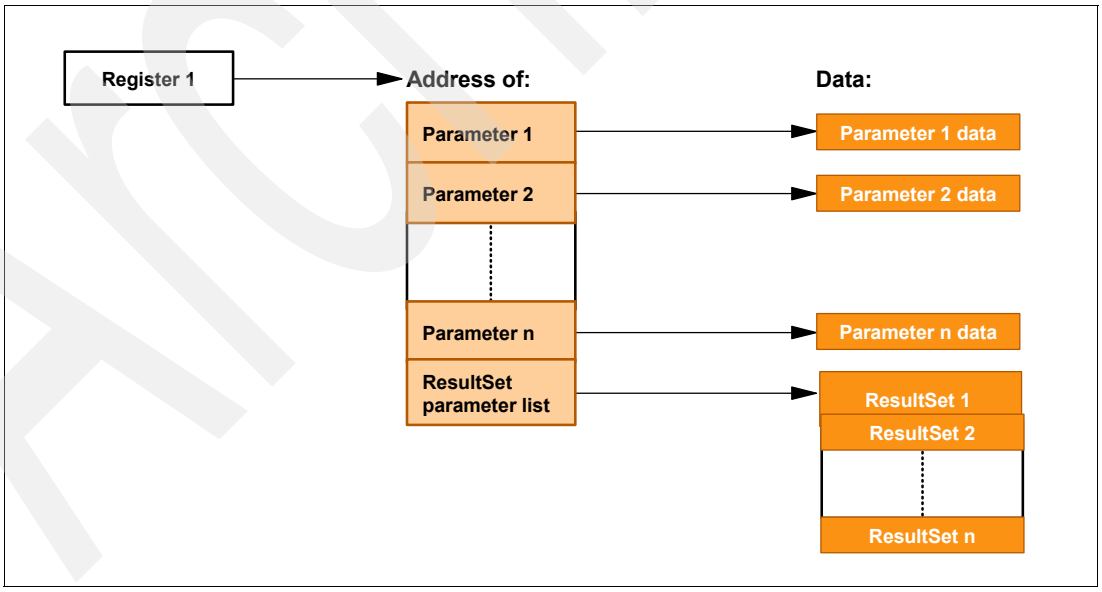

Figure 9-10 Parameter convention JAVA for a stored procedure

9.1.7 Deterministic stored procedures

When the stored procedure is called successively with a set of parameters with the same values, will it return the same result? If this is the case, specify it as `DETERMINISTIC`, otherwise, specify it as `NOT DETERMINISTIC`, which is the default.

A stored procedure that contains SQL can by definition return a different result for each call. Another such example is one where a random number is generated within the stored procedure. In such cases, it should be specified as `NOT DETERMINISTIC`. Only if you are certain that the result will be the same, should you specify `DETERMINISTIC`.

Note that DB2 does not verify that the stored procedure code is consistent with the specification of `DETERMINISTIC` or `NOT DETERMINISTIC`. For example, you can define the stored procedure as `DETERMINISTIC` when in reality its behavior is such that it returns different values when called with a set of identical values as input, and DB2 does not check the logic.

9.1.8 Package path

As of DB2 V9 you can specify PACKAGE PATH. This parameter specifies the package path to use when the procedure is run. This is the list of the possible package collections into which the DBRM that is associated with the procedure is bound. If the procedure definition includes a specification for PACKAGE PATH, DB2 sets CURRENT PACKAGE PATH special register to the value of PACKAGE PATH, independent of the value for SPECIAL REGISTERS.

You may specify

```
NO PACKAGE PATH
```

or

```
PACKAGE PATH package-path
```

NO PACKAGE PATH specifies that the list of package collections for the procedure is the same as the list of package collection IDs for the calling program. If the calling program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order).

PACKAGE PATH specifies a list of package collections, in the same format as the SET CURRENT PACKAGE PATH special register. If the COLLID clause is specified with PACKAGE PATH, the COLLID clause is ignored when the routine is invoked.

9.1.9 Optional caller information

When the stored procedure is called, do you need to pass information about the caller to it? If not, specify `NO DBINFO`. In this case, only the parameters are passed to the stored procedure. If you specify `DBINFO`, DB2 passes an additional argument that is a structure containing information such as the name of the current server, the application runtime authorization ID, and an identification of the version and release of the database manager that invoked the stored procedure. DB2 also passes a unique application ID (a token unique for each execution of the stored procedure) which can be helpful in performance monitoring. If this information is useful to you (for example, if the stored procedure is coded to take different actions depending on who calls it), specify `DBINFO`, otherwise, specify `NO DBINFO`.

`DBINFO` can only be specified if the `PARAMETER STYLE SQL` is specified. `NO DBINFO` is the default.

9.1.10 Collection ID the stored procedure runs in

If the stored procedure contains SQL, DB2 needs to know the collection ID of the package for the stored procedure. You may explicitly specify it in the CREATE PROCEDURE, for example:

```
COLLID DEVL7083
```

or you may specify:

```
NO COLLID
```

In any case, DB2 uses the following method to determine the collection ID *in this order:*

1. For DB2 V8 onwards, DB2 examines the CURRENT PACKAGE PATH special register if set by a stored procedure program. If it contains a value, DB2 uses this as the collection ID for the stored procedure.
2. DB2 examines the CURRENT PACKAGESET special register. If it contains a value set by the stored procedure program, DB2 uses this as the collection for the stored procedure.
3. If there is an explicit package associated with the CREATE PROCEDURE COLLID option, DB2 uses this collection ID for the stored procedure.
4. If the calling application has set the CURRENT PACKAGESET special register, DB2 uses this as the collection ID for the stored procedure. This is new with V8 and it allows a remote caller to determine the search path.
5. If the calling application has a package collection associated with it, DB2 uses it for the stored procedure.
6. DB2 examines the plan of the calling application and uses the list of collection IDs specified in the PKLIST in the specified order. This process is especially resource-intensive for distributed applications where the PKLIST consists of multiple collection IDs since a network request to locate the package in each collection is sent until the package is found. SET CURRENT PACKAGESET eliminates this search.

DB2 keeps track of the collection ID of the caller. Once the control is returned from the stored procedure to the client, DB2 resets the CURRENT PACKAGESET value to the collection ID of the caller. This is particularly useful where application programs are divided into two collections such as ONLINE and BATCH, and stored procedures will be called from both ONLINE and BATCH programs. If `NO COLLID` is specified, then the stored procedure packages need to be bound to both ONLINE and BATCH collection IDs. When you specify `COLLID xxxxx`, the stored procedures can be bound to their own collection ID, independent of the caller.

9.1.11 CPU threshold value

If you want to control the total amount of processor time in CPU service units for a single execution of a stored procedure, you can specify it through the ASUTIME parameter. ASUTIME NO LIMIT means that there is no limit on the service units (this is the default). ASUTIME n (where n is an integer between 1 and 2G) means that DB2 cancels the stored procedure if the stored procedure uses more service units than the specified limit. In this case, DB2 returns SQLCODE -905 (SQLSTATE 57014) to the stored procedure, as shown in Example 9-1.

Example 9-1 Stored procedure exceeding ASUTIME limit

```
UNSUCCESSFUL EXECUTION DUE TO RESOURCE LIMIT BEING EXCEEDED.  RESOURCE NAME =
"DEVL7083.EMPRSETS         ", LIMIT = "0000000001" CPU SECONDS ("000000000002" SERVICE
UNITS) DERIVED FROM "SYSIBM.SYSROUTINES".  SQLSTATE=57014
```

See *z/OS V1R9 MVS Initialization and Tuning Guide,* SA22-7591-05 for information on service units.

We recommend that a reasonable limit be established to handle a stored procedure that loops during testing. In addition, such a limit helps any accidental run-away stored procedures in a production environment.

This limit is independent of the ASUTIME specified in the resource limit facility (RLF), which applies to dynamic SQL only, and is at the statement level. In general, the lower limit applies. Exceeding this limit causes an SQLCODE -905 also, as shown in Example 9-2.

Example 9-2 Dynamic SQL statement exceeding ASUTIME limit

```
UNSUCCESSFUL EXECUTION DUE TO RESOURCE LIMIT
BEING EXCEEDED, RESOURCE NAME = ASUTIME LIMIT = 000000000000 CPU
SECONDS (000000000001 SERVICE UNITS) DERIVED FROM SYSIBM.DSNRLST01
SQLSTATE   = 57014 SQLSTATE RETURN CODE
```

Important: The ASUTIME limit for a stored procedure applies to all users, including those with a SYSADM authority. You cannot have a different limit for different authids. RLF limit does not apply to users with SYSADM authority, and you can specify different limits for different authids.

9.1.12 Stored procedure load module in memory

If you specify `STAY RESIDENT NO` (this is the default), DB2 deletes the load module from memory after the stored procedure ends, and it must be reloaded at the next execution. If you specify `STAY RESIDENT YES`, the load module remains resident in memory after the stored procedure ends. We recommend using `STAY RESIDENT YES` for all frequently run reentrant stored procedures.

Reentrant code in external stored procedures

Whenever possible, prepare your stored procedures to be reentrant. Using reentrant stored procedures can lead to improved performance for the following reasons:

- A reentrant stored procedure does not have to be loaded into storage every time it is called.
- A single copy of the stored procedure can be shared by multiple tasks in the stored procedure's address space. This decreases the amount of virtual storage used for code in the stored procedure's address space. To prepare a stored procedure as reentrant, compile it as reentrant and link-edit it as reentrant and reusable. For instructions on compiling programs to be reentrant, see the appropriate language manual.

To make a reentrant stored procedure remain resident in storage, specify STAY RESIDENT YES in the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

If your stored procedure cannot be reentrant, link-edit it as non-reentrant and non-reusable. The non-reusable attribute prevents multiple tasks from using a single copy of the stored procedure at the same time. A non-reentrant stored procedure must not remain in storage. You therefore need to specify STAY RESIDENT NO in the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

A stored procedure that is compiled and link-edited as reentrant may be defined with either STAY RESIDENT YES or NO. If you plan to have your stored procedure remain resident in storage after execution by specifying STAY RESIDENT YES, then you must also prepare the

stored procedure module as reentrant. If you prepare your stored procedure module as non-reentrant, then you must specify STAY RESIDENT NO. You cannot prepare a stored procedure module as non-reentrant and specify STAY RESIDENT YES. Table 9-1 summarizes the residency and re-entrancy characteristics.

Table 9-1 Re-entrant and resident stored procedures modules

Module	STAY RESIDENT	
	YES	NO
Reentrant	Allowed	Allowed
Non reentrant	Not allowed	Allowed

See also 19.2.5, “Overview of performance knobs” on page 411.

9.1.13 Main program versus subprogram

If you specify `PROGRAM TYPE MAIN`, the stored procedure runs as a main routine. This is the only option for REXX stored procedures. If you specify `PROGRAM TYPE SUB`, the stored procedure runs as a subroutine. This is the only option for Java.

The default program type depends on the language and/or the CURRENT RULES special register as shown here:

- For REXX, the default program type is MAIN.
- For Java, the default program type is SUB.
- For other languages:
  - If CURRENT RULES is DB2, default is MAIN.
  - If CURRENT RULES is STD, default is SUB.

We recommend using PROGRAM TYPE SUB for all languages except REXX, where it is not possible. This eliminates the need for the stored procedure to carry out the initial housekeeping routines at each invocation.

See also Section 19.2.5, “Overview of performance knobs” on page 411.

9.1.14 Security for non-SQL resources

When the stored procedure accesses a non-SQL resource such as an IMS database, what authorization ID should be used by the external security product such as RACF to check the security? This is specified by the SECURITY parameter. These are the possible choices:

DB2 The authorization ID of the stored procedures address space is used to check security; the user running the stored procedure does not need any access to such a resource. This is the default.

USER The authorization ID of the user running the stored procedure is used to check security.

DEFINER The authorization ID of the owner of the stored procedure is used to check security.

Specifying SECURITY DB2 leads to ease of implementation, and may in general be the best option for you.

9.1.15 Max number of failures

Up to DB2 V7, the maximum number of failures of a stored procedure before it is placed in a stopped status, can be controlled only at the subsystem level by the DSNZPARM STORMXAB on installation panel DSNTIPX. As of DB2 V8 you can control the maximum number of failures at the stored procedure level. The possible choices are discussed at 8.5, “Handling application failures” on page 87.

The option you should choose for this parameter is based on various factors, including the following:

- Frequency of execution
- Monitoring and early detection of failures
- Criticality of the stored procedure
- Most likely cause of failure (program logic, resources, data)

You may also want to configure the test environment different from production. For example, a large number of failures can be tolerated in test to eliminate frequent DBA intervention, but a lower limit can be specified in a production environment.

9.1.16 Runtime options

You can specify the Language Environment runtime options to be used for the stored procedure as a character string up to 254 bytes in length. This is an optional parameter, and if you omit it or pass an empty string, DB2 does not pass any runtime options, and Language Environment uses its installation defaults. See 5.2, “Language Environment runtime options” on page 48, and Chapter 25 of *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

We recommend using the following:

```
RUN OPTIONS 'MSGFILE(ddname,,,,ENQ | NOENQ)'
```

The additional option `RPTOPTS(ON)` causes I/O to the JES spool and should be used only for debugging purposes.

Note: Avoid grouping PROGRAM TYPE SUB stored procedures with different runtime options in the same WLM application environment because this can make those stored procedures behave like PROGRAM TYPE MAIN.

9.1.17 Use of commit before returning

Do you want DB2 to commit all work done in the unit of work after the stored procedure completes successfully? If so, specify COMMIT ON RETURN YES. Otherwise, specify `COMMIT ON RETURN NO` (this is the default).

The advantage of `COMMIT ON RETURN YES` occurs primarily in the distributed environment where a client application can otherwise continue to hold locks on the updated DB2 objects. By committing early, you can release the locks earlier. However, be aware of the fact that all work in the unit of work (including work done by the calling program) is committed.

If you specify `COMMIT ON RETURN YES`, and the stored procedure returns result sets, the cursors associated with the result sets must be declared using the WITH HOLD option to be usable after the commit.

We recommend `COMMIT ON RETURN YES` for distributed applications. Nevertheless, we also recommend to always commit at the client application. See Chapter 19, “General performance considerations” on page 391 for details.

We recommend `COMMIT ON RETURN NO` for non-distributed applications.

The `COMMIT ON RETURN` should be `NO` for nested stored procedures also. A stored procedure cannot call other stored procedures defined with `COMMIT ON RETURN YES`.

9.1.18 Values for special registers

You can specify that you want the stored procedure to obtain the values of special registers from the calling program by using the keywords INHERIT SPECIAL REGISTERS. Alternatively, you can specify that you want the stored procedure to obtain default values for special registers by using the DEFAULT SPECIAL REGISTERS keyword.

In some cases, the values can be modified by the SET command.

The section “Using special registers in a stored procedure” in Chapter 25 of *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841 shows the values for each register when using each of these options. Note that in some cases, the default is the same as the value received from the invoker.

9.1.19 Using null parameters

When all parameters for the stored procedure CALL are null, do you want DB2 to invoke the stored procedure? If so, specify CALLED ON NULL INPUT. This is the default. In this case, the stored procedure is responsible for testing for null arguments. If the parameter is not specified, the stored procedure is not called when all input parameters are null.

This parameter should not be confused with the PARAMETER STYLE parameter discussed in 9.1.6, “Passing parameters” on page 99.

Currently, CALLED ON NULL INPUT is the only option, so you do not really have a choice. We recommend specifying it for documentation and to prepare you for any future changes.

9.1.20 WLM environment

WLM ENVIRONMENT name can be specified in two ways

- WLM ENVIRONMENT name

  or

- WLM ENVIRONMENT (name, *)

  When an SQL application program directly calls a procedure, *name* specifies the WLM environment in which the procedure runs. If another procedure or a user-defined function calls the stored procedure, the procedure runs in the same WLM environment that the calling routine uses.

9.1.21 Naming your stored procedure

As of DB2 V8, the name of the stored procedure can be up to 128 characters. An external stored procedure is associated with a load module on a z/OS server. If you do not specify EXTERNAL NAME, EXTERNAL NAME procedure-name is implicit. In some cases, the default name will not be valid. To avoid invalid names, specify EXTERNAL NAME for a

procedure that has a name that is greater than 8 bytes in length, contains an underscore, or does not conform to the rules for an ordinary identifier.

If the stored procedure is a DB2 program, it contains a package. Because there is a one-to-one correspondence between stored procedure name, load module, and DB2 package, it's recommended to use the same name for all of them. The load module cannot be greater than eight characters. Hence restrict the stored procedure name also to eight characters, if your organization naming standards permit. Having the same name helps in identifying the package and load module for a stored procedure.

If the EXTERNAL NAME is specified or defaulted (to the full procedure name), it is checked for a valid MVS load module name, and if invalid SQLCODE -449 is issued.

9.2 Examples of stored procedure definition

In this section, we provide examples of the parameters we used in our case study to define COBOL, REXX, Java, SQL, and C stored procedures.

COBOL stored procedure

See Example 9-3.

Example 9-3 Parameters for COBOL stored procedures CREATE

```
CREATE PROCEDURE DEVL7083.EMPDTL1C
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
RESULT SETS 0
EXTERNAL NAME EMPDTL1C
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB9AWLM
STAY RESIDENT NO
COLLID DEVL7083
PROGRAM TYPE SUB
COMMIT ON RETURN NO ;
```

C stored procedure

See Example 9-4.

Example 9-4 Parameters for C stored procedures CREATE

```
CREATE PROCEDURE DEVL7083.EMPDTL1P
  ( IN EMPNO CHAR(6) CCSID EBCDIC
  , OUT FIRSTNME VARCHAR(12) CCSID EBCDIC
  , OUT MIDINIT CHAR(1) CCSID EBCDIC
```

```
  , OUT LASTNAME VARCHAR(15) CCSID EBCDIC
  , OUT WORKDEPT CHAR(3) CCSID EBCDIC
  , OUT HIREDATE DATE
  , OUT SALARY DEC(9,2)
  , OUT RETCODE INTEGER
  , OUT MESSAGE VARCHAR(1331) CCSID EBCDIC
  )
  RESULT SETS 0
  EXTERNAL NAME EMPDTL1P
  LANGUAGE C
  PARAMETER STYLE GENERAL
  MODIFIES SQL DATA
  WLM ENVIRONMENT DB9AWLM
STAY RESIDENT NO
  COLLID DEVL7083
  PROGRAM TYPE MAIN
  RUN OPTIONS 'TRAP(OFF),STACK(,,ANY,)'
  COMMIT ON RETURN NO
  ASUTIME NO LIMIT;
```

REXX stored procedure

See Example 9-5.

Example 9-5 Parameters for REXX stored procedures CREATE

```
CREATE PROCEDURE DEVL7083.EMPRSETR
(
 IN  PDEPTNO       CHAR(3)
,OUT PARMOUT       VARCHAR(295)
)
RESULT SETS 5
EXTERNAL NAME EMPRSETR
LANGUAGE REXX
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB9AREXX
STAY RESIDENT NO
COLLID DSNREXX
PROGRAM TYPE MAIN
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF)'
COMMIT ON RETURN NO
```

Java stored procedure

See Example 9-6.

Example 9-6 Parameters for Java stored procedures CREATE

```
CREATE PROCEDURE DEVL7083.EMPDTLSJ
( IN EMPNO CHARACTER(6),
OUT FIRSTNAME VARCHAR(12),
OUT MIDINIT CHAR(1),
OUT LASTNAME VARCHAR(15),
OUT WORKDEPT CHAR(3),
OUT SALARY DECIMAL(9,2),
OUT HIREDATE DATE,
OUT OUTPUTMESSAGE VARCHAR(250))
EXTERNAL NAME 'EmpDtlsJ.GetEmpDtls'
LANGUAGE JAVA
```

```
PARAMETER STYLE JAVA
COLLID DSNJDBC
PROGRAM TYPE SUB
WLM ENVIRONMENT DB9AWLMJ
```

SQL language external stored procedure

See Example 9-7.

Example 9-7 Parameters for SQL language external stored procedure CREATE

```
CREATE PROCEDURE DEVL7083.EMPDTLSS
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
DYNAMIC RESULT SETS 0
PARAMETER STYLE GENERAL WITH NULLS
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB9AWLM
STAY RESIDENT NO
COLLID DEVL7083
PROGRAM TYPE MAIN
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF)'
COMMIT ON RETURN NO
LANGUAGE SQL
...
```

Attention: Even though DB2 builds external SQL procedures as external C load modules, you must always specify LANGUAGE SQL in the CREATE PROCEDURE statement. Never redeploy externally the module as a LANGUAGE C stored procedure. Unpredictable errors will happen.

Native SQL language stored procedure

See Example 9-8.

Example 9-8 Parameters for native SQL language stored procedure

```
CREATE PROCEDURE
UPDATE_BALANCE
(
IN CUSTOMER_NO INTEGER
,IN AMOUNT DECIMAL(9,2)
)
VERSION V1
LANGUAGE SQL
READS SQL DATA
BEGIN
DECLARE CUSTOMER_NAME CHAR(20);
SELECT CUSTNAME
```

```
INTO CUSTOMER_NAME
FROM ACCOUNTS
WHERE CUSTNO = CUSTOMER_NO;
END
```

9.3 Multiple versions of stored procedures

DB2 can create and maintain multiple versions of a native SQL stored procedure; see 15.3, “Versioning” on page 291. For external SQL procedures and external procedures, you can manually maintain multiple versions of the procedures.

9.3.1 Multiple versions of external procedures and external SQL procedures

To create multiple versions of external procedures and external SQL procedures, use one of the following techniques:

- Define multiple procedures with the same name in different schemas. You can subsequently use the SQL path to determine which version of the procedure is to be used by a calling program.
- Define multiple versions of the executable code. You can subsequently use a particular version by specifying the name of the load module for the version that you want to use on the EXTERNAL clause of the CREATE PROCEDURE statement or ALTER PROCEDURE statement.
- Define multiple packages for a procedure. You can subsequently use the COLLID option, the CURRENT PACKAGESET special register, or the CURRENT PACKAGE PATH special register to specify which version of the procedure is to be used by the calling application.
- Set up multiple WLM environments to use different versions of a procedure.

For versioning in native SQL procedures, see Section 15.3, “Versioning” on page 291.

9.4 Summary of recommendations

Based on the preceding discussions, our recommendations for the values for the parameters are summarized in Table 9-2.

Table 9-2 Recommended stored procedures parameters

Parameter	Recommended value
DYNAMIC RESULT SETS n	Reasonable maximum keeping in mind that an ALTER PROCEDURE will be needed if this needs to change. The value of n influences the default signature of the Java routine, and (in cases of Java routine overloading) also influences which Java routine is used to execute the External routine.
LANGUAGE	N/A but note dependency of other parameters
NO SQL/ MODIFIES SQL DATA/ READS SQL DATA/ CONTAINS SQL	MODIFIES SQL DATA is the most general and requires no ALTER PROCEDURE if SQL is added later. There are no known performance implications.

Parameter	Recommended value
PARAMETER STYLE	For Java, use Java (required), for SQL it cannot be specified, for REXX use GENERAL unless the stored procedure has many large parameters that can contain nulls, in which case use GENERAL WITH NULLS; for all other languages use SQL
DETERMINISTIC/ NOT DETERMINISTIC	Currently DB2 does not use this information but may in the future. If you know for certain that same result will be returned, use DETERMINISTIC. In all other cases, use NOT DETERMINISTIC
PACKAGE PATH/ NO PACKAGE PATH	Start using PACKAGE PATH instead of COLLID as of DB2 V9
DBINFO/ NO DBINFO	Use DBINFO, except for Java
NO COLLID/ COLLID collection ID	Use a collection ID that corresponds to the schema name used in the CREATE PROCEDURE.
ASUTIME n	A reasonable maximum in service units. Do not specify NO LIMIT under any circumstances.
STAY RESIDENT YES/ STAY RESIDENT NO	STAY RESIDENT YES for better performance (production). STAY RESIDENT NO eliminates need to issue refresh when there is one user only (development).
PROGRAM TYPE SUB/ PROGRAM TYPE MAIN	PROGRAM TYPE SUB for all except REXX which requires MAIN.
SECURITY DB2/ SECURITY USER/ SECURITY DEFINER	SECURITY DB2 eases the administration of security.
STOP AFTER SYSTEM DEFAULT FAILURES/ STOP AFTER n FAILURES/ CONTINUE AFTER FAILURE	Depends on the application. If you specify CONTINUE AFTER FAILURE, make sure you have a monitoring system in place to detect any repeated failures that could impact the system.
RUN OPTIONS	MSGFILE(ENQ) (and RPTOPTS(ON) for test only). It cannot be specified for Java.
COMMIT ON RETURN YES/ COMMIT ON RETURN NO	COMMIT ON RETURN NO for non-distributed applications and COMMIT ON RETURN YES for distributed applications; make sure the application is aware of the impact and also commit at the client application.
INHERIT SPECIAL REGISTERS/ DEFAULT SPECIAL REGISTERS	Depends on the application. If it is using or modifying them.
CALLED ON NULL INPUT	Currently DB2 allows this as the only choice (not specifying this option uses the default, which is again CALLED ON NULL INPUT). We still recommend specifying it for documentation and to prepare for any future changes.

10

COBOL programming

In this chapter we focus on the development of stored procedures in a very traditional and most common language: COBOL. We refer to two simple complete applications developed in COBOL accessing sample tables. The first one for retrieving employee information for a specific employee number, and the second one for retrieving a list of employees for a specific department. We then discuss COBOL subprogram interfaces. We look at examples of nesting stored procedures and compare them to invoking the nested program with a COBOL language. We also discuss COBOL dynamic calls.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions can be found in Appendix B, “Additional material” on page 887.
Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 24 to decide what components are applicable to your environment.

This chapter contains the following:

- Verify the COBOL environment
- Developing COBOL stored procedures
- COBOL subprogram interfaces

10.1 Verify the COBOL environment

Before starting to develop stored procedures, it is important to have a clear understanding of the various steps necessary to define the stored procedures environment. These steps must be verified as completed before a stored procedure can be executed. These steps are covered in detail in other chapters of this book; we simply point to them here for convenience. They are:

1. The WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 39 for details.
2. The LE environment must be set up. See Chapter 5, “Language Environment setup” on page 47 for details.
3. The stored procedure must be defined to DB2. Note in particular that if the stored procedure is designed to return result sets, the maximum number of result sets that can be returned is specified in the definition. See Chapter 9, “Defining stored procedures” on page 91 for details.
4. Develop the stored procedure. This includes the preparation of the stored procedure for execution, including binding a package if it contains SQL.
5. Grant the necessary privileges to the authorization ID of the user that executes the stored procedure. See Chapter 7, “Security and authorization” on page 65 for details.
6. Develop the calling application, if needed.
7. See Chapter 16, “Debugging” on page 313 for details on testing and debugging.

10.2 Developing COBOL stored procedures

In this section we discuss the development of COBOL stored procedures. We describe the following activities:

- Passing parameters
- Preparing and binding a COBOL stored procedure
- Actions for the calling application
- Actions for the stored procedure
- Handling PARAMETER STYLE SQL
- Handling the DBINFO parameter
- Handling result sets in the calling program

10.2.1 Passing parameters

Example 9-3 on page 108 shows our first sample COBOL stored procedure. More examples are available in Appendix B, “Additional material” on page 887.

A stored procedure can receive and send back parameters to the calling application. When the calling application issues an SQL CALL to the stored procedure, DB2 builds a parameter list based on the parameters coded in the SQL call, and the information specified when the stored procedure is initially defined. One of the options on the CREATE PROCEDURE statement is PARAMETER STYLE, which specifies whether or not nulls can be passed as parameters. This is discussed in detail in 9.1, “CREATE or ALTER PROCEDURE parameters” on page 92. When nulls are permitted, the stored procedure and the calling program must take some additional steps. This is discussed in 10.2.5, “Handling null values in parameters” on page 117. In this section we assume that nulls are not permitted.

For a COBOL stored procedure retrieving information about a specific employee, the parameter list specified when defining the stored procedure looks like the contents of Example 10-1.

Example 10-1 COBOL example of CREATE PROCEDURE

```
CREATE PROCEDURE DEVL7083.EMPDTLSC
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
) ...
```

This definition specifies whether the parameter is IN (input to the stored procedure), OUT (output from the stored procedure) or INOUT (input to and output from the stored procedure). It also specifies the data type and size of each parameter. This list must be compatible with the parameter list in the calling application and is shown in Example 10-2.

Example 10-2 Parameter definition of calling application

```
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
       49  PFIRSTNME-LEN  PIC S9(4) COMP.
       49  PFIRSTNME-TEXT PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
    49  PLASTNAME-LEN  PIC S9(4) COMP.
    49  PLASTNAME-TEXT PIC X(15).
01  PWORKDEPT     PIC X(3).
01  PHIREDATE     PIC X(10).
01  PSALARY       PIC S9(7)V9(2) COMP-3.
01  PSQLCODE      PIC S9(9) COMP.
01  PSQLSTATE     PIC X(5).
01  PSQLERRMC.
    49  PSQLERRMC-LEN   PIC S9(4) COMP.
    49  PSQLERRMC-TEXT  PIC X(250).
```

The defined variables must also be compatible with the parameter list defined in the linkage section of the stored procedure, the procedure division **using** statement and with their definition of parameters in the CREATE PROCEDURE statement. For our sample procedure it looks like Example 10-3.

Example 10-3 Parameter definition in the linkage section

```
LINKAGE SECTION.
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
    49  PFIRSTNME-LEN     PIC S9(4) COMP.
    49  PFIRSTNME-TEXT    PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
    49  PLASTNAME-LEN     PIC S9(4) COMP.
    49  PLASTNAME-TEXT    PIC X(15).
```

```
01  PWORKDEPT     PIC X(3).
01  PHIREDATE     PIC X(10).
01  PSALARY       PIC S9(7)V9(2) COMP-3.
01  PSQLCODE      PIC S9(9) COMP.
01  PSQLSTATE     PIC X(5).
01  PSQLERRMC.
    49  PSQLERRMC-LEN     PIC S9(4) COMP.
    49  PSQLERRMC-TEXT    PIC X(250).
```

The procedure division of the stored procedure is shown in Example 10-4.

Example 10-4 Procedure division using the parameters

```
PROCEDURE DIVISION USING PEMPNO, PFIRSTNME, PMIDINIT, PLASTNAME
                   PWORKDEPT, PHIREDATE, PSALARY, PSQLCODE,
                   PSQLSTATE, PSQLERRMC.
```

10.2.2 Preparing and binding a COBOL stored procedure

Complete the following steps for preparing a stored procedure:

- Precompile and compile the application. You can compile COBOL stored procedures with either the DYNAM option or the NODYNAM option. If you use DYNAM, ensure that the correct DB2 language interface module is loaded dynamically by performing one of the following actions:
  - Use the ATTACH(RRSAF) precompiler option
  - Copy the DSNRLI module into a load library that is concatenated in front of the DB2 libraries. Use the member name DSNHLI

  See also “Solution 2: Dynamic invocation of the language interface module” on page 141.
- Link-edit the application using DSNRLI, the language interface module for the Resource Recovery Services attachment facility. You must specify the parameter AMODE(31) when you link-edit it
- Bind the DBRM to DB2 using the command BIND PACKAGE. If you use the ENABLE option of the BIND PACKAGE command to control access to the stored procedure package, you must enable the system connection type of the calling application.
  - The package for the stored procedure need not be bound with the plan for the program that calls it since it runs under the thread for the calling application.
  - The owner of the package that contains the SQL CALL must have the EXECUTE authority on the procedure. See Chapter 7, “Security and authorization” on page 65 for details.
  - The collection ID associated with the stored procedure package must be based on the following rules:
    - If you specify NO PACKAGE PATH and NO COLLID when creating the stored procedure, the package must use the same collection ID as the calling program.
    - If you specify PACKAGE PATH *collection_id1, collection_id2,...* when creating the stored procedure, the stored procedure must use one of these collections.
    - If you specify NO PACKAGE PATH and COLLID *collection_id* when creating the stored procedure, the stored procedure must use this *collection_id*.

    Also, see 9.1.8, “Package path” on page 102 and 9.1.10, “Collection ID the stored procedure runs in” on page 103 for details on the definition of the collection ID.
- Define the stored procedure to DB2. For details, see Chapter 9, “Defining stored procedures” on page 91.

- Use GRANT EXECUTE to authorize the appropriate users to use the stored procedure.

10.2.3 Actions for the calling application

The calling application must initialize all passed parameters declared as INPUT or INOUT before calling the stored procedure. If the calling program is a distributed application, all passed parameters (including those declared as OUT) must be initialized before calling the stored procedure. The SQL CALL is listed in Example 10-5.

Example 10-5 SQL CALL COBOL example

```
EXEC SQL
     CALL EMPDTLSC( :PEMPNO
                   ,:PFIRSTNME
                   ,:PMIDINIT
                   ,:PLASTNAME
                   ,:PWORKDEPT
                   ,:PHIREDATE
                   ,:PSALARY
                   ,:PSQLCODE
                   ,:PSQLSTATE
                   ,:PSQLERRMC
                  )
END-EXEC.
```

If the stored procedure returns a result set, additional processing is required; this is discussed in 10.2.8, “Handling result sets in the calling program” on page 128.

10.2.4 Actions for the stored procedure

The stored procedure behaves just like any subprogram, taking action based on input parameters (if any), and setting the values of the output parameters (if any). If the stored procedure must return a result set, additional processing is required; this is discussed in 10.2.8, “Handling result sets in the calling program” on page 128.

10.2.5 Handling null values in parameters

When the number and size of parameters passed is large, you should consider allowing nulls to reduce transmission times when the stored procedure is called from a distributed environment. While this adds some complexity to the calling application and the stored procedure, there is a substantial reduction in network transmission when the variables are null, and the corresponding parameter does not need to be transmitted. This saving is larger as the number and size of the optional parameters increases. In this section, we discuss how you handle nulls.

In the calling program, you must define an additional set of indicator variables with one for each nullable parameter. In our example, when each parameter is nullable, the working storage of the calling application looks something like Example 10-6.

Example 10-6 Parameter list of calling application when nulls are allowed

```
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
       49  PFIRSTNME-LEN  PIC S9(4) COMP.
       49  PFIRSTNME-TEXT PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
```

```
    49  PLASTNAME-LEN  PIC S9(4) COMP.
    49  PLASTNAME-TEXT PIC X(15).
01  PWORKDEPT     PIC X(3).
01  PHIREDATE     PIC X(10).
01  PSALARY       PIC S9(7)V9(2) COMP-3.
01  PSQLCODE      PIC S9(9) COMP.
01  PSQLSTATE     PIC X(5).
01  PSQLERRMC.
    49  PSQLERRMC-LEN   PIC S9(4) COMP.
    49  PSQLERRMC-TEXT  PIC X(250).
01  NULL-IND-VARS.
    05  PEMPNO-IV     PIC S9(4) COMP.
    05  PFIRSTNME-IV  PIC S9(4) COMP.
    05  PMIDINIT-IV   PIC S9(4) COMP.
    05  PLASTNAME-IV  PIC S9(4) COMP.
    05  PWORKDEPT-IV  PIC S9(4) COMP.
    05  PHIREDATE-IV  PIC S9(4) COMP.
    05  PSALARY-IV    PIC S9(4) COMP.
    05  PSQLCODE-IV   PIC S9(4) COMP.
    05  PSQLSTATE-IV  PIC S9(4) COMP.
    05  PSQLERRMC-IV  PIC S9(4) COMP.
```

The grouping of all null indicator variables shown in NULL-IND-VARS is a good programming practice, although the only requirement is that they be defined in the LINKAGE SECTION. This is not a requirement of the calling program, only of the stored procedure.

The parameter list in the linkage section of the stored procedure must also include the null indicator variables as shown in Example 10-7.

Example 10-7 Parameter list in the linkage section when nulls are allowed

```
LINKAGE SECTION.
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
    49  PFIRSTNME-LEN     PIC S9(4) COMP.
    49  PFIRSTNME-TEXT    PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
    49  PLASTNAME-LEN     PIC S9(4) COMP.
    49  PLASTNAME-TEXT    PIC X(15).
01  PWORKDEPT     PIC X(3).
01  PHIREDATE     PIC X(10).
01  PSALARY       PIC S9(7)V9(2) COMP-3.
01  PSQLCODE      PIC S9(9) COMP.
01  PSQLSTATE     PIC X(5).
01  PSQLERRMC.
    49  PSQLERRMC-LEN     PIC S9(4) COMP.
    49  PSQLERRMC-TEXT    PIC X(250).
01  NULL-IND-VARS.
    05  PEMPNO-IV     PIC S9(4) COMP.
    05  PFIRSTNME-IV  PIC S9(4) COMP.
    05  PMIDINIT-IV   PIC S9(4) COMP.
    05  PLASTNAME-IV  PIC S9(4) COMP.
    05  PWORKDEPT-IV  PIC S9(4) COMP.
    05  PHIREDATE-IV  PIC S9(4) COMP.
    05  PSALARY-IV    PIC S9(4) COMP.
    05  PSQLCODE-IV   PIC S9(4) COMP.
    05  PSQLSTATE-IV  PIC S9(4) COMP.
    05  PSQLERRMC-IV  PIC S9(4) COMP.
```

Unlike the definition in the calling application, the grouping of all null indicator variables shown in NULL-IND-VARS is not only good programming practice; it is a requirement that they be defined as a group in the LINKAGE SECTION. There must be one null indicator per parameter.

The procedure division for the stored procedure must receive the additional parameters shown in Example 10-8.

Example 10-8 Procedure division using the parameters when nulls are allowed

```
PROCEDURE DIVISION USING PEMPNO, PFIRSTNME, PMIDINIT, PLASTNAME
                   PWORKDEPT, PHIREDATE, PSALARY, PSQLCODE,
                   PSQLSTATE, PSQLERRMC, NULL-IND-VARS.
```

The calling program must include the indicator variables in the CALL shown in Example 10-9. The indicators are matched with parameters strictly on a positional basis, not by names.

Example 10-9 SQL CALL COBOL example when nulls are allowed

```
EXEC SQL
     CALL EMPDTLSC( :PEMPNO        :PEMPNO-IV
                   ,:PFIRSTNME     :PFIRSTNME-IV
                   ,:PMIDINIT      :PMIDINIT-IV
                   ,:PLASTNAME     :PLASTNAME-IV
                   ,:PWORKDEPT     :PWORKDEPT-IV
                   ,:PHIREDATE     :PHIREDATE-IV
                   ,:PSALARY       :PSALARY-IV
                   ,:PSQLCODE      :PSQLCODE-IV
                   ,:PSQLSTATE     :PSQLSTATE-IV
                   ,:PSQLERRMC     :PSQLERRMC-IV
                  )
END-EXEC.
```

In summary, do the following to handle nullable parameters:

- Make sure the stored procedure definition allows null parameters.
- In the calling program, declare a set of indicator variables and set their value to 0 if the parameter is not null, and to -1 if parameter is null.
- Include the set of indicator variables in the CALL statement.
- In the stored procedure declare the indicator variables in the linkage section.
- In the stored procedure include the indicator variables in the procedure division **using**.
- In the stored procedure, check for the value of the null indictor to determine if the parameter is null and take appropriate action.
- If you need to set an OUTPUT or INOUT parameter to null, set its indicator variable to -1.

10.2.6 Handling PARAMETER STYLE SQL

When you specify PARAMETER STYLE SQL for a stored procedure, you can specify null values for each parameter as above when you specify GENERAL WITH NULLS. In addition, DB2 passes input and output parameters to the stored procedures that contain the following information:

- SQLSTATE defined as CHAR(5).
- Qualified name of the stored procedure defined as VARCHAR(517).

The fully qualified delimited identifier is made up of identifiers with all double quotes. So, double each double quote character, plus the outer delimiters (4) plus the period (1): 128*2+128*2+4+1=517.

- Specific name of the stored procedure defined as VARCHAR(128). It is the same as the unqualified name.
- SQL diagnostic string defined as VARCHAR(70).

Important: When you use PARAMETER STYLE SQL, be aware of three important code requirements:

- The CREATE PROCEDURE ddl must *not* specify these additional parameters:

```
,OUT DSQLSTATE     CHAR(5)
,OUT DSPNAME       VARCHAR(517)
,OUT DSPECNAME     VARCHAR(128)
,OUT DDIAGMSG      VARCHAR(70)
```

- Define the additional variables in the linkage section of the stored procedure.
- Define the indicator variables as elementary items (when using parameter style GENERAL WITH NULLS they must be part of a group item).

The valid definition is shown in Example 10-10.

Example 10-10 DDL for PARAMETER STYLE SQL

```
CREATE PROCEDURE DEVL7083.EMPDTLSC
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
```

In this case, the working storage of the calling application looks like Example 10-11. Note that this is exactly the same as for parameter style GENERAL WITH NULLS.

Example 10-11 Parameter list of calling application using PARAMETER STYLE SQL

```
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
     49  PFIRSTNME-LEN  PIC S9(4) COMP.
     49  PFIRSTNME-TEXT PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
    49  PLASTNAME-LEN  PIC S9(4) COMP.
    49  PLASTNAME-TEXT PIC X(15).
01  PWORKDEPT     PIC X(3).
01  PHIREDATE     PIC X(10).
01  PSALARY       PIC S9(7)V9(2) COMP-3.
01  PSQLCODE      PIC S9(9) COMP.
01  PSQLSTATE     PIC X(5).
01  PSQLERRMC.
    49  PSQLERRMC-LEN   PIC S9(4) COMP.
```

```
    49  PSQLERRMC-TEXT  PIC X(250).
01  NULL-IND-VARS.
    05  PEMPNO-IV     PIC S9(4) COMP.
    05  PFIRSTNME-IV  PIC S9(4) COMP.
    05  PMIDINIT-IV   PIC S9(4) COMP.
    05  PLASTNAME-IV  PIC S9(4) COMP.
    05  PWORKDEPT-IV  PIC S9(4) COMP.
    05  PHIREDATE-IV  PIC S9(4) COMP.
    05  PSALARY-IV    PIC S9(4) COMP.
    05  PSQLCODE-IV   PIC S9(4) COMP.
    05  PSQLSTATE-IV  PIC S9(4) COMP.
    05  PSQLERRMC-IV  PIC S9(4) COMP.
```

The parameter list in the linkage section of the stored procedure must also include the indicator variables as level 01 (highlighted in the example) shown in Example 10-12.

Example 10-12 Parameter list in the linkage section using PARAMETER STYLE SQL

```
LINKAGE SECTION.
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
    49  PFIRSTNME-LEN     PIC S9(4) COMP.
    49  PFIRSTNME-TEXT    PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
    49  PLASTNAME-LEN     PIC S9(4) COMP.
    49  PLASTNAME-TEXT    PIC X(15).
01  PWORKDEPT     PIC X(3).
01  PHIREDATE     PIC X(10).
01  PSALARY       PIC S9(7)V9(2) COMP-3.
01  PSQLCODE      PIC S9(9) COMP.
01  PSQLSTATE     PIC X(5).
01  PSQLERRMC.
    49  PSQLERRMC-LEN     PIC S9(4) COMP.
    49  PSQLERRMC-TEXT    PIC X(250).
01 PEMPNO-IV     PIC S9(4) COMP.
01 PFIRSTNME-IV  PIC S9(4) COMP.
01 PMIDINIT-IV   PIC S9(4) COMP.
01 PLASTNAME-IV  PIC S9(4) COMP.
01 PWORKDEPT-IV  PIC S9(4) COMP.
01 PHIREDATE-IV  PIC S9(4) COMP.
01 PSALARY-IV    PIC S9(4) COMP.
01 PSQLCODE-IV   PIC S9(4) COMP.
01 PSQLSTATE-IV  PIC S9(4) COMP.
01 PSQLERRMC-IV  PIC S9(4) COMP.
01  DSQLSTATE     PIC X(5).
01  DSPNAME.
    49  DSPNAME-LEN       PIC S9(4) COMP.
    49  DSPNAME-TEXT      PIC X(517).
01  DSPECNAME.
    49  DSPECNAME-LEN     PIC S9(4) COMP.
    49  DSPECNAME-TEXT    PIC X(128).
01  DDIAGMSG.
    49  DDIAGMSG-LEN      PIC S9(4) COMP.
    49  DDIAGMSG-TEXT     PIC X(70).
```

The procedure division for the stored procedure must receive the additional parameters shown in Example 10-13.

Example 10-13 Procedure division using the parameters using PARAMETER STYLE SQL

```
PROCEDURE DIVISION USING PEMPNO, PFIRSTNME, PMIDINIT, PLASTNAME
                   PWORKDEPT, PHIREDATE, PSALARY, PSQLCODE,
                   PSQLSTATE, PSQLERRMC,
                    PEMPNO-IV, PFIRSTNME-IV, PMIDINIT-IV, PLASTNAME-IV, PWORKDEPT-IV,
                   PHIREDATE-IV, PSALARY-IV, PSQLCODE-IV, PSQLSTATE-IV, PSQLERRMC-IV,
                   DSQLSTATE, DSPNAME, DSPECNAME, DDIAGMSG.
```

The calling program must include the indicator variables for all defined parameters, but must not include the parameters associated with parameter style SQL, as shown in Example 10-14.

Example 10-14 SQL CALL COBOL example using PARAMETER STYLE SQL

```
EXEC SQL
     CALL EMPDTLSC( :PEMPNO        :PEMPNO-IV
                   ,:PFIRSTNME     :PFIRSTNME-IV
                   ,:PMIDINIT      :PMIDINIT-IV
                   ,:PLASTNAME     :PLASTNAME-IV
                   ,:PWORKDEPT     :PWORKDEPT-IV
                   ,:PHIREDATE     :PHIREDATE-IV
                   ,:PSALARY       :PSALARY-IV
                   ,:PSQLCODE      :PSQLCODE-IV
                   ,:PSQLSTATE     :PSQLSTATE-IV
                   ,:PSQLERRMC     :PSQLERRMC-IV
                  )
END-EXEC.
```

When using parameter style SQL, the SQLSTATE value you set in the stored procedure before returning to the caller affects the SQLCODE, SQLSTATE, and the diagnostic string passed back to the caller. If DB2 sets the SQLSTATE value (for example, in case of a timeout or deadlock), this value overrides the value set by the stored procedure, and is unconditionally returned to the caller. Table 10-1 shows, for each value or range of values, the corresponding data received by the caller. In these examples the output ERRMC was set to the string +++**ANY MESSAGE**+++. The entries for 38yxx and 385xx are the same.

Table 10-1 Impact of SQLSTATE values set by the stored procedure

Stored procedure sets this SQLSTATE	Caller receives this SQLCODE	Caller receives this SQLSTATE	Sample SQLCA output
00000	0	00000	N/A
01Hxy (e.g. 01H12)	+462	01Hxy (e.g. 01H12)	DSNT404I SQLCODE = 462, WARNING: EXTERNAL FUNCTION OR PROCEDURE EMPDTLSC (SPECIFIC NAME EMPDTLSC) HAS RETURNED A WARNING SQLSTATE, WITH DIAGNOSTIC TEXT **+++ ANY MESSAGE +++** DSNT418I SQLSTATE = 01H12 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -821 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFCCB' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION

Stored procedure sets this SQLSTATE	Caller receives this SQLCODE	Caller receives this SQLSTATE	Sample SQLCA output
02000	-463	39001	DSNT408I SQLCODE = -463, ERROR: EXTERNAL FUNCTION EMPDTLSC (SPECIFIC NAME EMPDTLSC) HAS RETURNED AN INVALID SQLSTATE 02000, WITH DIAGNOSTIC TEXT **+++ ANY MESSAGE +++** DSNT418I SQLSTATE = 39001 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -881 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFC8F' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION
38yxx (y <> 5) (e.g. 38999)	-443	38yxx (e.g. 38999)	DSNT408I SQLCODE = -443, ERROR: EXTERNAL FUNCTION EMPDTLSC (SPECIFIC NAME EMPDTLSC) HAS RETURNED AN ERROR SQLSTATE WITH DIAGNOSTIC TEXT **+++** **ANY MESSAGE +++** DSNT418I SQLSTATE = 38999 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -891 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFC85' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION
385xx (e.g. 38555)	-443	385xx (e.g. 38555)	DSNT408I SQLCODE = -443, ERROR: EXTERNAL FUNCTION EMPDTLSC (SPECIFIC NAME EMPDTLSC) HAS RETURNED AN ERROR SQLSTATE WITH DIAGNOSTIC TEXT **+++** **ANY MESSAGE +++** DSNT418I SQLSTATE = 38555 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -891 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFC85' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION
38001	-487	38001	DSNT408I SQLCODE = -487, ERROR: PROCEDURE EMPDTLSC ATTEMPTED TO EXECUTE AN SQL STATEMENT WHEN THE DEFINITION OF THE FUNCTION OR PROCEDURE DID NOT SPECIFY THIS ACTION DSNT418I SQLSTATE = 38001 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -831 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFCC1' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION

Stored procedure sets this SQLSTATE	Caller receives this SQLCODE	Caller receives this SQLSTATE	Sample SQLCA output
38002	-577	38002	DSNT408I SQLCODE = -577, ERROR: PROCEDURE EMPDTLSC ATTEMPTED TO MODIFY DATA WHEN THE DEFINITION OF THE FUNCTION OR PROCEDURE DID NOT SPECIFY THIS ACTION DSNT418I SQLSTATE = 38002 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -841 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFCB7' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION
38003	-751	38003	DSNT408I SQLCODE = -751, ERROR: PROCEDURE EMPDTLSC (SPECIFIC NAME EMPDTLSC) ATTEMPTED TO EXECUTE AN SQL STATEMENT THAT IS NOT ALLOWED DSNT418I SQLSTATE = 38003 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -861 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFCA3' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION
38004	-579	38004	DSNT408I SQLCODE = -579, ERROR: PROCEDURE EMPDTLSC ATTEMPTED TO READ DATA WHEN THE DEFINITION OF THE FUNCTION OR PROCEDURE DID NOT SPECIFY THIS ACTION DSNT418I SQLSTATE = 38004 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -851 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFCAD' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION
Other (e.g. 21000)	-463	39001	DSNT408I SQLCODE = -463, ERROR: EXTERNAL FUNCTION EMPDTLSC (SPECIFIC NAME EMPDTLSC) HAS RETURNED AN INVALID SQLSTATE 21000, WITH DIAGNOSTIC TEXT **+++ ANY MESSAGE +++** DSNT418I SQLSTATE = 39001 SQLSTATE RETURN CODE DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR DSNT416I SQLERRD = -881 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION DSNT416I SQLERRD = X'FFFFFC8F' X'00000000' X'00000000' X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION

While you have a great amount of flexibility in terms of what SQLSTATE should be set by the stored procedures, we recommend that you keep these things in mind:

- In general you should not set SQLSTATE to a value that can be misinterpreted by the calling application since it may not be aware of the fact that it was set manually instead of by DB2. For example, setting a value to 38002 causes the error text to be:

```
PROCEDURE EMPDTLSC ATTEMPTED TO MODIFY DATA
WHEN THE DEFINITION OF THE FUNCTION OR PROCEDURE DID NOT SPECIFY THIS
ACTION
```

You may then spend valuable resources tracking down the update statements that never existed. We strongly suggest that you report back only the SQLSTATEs you encounter.

- Except for the special cases noted in Table 10-1 above, the SQLCODE returned is `-443` or `-463` for all cases where you specify the SQLSTATE value. Your calling application must be coded to handle these SQLCODEs and interpret the SQLSTATEs.

The rules are different as of DB2 V8: SQLCODE-463 is replaced for Java by SQLCODE -4302, SQLSTATE 38000.

10.2.7 Handling the DBINFO parameter

If you specify the DBINFO parameter when you define a stored procedure to DB2, DB2 passes a structure to the stored procedure that contains environment information. Parameter style SQL is a prerequisite before specifying DBINFO. Because the structure is also used for user-defined functions, some fields in the structure are not populated when calling a stored procedure. We discuss some of the important fields below. For complete details, see Chapter 26 of *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

- Location name

  A 128-byte character field containing the location to which the invoker is currently connected. Example:

  ```
  SP-DBINFO-LOCATION      =   DB9A
  ```

- Authorization

  A 128-byte character field containing the authorization ID of the application from which the stored procedure is invoked. If nested, this contains the authorization ID of the highest level routine. Example:

  ```
  SP-DBINFO-AUTHORIZATION =   PAOLOR6
  ```

  If the authorization ID field of DBINFO is set to the invoker of a stored procedure, then the field can be a role.

- Subsystem code page

  A 48-byte structure that consists of 10 integer fields and an eight-byte reserved area. These fields provide information about the CCSIDs of the subsystem from which the stored procedure is invoked.

- Product information

  An 8-byte character field containing the product on which the stored procedure executes. It is in the form pppvvrrm, where:

  – ppp is a 3-byte product code:

    ARI DB2 server for VSE and VM

    DSN DB2 UDB for z/OS

    QSQ DB2 UDB for iSeries®

    SQL DB2 UDB for UNIX, Windows and Linux

  – vv is a two-digit version identifier.

  – rr is a two-digit release identifier.

  – m is a one-digit modification level identifier

Example:

```
SP-DBINFO-VERREL        = DSN09015
```

- Operating system

  A 4-byte integer field that identifies the operating system on which the invoking program runs. The value is one of these:

0	Unknown
1	OS/2®
3	Windows
4	AIX®
5	Windows NT®
6	HP-UX
7	Solaris™
8	z/OS
13	Siemens Nixdorf
15	Windows 95
16	SCO UNIX
18	Linux
19	DYNIX/ptx®
24	Linux for S/390®
25	Linux for System z™
26	Linux/IA64 27 Linux/PPC
28	Linux/PPC64
29	Linux/AMD64
400	iSeries

  Example:

  ```
  SP-DBINFO-PLATFORM      = 000000008
  ```

- Unique application identifier

  This field is a pointer to a string that uniquely identifies the application's connection to DB2. The string is regenerated at each connection to DB2.

  The string is the LUWID, which consists of a fully-qualified LU network name followed by a period, and an LUW instance number. The LU network name consists of a one- to eight-character network ID, a period and a one- to eight-character network LU name. The LUW instance number consists of 12 hexadecimal characters that *uniquely identify the unit of work*. Example (when called by the IBM Data Studio):

  ```
  APPLICATION-ID=G91E1CAF.G977.C16F2D1D098B
  ```

  or when called by a local application:

  ```
  APPLICATION-ID= USIBMSC.SCPDB9A.C16F2F12B63A
  ```

The calling program does not change.

The stored procedure definition must include these parameters shown below:

```
PARAMETER STYLE SQL
...
```

DBINFO

The parameter list in the linkage section of the stored procedure must also include these additional variables, as shown in Example 10-15.

Example 10-15 Parameter list in the linkage section using DBINFO

```
LINKAGE SECTION.
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
    49  PFIRSTNME-LEN     PIC S9(4) COMP.
    49  PFIRSTNME-TEXT    PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
    49  PLASTNAME-LEN     PIC S9(4) COMP.
    49  PLASTNAME-TEXT    PIC X(15).
01  PWORKDEPT     PIC X(3).
01  PHIREDATE     PIC X(10).
01  PSALARY       PIC S9(7)V9(2) COMP-3.
01  PSQLCODE      PIC S9(9) COMP.
01  PSQLSTATE     PIC X(5).
01  PSQLERRMC.
    49  PSQLERRMC-LEN     PIC S9(4) COMP.
    49  PSQLERRMC-TEXT    PIC X(250).
01  PEMPNO-IV     PIC S9(4) COMP.
01  PFIRSTNME-IV  PIC S9(4) COMP.
01  PMIDINIT-IV   PIC S9(4) COMP.
01  PLASTNAME-IV  PIC S9(4) COMP.
01  PWORKDEPT-IV  PIC S9(4) COMP.
01  PHIREDATE-IV  PIC S9(4) COMP.
01  PSALARY-IV    PIC S9(4) COMP.
01  PSQLCODE-IV   PIC S9(4) COMP.
01  PSQLSTATE-IV  PIC S9(4) COMP.
01  PSQLERRMC-IV  PIC S9(4) COMP.
01  D5    PIC X(5).
01  D27.
    49  D27-LEN           PIC S9(4) COMP.
    49  D27-TEXT          PIC X(517).
01  D18.
    49  D18-LEN           PIC S9(4) COMP.
    49  D18-TEXT          PIC X(128).
01  D70.
    49  D70-LEN           PIC S9(4) COMP.
    49  D70-TEXT          PIC X(70).
01 SP-DBINFO.
*      LOCATION LENGTH AND NAME
     02 SP-DBINFO-LOCATION.
       49 SP-DBINFO-LLEN PIC 9(4) USAGE BINARY.
       49 SP-DBINFO-LOC   PIC X(128).
*      AUTHORIZATION ID LENGTH AND NAME
     02 SP-DBINFO-AUTHORIZATION.
       49 SP-DBINFO-ALEN PIC 9(4) USAGE BINARY.
       49 SP-DBINFO-AUTH PIC X(128).
*       CCSIDS FOR DB2 FOR OS/390
     02 SP-DBINFO-CCSID PIC X(48).
     02 SP-DBINFO-CCSID-REDEFINE REDEFINES SP-DBINFO-CCSID.
        03 SP-DBINFO-ASBCS    PIC 9(9) USAGE BINARY.
        03 SP-DBINFO-ADBCS    PIC 9(9) USAGE BINARY.
        03 SP-DBINFO-AMIXED   PIC 9(9) USAGE BINARY.
        03 SP-DBINFO-ESBCS    PIC 9(9) USAGE BINARY.
        03 SP-DBINFO-EDBCS    PIC 9(9) USAGE BINARY.
```

```
           03 SP-DBINFO-EMIXED   PIC 9(9) USAGE BINARY.
           03 SP-DBINFO-ENCODE   PIC 9(9) USAGE BINARY.
           03 SP-DBINFO-RESERVO PIC X(20).
*          SCHEMA LENGTH AND NAME
        02 SP-DBINFO-SCHEMAO.
           49 SP-DBINFO-SLEN PIC 9(4) USAGE BINARY.
           49 SP-DBINFO-SCHEMA PIC X(128).
*          TABLE LENGTH AND NAME
        02 SP-DBINFO-TABLEO.
           49 SP-DBINFO-TLEN PIC 9(4) USAGE BINARY.
           49 SP-DBINFO-TABLE   PIC X(128).
*          COLUMN LENGTH AND NAME
        02 SP-DBINFO-COLUMNO.
           49 SP-DBINFO-CLEN PIC 9(4) USAGE BINARY.
           49 SP-DBINFO-COLUMN   PIC X(128).
*          DB2 RELEASE LEVEL
        02 SP-DBINFO-VERREL PIC X(8).
*          UNUSED
        02 FILLER            PIC X(2).
*          DATABASE PLATFORM
        02 SP-DBINFO-PLATFORM PIC 9(9) USAGE BINARY.
*          # OF ENTRIES IN TABLE FUNCTION COLUMN LIST
        02 SP-DBINFO-NUMTFCOL PIC 9(4) USAGE BINARY.
*          RESERVED
        02 SP-DBINFO-RESERV1 PIC X(24).
*          UNUSED
        02 FILLER            PIC X(2).
*          POINTER TO TABLE FUNCTION COLUMN LIST
        02 SP-DBINFO-TFCOLUMN PIC 9(9) USAGE BINARY.
*          POINTER TO APPLICATION ID
        02 SP-DBINFO-APPLID   USAGE POINTER.
*          RESERVED
        02 SP-DBINFO-RESERV2 PIC X(20).
     01 APPLICATION-ID        PIC X(32).
```

10.2.8 Handling result sets in the calling program

When the stored procedure returns a small amount of data that does not contain repeating groups (for example, information about an employee), it is much simpler to avoid result sets altogether, returning the data as parameters as discussed above. When the stored procedure must return result sets, each consisting of multiple rows (for example, information about all employees in a department), there are two possibilities:

- The number of result sets is fixed, and you know the contents.
- The number of result sets is variable, and you do not know the contents.

Handling the first case is simpler to develop, but the second case is more general and requires minimal modifications if the calling program or stored procedure happens to change. We discuss the two alternatives in this section.

The following steps are required to handle result sets:

1. When defining the stored procedure to DB2 (see 9.1.3, “Number of returned result sets” on page 98), specify the maximum number of result sets that could be generated by the stored procedure.
2. In the stored procedure, declare and open a cursor for each result set. Note that the stored procedure must not fetch rows from the cursor nor close the cursor. You must declare each such cursor using the WITH RETURN clause. If the stored procedure is created

using the COMMIT ON RETURN option (see 9.1.17, “Use of commit before returning” on page 106 for details), the cursor must also be declared using the WITH HOLD clause to prevent it from being closed when control returns to the calling program.

3. In the calling program, declare a locator variable for each result set that will be returned. If you do not know how many result sets will be returned, declare enough result set locators for the maximum number possible. An example follows:

```
01 LOC-EMPRSETC USAGE SQL TYPE IS
   RESULT-SET-LOCATOR VARYING.
```

4. In the calling program, call the stored procedure and check the return code. If the SQLCODE is +466 (SQLSTATE is 0100C), the stored procedure has returned result sets.
5. If you already know how many result sets the stored procedure returns, go to step 6. Otherwise, issue a DECRIBE PROCEDURE statement as the following example shows. Note that SQLDA is a structure that contains a set of variables, each set corresponding to a cursor that returned a result set. For more information on the SQLDA structure and use, refer to *DB2 Version 9.1 for z/OS Application Programming and SQL Guide*, SC18-9841.

```
EXEC SQL
DESCRIBE PROCEDURE EMPRSETC INTO :PSQLDA
END-EXEC.
```

At this point, assuming the stored procedure has opened two cursors called C1 and MYCURSOR with the return, the SQLDA looks like what is shown in Figure 10-1.

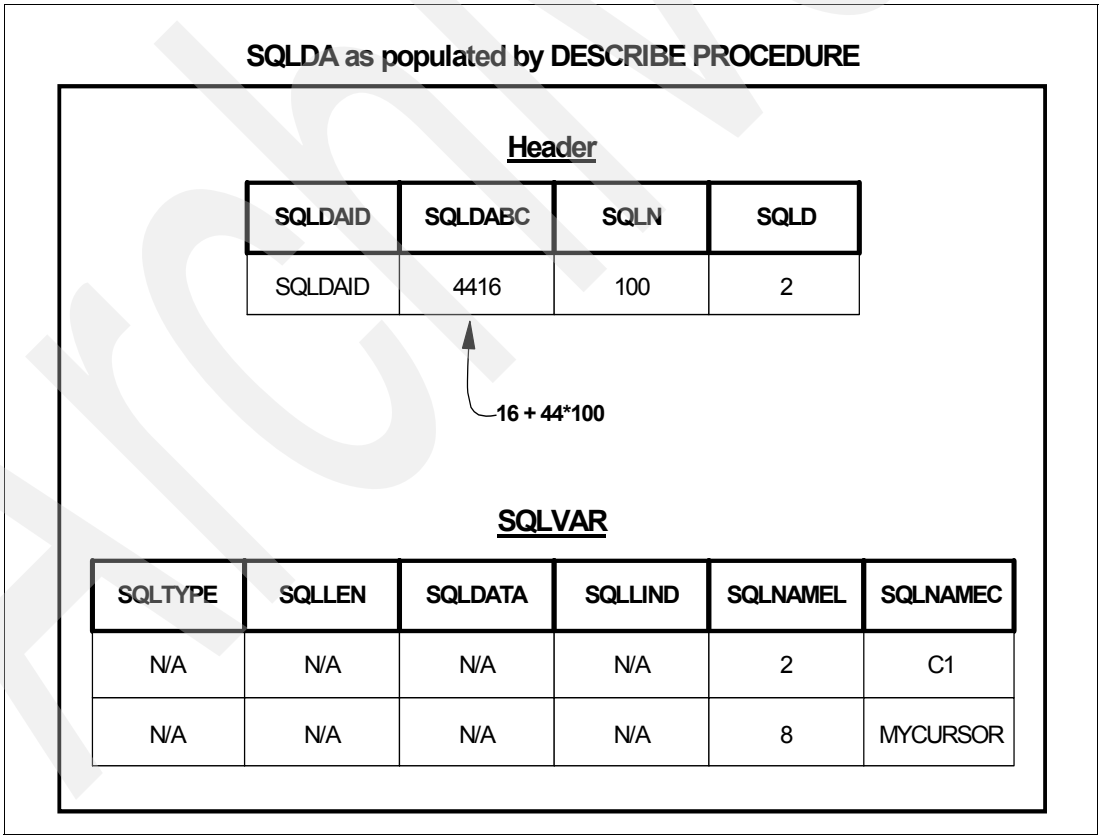

Figure 10-1 SQLDA as populated by the DESCRIBE PROCEDURE statement

6. SQLD contains the number of result sets returned by the stored procedure.
7. SQLNAMEL specifies the length of the cursor name field SQLNAMEC. SQLNAMEC contains the name of the cursor in the stored procedure that returned the result set.

8. Link the result set locators to the result sets:

```
EXEC SQL ASSOCIATE LOCATORS (:LOC-EMPRSETC)
     WITH PROCEDURE EMPRSETC
END-EXEC.
```

9. Allocate a cursor for each result set to be processed:

```
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET
    :LOC-EMPRSETC
END-EXEC.
```

10. Determine the contents of the result sets. If you already know the format of the result set, go to step 11. Otherwise, issue the following:

```
EXEC SQL
DESCRIBE CUSROR C1
INTO :SQLDA
```

11. The SQLDA must be large enough to hold the descriptions of all columns in the result set.
12. Fetch and process all rows from the cursors. This process is similar to processing any normal cursor, except that the cursor has already been opened by the stored procedure. If the cursor is declared as scrollable, fetch operations such as FETCH LAST, FETCH RELATIVE *n* are possible in the calling application.
13. Close the cursor after receiving an SQLCODE of +100 that indicates you have fetched all the rows. Closing the cursor is a good practice to ensure that resources are freed and threads are reusable.

10.3 COBOL subprogram interfaces

In this section we discuss COBOL subprogram interfaces. We look at examples of nesting stored procedures and compare them to invoking the subprogram with the COBOL language. In our examples, the COBOL CALL statement always calls a separate subprogram, but it can also call a nested COBOL program within the same compilation unit. This term *nested* in COBOL, though not a commonly used feature, refers to coding subprograms within (contained in) COBOL programs. We also discuss COBOL dynamic calls.

This section contains the following:

- Nested stored procedures
- COBOL subprograms
- Hybrid approach for optimization
- Summary

10.3.1 Nested stored procedures

Note: The description of nested stored procedures in this section is applicable to most languages, not just COBOL.

A program that is executing as a stored procedure, a user-defined function, or a trigger can issue a CALL statement. When a stored procedure, user-defined function, or trigger calls a stored procedure, user-defined function, or trigger, the call is considered to be nested. Stored procedures, user-defined functions, and triggers can be nested up to 16 levels deep on a single system. Nesting can occur within a single DB2 subsystem, or when a stored procedure or user-defined function is invoked at a remote server.

If a stored procedure returns any query result sets, the result sets are returned to the caller of the stored procedure. If the SQL CALL statement is nested, the result sets are visible only to the program that is at the previous nesting level. For example, Figure 10-2 illustrates a scenario in which a client program calls stored procedure PROCA, which in turn calls stored procedure PROCB. Only PROCA can access any result sets that PROCB returns; the client program has no access to the query result sets. The number of query result sets that PROCB returns does not count toward the maximum number of query results that PROCA can return.

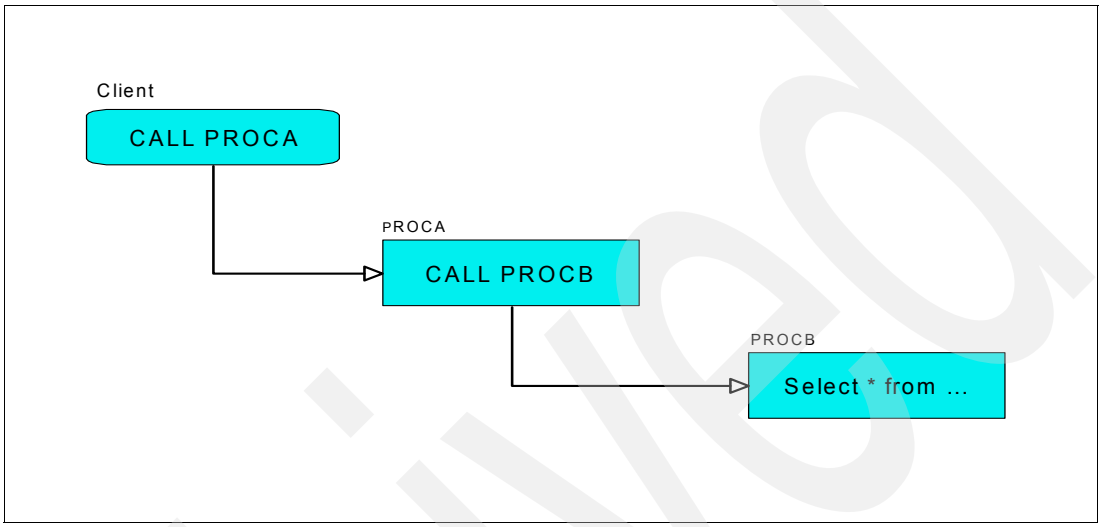

Figure 10-2 Nested stored procedures

Some stored procedures cannot be nested. A stored procedure, user-defined function, or trigger cannot call a stored procedure that is defined with the COMMIT ON RETURN attribute. A stored procedure can call another stored procedure only if they execute in the same type of address space; they must both execute in a WLM-established address space.

DB2 V8 considerations for nested stored procedures

DB2 V8 behaves differently from previous versions in case of *multiple calls to the same stored procedure.*

If the server and requester are both Version 8 of DB2 UDB for z/OS (running in new-function mode), you can call a stored procedure multiple times within an application and at the same nesting level. DB2 is now capable of distinguishing each running instance created by each call to the same stored procedure. If the stored procedure returns result sets, each instance of the stored procedure opens its own set of result set cursors.

The application might receive a resource unavailable message if the CALL statement causes the values of the maximum number of active stored procedures or maximum number open cursors to be exceeded. The value of field MAX STORED PROCEDURES (on installation panel DSNTIPX) defines the maximum number of active stored procedures that are allowed per thread (reset at commit). The value of field MAX OPEN CURSORS (on installation panel DSNTIPX) defines the maximum number of open cursors (both result set cursors and regular cursors) that are allowed per thread.

If you make multiple calls to the same stored procedure within an application, be aware of the following considerations:

- A DESCRIBE PROCEDURE statement describes the last instance of the stored procedure.
- The ASSOCIATE LOCATOR statement works on the last instance of the stored procedure. You should issue an ASSOCIATE LOCATOR statement after each call to the stored procedure to provide a unique locator value for each result set.
- The ALLOCATE CURSOR statement must specify a unique cursor name for the result set of each instance of the stored procedure. Otherwise, you will lose the data from the results sets that are returned from prior instances or calls to the stored procedure.

One of the benefits of stored procedures is their reusability. Stored procedures, once developed, can be called from anywhere and hence contributes to the reusability. Nested stored procedures still increase this benefit. Nested stored procedures are available at no extra network cost, but extra cost in terms of CPU. This is not an abnormal behavior. Each execution of stored procedure involves some amount of CPU cost for the scheduling. If a transaction contains multiple stored procedures at same level or nested stored procedures, this extra cost may become significant especially if the actual cost to execute SQL inside the stored procedure is less.

Another bottleneck with nested procedures is “queuing.” As explained earlier, every request to execute a stored procedure should go through the WLM queue and wait for its turn. So, for a multi-level nested stored procedure, the elapsed time to complete a transaction increases. So, simple nested stored procedures may cause some CPU overhead and elongated response time. This may be an issue for few organizations but not too many as it depends on the service level agreement (SLA) of the application between you and your customer. Your organization should take into consideration the benefits of the nested stored procedure, and the slight overhead associated with them while designing the application and levels of nesting. The queuing can be tuned by setting proper performance goals and/or assigning the “right” number of TCBs to each WLM application environments.

Currently, there are no hard and fast rules for setting the NUMTCB parameter. In this book we provide some guidelines based on the experience of different customers. See 20.1.2, “NUMTCB” on page 425 for details.

In an informal test, we prepared a non-DB2 stored procedure and non-DB2 subprogram. We ran tests to compare the cost of executing an SQL CALL statement with the COBOL CALL statement, and it is approximately 90% more. But this case is purely an ideal one. In more practical situations, the stored procedures contain some SQL statements, and when you compare the cost of the total transaction with the cost of just the CALL statement, the cost of the CALL might be negligible.

For more information on instrumentation, refer to 20.2, “Managing server address spaces” on page 429.

The discussion so far in this section applies to all types of stored procedures. For users of COBOL stored procedures, there is an alternative for nested stored procedures. As most of the legacy applications on z/OS are built with the COBOL language, the following sections provide a discussion on alternatives available for COBOL users.

10.3.2 COBOL subprograms

Stored procedures are designed with two main concepts in mind: reduce network traffic and create reusable components. For distributed applications, network traffic is a consideration. But for local applications (native to z/OS), the cost of network traffic is negligible and it makes sense to either use embedded SQL (if you do not want the same logic from multiple programs) or use a subprogram (if you want to use same logic again and again from multiple programs). COBOL provides a CALL statement to call another program within the same run unit. For sites where the overhead associated with scheduling is *significant*, a COBOL CALL statement can be used as an alternative. The program containing the CALL statement is the calling program whereas the program identified in the CALL statement is the subprogram. Called programs can contain CALL statements. For complete descriptions and use of COBOL subprograms, refer to the following COBOL manuals:

- *Enterprise COBOL for z/OS Programming Guide Version 3 Release 4*, SC27-1412-05
- *Enterprise COBOL for z/OS Language Reference Version 3 Release 4*, SC27-1408-04

How can COBOL subprograms solve the performance issues with stored procedures?

As explained earlier, SQL CALL incurs some overhead associated with the scheduling of stored procedures. It may also experience elongated response time due to the queuing within WLM. The COBOL CALL statement overcomes these two issues without compromising functionality and performance.

Differences between subprograms and stored procedures

The effort to develop a COBOL program to be used either as a stored procedure or a subprogram will be the same. For stored procedures, additionally, DDL has to be created to define the stored procedure to the DB2 subsystem. The difference lies in the way you invoke them; see Table 10-2. Stored procedures (irrespective of the type) can be invoked by an SQL CALL statement and COBOL subprograms can be invoked by a COBOL CALL statement.

Table 10-2 Main differences between COBOL stored procedures and subprograms

Criteria	Stored procedures	Subprograms
Development effort	Same or more compared to subprograms as additional component DDL has to be created for stored procedure definitions.	Same or less compared to stored procedures. Additional parameters needed to compensate for SQL and DB2INFO information.

Criteria	Stored procedures	Subprograms
Ability to call from anywhere	Possible. Generally should not be invoked by batch programs instead of local subroutines, as this causes large overhead.	Not possible to call from distributed or remote applications. But possible to call from local subsystems like CICS or batch.
Code reusability	Yes	Yes
Security	Can control through “execution” privilege on the procedure as well as on the DB2 package.	Can control through “execution” privilege on the DB2 package.
Nested activity	16 levels are allowed. Each level of nested activity requires some additional cost in scheduling.	Allowed without additional cost. There is no imposed limit on the number of CALLs or CALL levels using the COBOL CALL statement.
Execution environment	All stored procedures execute from WLM established stored procedure address spaces (WLM SPAS).	Executes in native address space like TSO or CICS or WLM SPAS depending on where the call to subprogram is originated.
TCB consumption	More, as each execution of a stored procedure in a run unit requires a TCB. So a nested stored procedure requires more than one TCB depending on the levels of nesting.	All subprograms within a run unit require just one TCB.
Result sets	Supports, the caller can fetch result sets from stored procedures.	Not supported; the caller cannot fetch result sets from a subprogram[a].

a. See the discussion on using temporary tables to overcome this in the following sections.

As shown in Table 10-2, the program preparation part of COBOL subprograms will be more or less the same as COBOL stored procedures. They exhibit different behavior during runtime due to the underlying architecture. COBOL subprograms provide an alternative to nested stored procedures. An SQL CALL within a stored procedure can be replaced by a COBOL CALL to provide the benefits of reusability. This technique improves the performance of nested stored procedures by eliminating the wait time in scheduling.

Example 10-16 shows the differences in the way subprograms are invoked compared to stored procedures.

Replacing the SQL call with the COBOL call is not a solution for every performance issue associated with nested stored procedures. However, under certain conditions it improves the performance by eliminating wait time with scheduling. There are some exceptions and some special considerations to be followed to replace an SQL call with a COBOL call, which are described in 10.3.3, “Hybrid approach for optimization” on page 135.

Example 10-16 Invocation of stored procedure and subprogram

```
Stored Procedure:

       EXEC SQL
            CALL PROC1 (parameter_list)
       END-EXEC.

Subprogram: Static call

       CALL ‘PROC1’ USING parameter_list
```

```
Subprogram: Dynamic call

        MOVE 'PROC1' TO WS-PGM-NAME.
        CALL WS-PGM-NAME USING parameter_list
```

Dynamic versus static call

COBOL subprogram calls can be either dynamic or static. A dynamic call resolves the name of the subprogram during execution whereas a static call resolves the subprogram name during link-edit time.

The main differences between dynamic call and static call are summarized in Table 10-3.

Table 10-3 Main differences between COBOL static call versus dynamic call

Feature	Program with static call	Program with dynamic call
Link-edit	All subprograms should be link-edited to the main program.	None of the subprograms need to be link-edited with main program.
Load module size	Big, relatively, because it contains all load modules of the subprograms.	No difference
Performance	Performs better because all load modules are loaded into memory in one pass	Slight performance overhead because at each call to subprogram requires the subprogram to be located and loaded into memory
Flexibility	Not flexible, because whenever a subprogram is compiled, the main program also needs to be link-edited[a].	Flexible, because compilation of the subprogram does not require the main program to be link-edited.

a. If there is logic change between main program and subprogram, then both need to be compiled and link-edited.

As shown above, it is recommended to use dynamic calls because these allow flexibility and provide similar benefits as stored procedures in terms of maintenance.

10.3.3 Hybrid approach for optimization

It is not unusual to have multiple components like CICS, batch, and distributed/remote for an application.

If your application experiences lengthened response times and more wait time in scheduling stored procedures (long accounting report will show this), then the following *recommendations* can be implemented:

- Use COBOL CALL instead of SQL CALL in all stored procedures to avoid nested stored procedure activity.
- Use stored procedures (SQL CALL) only in distributed applications. For local applications (CICS, batch) use subprograms (COBOL CALL).

When we recommend to use stored procedures for remote applications and subprograms for local applications, we do not mean to maintain two versions of the same program, one as a stored procedure and another as a subprogram. Our intention is to maintain one single program and invoke it differently. Detailed program preparation and invocation examples are shown in the following sections on this topic.

Recommendation 1: COBOL CALL instead of SQL CALL

By implementing this recommendation, you can avoid nested stored procedures. As shown in Figure 10-3, with nested subprograms, the client invokes the outermost program as a stored procedure. Afterwards, the inner levels will be subprogram calls.

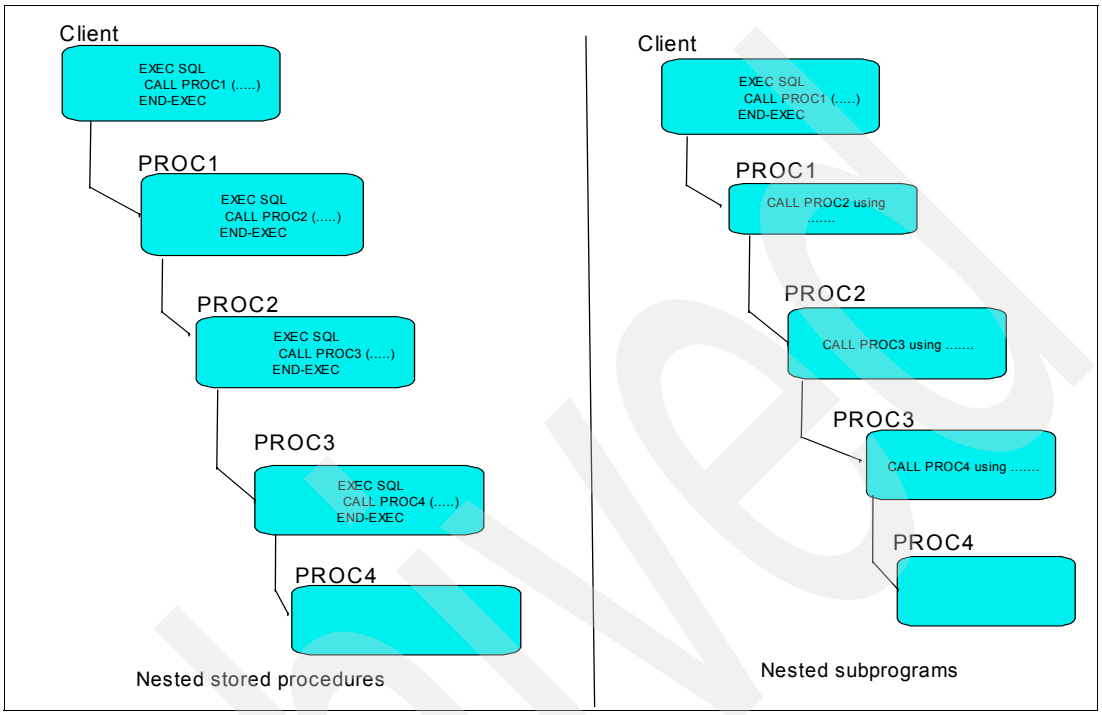

Figure 10-3 Nested stored procedure versus nested subprograms

For example, your application ABC contains four levels of nesting:

```
PROC1 --> PROC2 --> PROC3 --> PROC4.
```

Remote application ABC has a requirement to invoke PROC1 and go through the nested activity until PROC4. As per our recommendation, PROC1 will be called as a stored procedure, and all other inner level calls will be subprograms. After few months, let us say some other remote application DEF has a requirement to invoke PROC3, it still can call PROC3 as a stored procedure. Since application ABC accessed PROC3 as a subprogram through PROC2, DEF does not have to invoke PROC3 as a subprogram. It still can be accessed as a stored procedure. This is the advantage of this technique: No compromise of reusability. The same rule applies to all programs. The way you invoke matters, not the way you prepared the stored procedure or subprogram.

Preparation

These are the steps:

1. Develop the programs and ensure that subprogram calls are made to PROC2, PROC3, and PROC4 from their higher level programs. For example:

```
PROC1 code contains
   CALL PROC2 using .........
PROC2 code contains
   CALL PROC3 using .........
PROC3 code contains
   CALL PROC4 using .........
```

2. Precompile and compile the programs.
3. Link-edit with DSNRLI.

4. Bind the DBRMs to produce DB2 packages.
5. Refresh WLM SPAS.
6. Define PROC1, PROC2, PROC3, and PROC4 as stored procedures.

Invocation

From application ABC, invoke PROC1:

```
EXEC SQL
CALL PROC1 (..............)
END-EXEC.
```

From application DEF, invoke PROC3:

```
EXEC SQL
CALL PROC3 (..............)
END-EXEC.
```

Special considerations

However, as shown in Table 10-2, COBOL subprograms cannot return result sets. So, if you design your stored procedure to return the result, the same stored procedure cannot be replaced with the subprogram. This is a known restriction.

To overcome the restriction of result sets with subprograms, DB2 temporary tables can be used. Throughout this chapter, our intention is to provide alternatives to nested stored procedures without compromising the benefits of their reusability. Table 10-4 shows how result sets can be used with subprograms.

Table 10-4 Handling result sets, COBOL stored procedures versus subprograms

Stored procedure	Subprogram
Stored procedure: 1. Declare a cursor WITH RETURN. 2. Open the cursor. Caller (with SQL CALL): ► Associate the result set to a cursor and fetches from it. ► Close the cursor after finishing fetching.	Subprogram: 1. Declare a DB2 temporary table with structure similar to the result set. 2. Select rows from regular table and insert into temporary table. 3. Declare cursor on temporary table WITH RETURN. 4. Open the cursor. Caller (with SQL CALL): ► Associate the result set to a cursor and fetches from it. ► Close the cursor after finishing fetching. Caller (with COBOL CALL): ► Declare a cursor similar to the one in subprogram and fetch from it. ► Close the cursor after finishing fetching.

Note: As you see from Table 10-4, handling result sets with subprograms may be costlier because it involves the opening of two cursors on the same table for the same purpose, once in the subprogram and once in the caller. The cost of opening a cursor depends on the number of qualifying rows. The approach above can be used when the result sets are intended to handle a small number of rows.

For more information on temporary tables, refer to *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

Based on our experience, we recommend using *created temporary tables* (CTT) over *declared temporary tables* (DTT) due to the following benefits:

- CTT can be defined under the same schema as your other tables. DTT requires definition each time.
- DTT has to be qualified with SESSION whenever referenced within the program.
- DTTs are treated as dynamic SQLs, and hence the program involves an "incremental bind" every time the table is referenced.

Important: Using created temporary tables instead of declared temporary tables for handling result sets in subprograms typically provides significant performance improvements.

Recommendation 2: Subprogram in local applications and stored procedures otherwise

By making a subprogram call from local applications like CICS or batch instead of a stored procedure call, we can still reduce the wait time associated with stored procedures. In some instances, the wait time of stored procedures will have a cascading effect on the entire application. For example, consider an application with components in CICS, distributed, and batch. A batch program can call the same stored procedure that is available to CICS and distributed components. Traditionally, a batch workload runs under low priority compared to the other two components. If the batch program waits more time to schedule a stored procedure, any locks held by it may affect its online counterparts.

This scenario poses a challenge in reusing a COBOL subprogram in local applications (CICS and batch) and remote applications due to the requirement of different language interface modules. The COBOL-DB2 subprogram requires the following language interface modules:

- DSNELI for TSO
- DFSLI000 for IMS
- DSNCLI for CICS
- DSNALI for CAF
- DSNRLI for WLM-based stored procedure address spaces

The issue can be resolved in two ways:

1. Maintain multiple load modules for the same program, each link-edited differently using one of the above language interface modules.
2. Maintain one load module and dynamically load the language interface module during runtime.

Let us consider a sample scenario where your business application ABC contains four levels of nesting:

```
PROC1 --> PROC2 --> PROC3 --> PROC4
```

All of the programs (PROC1, PROC2, PROC3, and PROC4) are required in CICS, batch, remote, and IMS components of the applications. Since the same program has to execute under different address spaces (CICS, TSO, WLM, and IMS) the program needs to have an appropriate language interface module. Let us study step-by-step the program preparation and setting up of the runtime environment with respect to both solutions mentioned above.

Solution 1: Multiple load modules

Here we link-edit the program into multiple load modules, with the same name stored in separate data sets.

Program preparation

These are the steps:

1. Develop the programs and ensure that subprogram calls are made to PROC2, PROC3, and PROC4 from their higher level programs. For example:

```
PROC1 code contains

   CALL PROC2 using .........

PROC2 code contains

   CALL PROC3 using .........

PROC3 code contains

   CALL PROC4 using .........
```

2. Pre-compile and compile the programs. Ensure that the following compile options are specified:
   `RENT, NODYNAM`
3. Have three link-edit steps each with different interface modules, as shown in Example 10-17. Ensure that the following link-edit options are specified:

```
RENT, REUS, AMODE(31), RMODE(ANY)
```

Example 10-17 Sample JCL to compile and link-edit

```
//*****************************************************************
//*        PRECOMPILE                                             *
//*****************************************************************
//PC       EXEC PGM=DSNHPC,PARM='HOST(IBMCOB)',REGION=4096K
//DBRMLIB  DD DISP=SHR,DSN=dbrm_library(PROC1)
//STEPLIB  DD DISP=SHR,DSN=<< sdsnexit >>
//         DD DISP=SHR,DSN=<< sdsnload >>
//SYSCIN   DD DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//            SPACE=(800,(500,500))
//SYSLIB DD DISP=SHR,DSN=<< copy library >>
//SYSIN  DD DISP=SHR,DSN=<< source library >>
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSUT1   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT2   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//*****************************************************************
//*        COMPILE                                                *
//*****************************************************************
//COB      EXEC PGM=IGYCRCTL,REGION=4M,
//            PARM='RENT,NODYNAM,LIST',
//            COND=(4,LT,PC)
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//            SPACE=(800,(500,500))
//SYSIN    DD DSN=&&DSNHOUT,DISP=(OLD,DELETE)
//SYSUT1   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT2   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT3   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT4   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT5   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
```

```
//SYSUT6   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT7   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//*******************************************************************
//*  PRELINK                                                        *
//*******************************************************************
//PLKED   EXEC PGM=EDCPRLK,REGION=2048K,COND=((4,LT,PC),(4,LT,COB))
//STEPLIB  DD  DSN=CEE.SCEERUN,DISP=SHR
//         DD  DSN=CEE.SCEELKED,DISP=SHR
//         DD  DISP=SHR,DSN=<<sdsnload >>
//SYSMSGS  DD  DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
//SYSIN    DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD   DD  DSN=&&PLKSET,UNIT=SYSDA,DISP=(MOD,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD  DUMMY
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSTERM  DD  SYSOUT=*
//*******************************************************************
//*        LINKEDIT for Batch                                       *
//*******************************************************************
//BATCLKED EXEC PGM=IEWL,PARM='MAP,RENT,AMODE(31),RMODE(ANY)',
//         REGION=4M,
//         COND=((4,LT,PC),(4,LT,COB),(4,LT,PLKED))
//SYSLIB   DD  DISP=SHR,DSN=CEE.SCEELKED
//         DD  DISP=SHR,DSN=<<sdsnload>>
//SYSLIN   DD DDNAME=SYSIN
//         DD DSN=&&PLKSET,DISP=(OLD,PASS)
//SYSLMOD DD DSN=<<hlq.LOAD.ELI>>   <== Load library for Batch
//        DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSIN    DD *
       INCLUDE SYSLIB(DSNELI)
       NAME PROC1(R)
/*
//*******************************************************************
//*        LINKEDIT for CICS                                        *
//*******************************************************************
//CICSLKED     EXEC PGM=IEWL,PARM='MAP,RENT,AMODE(31),RMODE(ANY)',
//          REGION=4M,
//         COND=((4,LT,PC),(4,LT,COB),(4,LT,PLKED))
//SYSLIB   DD  DISP=SHR,DSN=CEE.SCEELKED
//         DD  DISP=SHR,DSN=<< sdsnload >>
//         DD  DISP=SHR,DSN=<< sdfhload >>
//SYSLIN   DD DDNAME=SYSIN
//         DD DSN=&&PLKSET,DISP=(OLD,PASS)
//SYSLMOD DD  DSN=<<hlq.LOAD.CLI>>   <== Load library for CICS
//        DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSIN    DD *
       INCLUDE SYSLIB(DSNCLI)
       NAME PROC1(R)
/*
//*******************************************************************
//*        LINKEDIT for WLM SPAS                                    *
//*******************************************************************
//WLMLKED   EXEC PGM=IEWL,PARM='MAP,RENT,AMODE(31),RMODE(ANY)',
//         REGION=4M,
```

```
//         COND=((4,LT,PC),(4,LT,COB),(4,LT,PLKED))
//SYSLIB   DD  DISP=SHR,DSN=CEE.SCEELKED
//         DD  DISP=SHR,DSN=<< sdsnload >>
//SYSLIN   DD DDNAME=SYSIN
//         DD DSN=&&PLKSET,DISP=(OLD,DELETE)
//SYSLMOD DD  DSN=<<hlq.LOAD.RLI >>   <== Load library for WLM SPAs
//        DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSIN    DD *
       INCLUDE SYSLIB(DSNRLI)
       NAME PROC1(R)
/*
//*******************************************************************
```

4. Bind the DBRMs to produce DB2 packages.
5. Refresh WLM SPAS.
6. Define PROC1, PROC2, PROC3 and PROC4 as stored procedures.

Each of the programs that can be potentially executed across environments should undergo the process above.

Runtime environment setup

In the preparation step, we created three load libraries for the same program. We need to use them appropriately as shown in Example 10-18.

Example 10-18 Sample runtime environment setup

```
In Batch JCL, use "hlq.LOAD.ELI" dataset in STEPLIB.
In CICS started task, use "hlq.LOAD.CLI" dataset in STEPLIB.
In WLM SPA started task, use "hlq.LOAD.RLI" dataset in STEPLIB.
```

Solution 2: Dynamic invocation of the language interface module

With solution 1, you have to maintain multiple load libraries for the same program. To overcome this, another approach can be taken where the language interface module will be invoked dynamically during runtime using the dummy entry point DSNHLI. To accomplish this, the following setup is required.

Create aliases for language interface modules

We need to create aliases for the target language interface modules (DSNRLI, DSNCLI, DSNELI, DSNALI etc.,) to DSNHLI. Once this is done, any reference to DSNHLI will be resolved to the appropriate language interface module depending on the target environment. See Example 10-19.

Example 10-19 Sample job to create alias

```
//DSNALI   EXEC PGM=IEWL,PARM='XREF,LIST,NCAL',REGION=4M
//SYSLIB   DD  DISP=SHR,DSN=<< sdsnload >>
//SYSLMOD  DD  DISP=SHR,DSN=<< sdsnload.ALI >>
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSLIN    DD *
       INCLUDE SYSLIB(DSNALI)
       ALIAS DSNHLI
        ENTRY DSNALI
```

```
       NAME DSNALI(R)
//*****************************************************
//DSNRLI   EXEC PGM=IEWL,PARM='XREF,LIST,NCAL',REGION=4M
//SYSLIB   DD  DISP=SHR,DSN=<< sdsnload >>
//SYSLMOD  DD  DISP=SHR,DSN=<< sdsnload.ALI >>
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSLIN    DD *
       INCLUDE SYSLIB(DSNRLI)
       ALIAS DSNHLI
        ENTRY DSNRLI
       NAME DSNRLI(R)
//*****************************************************
//DSNELI   EXEC PGM=IEWL,PARM='XREF,LIST,NCAL',REGION=4M
//SYSLIB   DD  DISP=SHR,DSN=<< sdsnload >>
//SYSLMOD  DD  DISP=SHR,DSN=<< sdsnload.ELI >>
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSLIN    DD *
       INCLUDE SYSLIB(DSNELI)
       ALIAS DSNHLI
        ENTRY DSNELI
       NAME DSNELI(R)
//*****************************************************
//DSNCLI   EXEC PGM=IEWL,PARM='XREF,LIST,NCAL',REGION=4M
//SYSLIB   DD  DISP=SHR,DSN=<< sdfhload >>
//SYSLMOD  DD  DISP=SHR,DSN=<< sdfhload.CLI >>
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSLIN    DD *
       INCLUDE SYSLIB(DSNCLI)
       ALIAS DSNHLI
        ENTRY DSNCLI
       NAME DSNCLI(R)
```

Program preparation

These are the steps:

1. Develop the programs and ensure that subprogram calls are made to PROC2, PROC3, and PROC4 from their higher level programs. For example:

```
PROC1 code contains

   CALL PROC2 using .........

PROC2 code contains

   CALL PROC3 using .........

PROC3 code contains

   CALL PROC4 using .........
```

2. Precompile and compile the programs. Ensure that the following compile options are specified. Note the DYNAM option here. All programs have to compile with the DYNAM option:

```
RENT, DYNAM
```

3. Link-edit to create the load module. One link-edit is sufficient:

```
RENT, REUS, AMODE(31), RMODE(ANY)
```

Compile and link-edit the JCL as shown in Example 10-20.

Example 10-20 Sample JCL to compile and link-edit

```
//*********************************************************************
//*        PRECOMPILE                                                 *
//*********************************************************************
//PC       EXEC PGM=DSNHPC,PARM='HOST(IBMCOB)',REGION=4096K
//DBRMLIB  DD DISP=SHR,DSN=dbrm_library(PROC1)
//STEPLIB  DD DISP=SHR,DSN=<< sdsnexit >>
//         DD DISP=SHR,DSN=<< sdsnload >>
//SYSCIN   DD DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//            SPACE=(800,(500,500))
//SYSLIB DD DISP=SHR,DSN=<< copy library >>
//SYSIN  DD DISP=SHR,DSN=<< source library >>
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSUT1   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT2   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//*********************************************************************
//*        COMPILE                                                    *
//*********************************************************************
//COB      EXEC PGM=IGYCRCTL,REGION=4M,
//            PARM='RENT,DYNAM,LIST',
//            COND=(4,LT,PC)
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//            SPACE=(800,(500,500))
//SYSIN    DD DSN=&&DSNHOUT,DISP=(OLD,DELETE)
//SYSUT1   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT2   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT3   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT4   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT5   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT6   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT7   DD SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//*********************************************************************
//*  PRELINK                                                          *
//*********************************************************************
//PLKED   EXEC PGM=EDCPRLK,REGION=2048K,COND=((4,LT,PC),(4,LT,COB))
//STEPLIB  DD  DSN=CEE.SCEERUN,DISP=SHR
//         DD  DSN=CEE.SCEELKED,DISP=SHR
//         DD  DISP=SHR,DSN=<<sdsnload >>
//SYSMSGS  DD  DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
//SYSIN    DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD   DD  DSN=&&PLKSET,UNIT=SYSDA,DISP=(MOD,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD  DUMMY
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSTERM  DD  SYSOUT=*
//*********************************************************************
//*        LINKEDIT
//*********************************************************************
//BATCLKED EXEC PGM=IEWL,PARM='MAP,RENT,AMODE(31),RMODE(ANY)',
//         REGION=4M,
//         COND=((4,LT,PC),(4,LT,COB),(4,LT,PLKED))
//SYSLIB   DD  DISP=SHR,DSN=CEE.SCEELKED
```

```
//         DD  DISP=SHR,DSN=<<sdsnload>>
//SYSLIN   DD DDNAME=SYSIN
//         DD DSN=&&PLKSET,DISP=(OLD,PASS)
//SYSLMOD DD DSN=<<hlq.LOAD >>   <== Common load library
//        DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSIN    DD *
        NAME PROC1(R)
/*
//*******************************************************************
```

4. Bind the DBRMs to produce a DB2 package.
5. Refresh WLM SPAS.
6. Define PROC1, PROC2, PROC3, and PROC4 as stored procedures.

Runtime environment setup

Ensure that the data sets containing aliases are concatenated above the regular load library data sets, as shown in Example 10-21.

Example 10-21 Sample runtime environment setup

```
In Batch JCL, STEPLIB
//         DD DISP=SHR,DSN=<<sdsnload.ELI>>
//         DD DISP=SHR,DSN=<<sdsnload>>

In WLM SPA, STEPLIB
//         DD DISP=SHR,DSN=<<sdsnload.RLI>>
//         DD DISP=SHR,DSN=<<sdsnload>>

In CICS started task, DFHRPL
//         DD DISP=SHR,DSN=<<sdsnload.CLI>>
//         DD DISP=SHR,DSN=<<sdsnload>>
```

With this approach, it is possible to maintain one single load module for a program and use it across environments. The program can be invoked as a stored procedure as well as a subprogram.

Special considerations

Because the same program has to execute in different address spaces, environment-specific restrictions will apply. For example, there are programming restrictions in CICS, such as no native COBOL READ and WRITE statements, which do not apply in batch and WLM SPAS. For further information, refer to *CICS Transaction Server for z/OS Version 3.2 CICS Application Programming Guide*, SC34-6818, and to the appropriate reference manual for your language to determine which syntax to use.

10.4 Summary

This is a summary of the chapter:

- Use stored procedures to reduce and avoid network traffic, and to create reusable components.
- Nested stored procedures enhance the benefit of reusability.

- If the elapsed time of COBOL nested stored procedures is not acceptable, use the COBOL subprogram calls for inner levels of nesting.
- If the overhead of scheduling is increased, use the subprogram calls instead of stored procedure calls in local applications.

11

C programming

In this chapter we focus on the development of stored procedures in the C language. The C language is great for producing code that is portable across platforms and is primarily used for writing applications that perform operating system functions. If you write stored procedures that need to execute performance critical functions or exploit advanced operating system features including multi-threading, advanced file I/O, interprocess communication and network I/O, writing them in C is a good choice.

We refer to two simple applications developed in C that access sample tables. The first one 32**aA**retrieves employee information for a specific employee number, while the second retrieves a list of employees for a specific department.

Note: Complete sample programs in C can be downloaded from the Web as additional material. Download instructions can be found in Appendix B, “Additional material” on page 887.

This chapter contains the following:

- Introduction and C environment
- Passing parameters
- Elements of a C stored procedure
- Preparing and binding a C stored procedure
- Actions that the calling application must take
- Handling NULL values in parameters
- Handling result sets in the calling program
- Handling result sets using Global Temporary Tables
- Changing the security context in a C stored procedure

11.1 Introduction and C environment

The kernels of most modern operating systems, including Linux, UNIX, and Windows, as well as most system services and applications for these operating systems including HTTP, FTP, and Telnet servers, are written in C. The C language provides a runtime library on z/OS that gives you the flexibility and performance of Assembler in a high-level language that is portable across all platforms where C is available. z/OS C/C++ provides excellent support for the following operations:

- ASA text file, DBCS file, and VSAM file I/O
- Memory file and hiperspace I/O
- MTF and POSIX threading
- Interprocess communication using message queues, semaphores, shared memory, and memory mapping
- Network communication using stream sockets; datagram sockets for both the UNIX domain and Internet Address Family

In addition, C allows inter-language calls that enable you to call Assembler functions when needed if an equivalent runtime library function does not exist (such as for managing the Extended Console). C allows you to write programs for the DB2 Instrumentation Facility Interface (IFI) that issue READS, READA and COMMAND functions. Hence, if the function your stored procedure has to perform requires the use of the above-mentioned functions or facilities, C is a good choice. Most of the DB2-supplied stored procedures (see Chapter 26, “DB2-supplied stored procedures” for details) are written in C for that reason.

Before starting to develop a stored procedure, it is important to have a clear understanding of the various steps necessary to define the stored procedures environment. These steps must be verified as completed before a stored procedure can be executed. These steps are covered in detail in other chapters of this book; we simply point to them here for convenience. They are:

1. The WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 39 for details.
2. The LE environment must be set up. See Chapter 5, “Language Environment setup” on page 47 for details.
3. The stored procedure must be defined to DB2. Note in particular that if the stored procedure is designed to return result sets, the maximum number of result sets that can be returned is specified in the definition. See Chapter 9, “Defining stored procedures” on page 91 for details.
4. Develop the stored procedure. This includes the preparation of the stored procedure for execution, including binding a package if the stored procedure contains SQL statements.
5. Grant the necessary privileges to the authorization ID of the user that executes the stored procedure. See Chapter 7, “Security and authorization” on page 65 for details.
6. Develop the calling application, if needed.
7. See Chapter 16, “Debugging” on page 313 for details on testing and debugging.

11.2 Passing parameters

Example 9-4 on page 108 shows our first sample C stored procedure. More examples are available as described in Appendix B, “Additional material” on page 887.

A stored procedure can receive parameters from and send back parameters to the calling application. When the calling application issues an SQL CALL to the stored procedure, DB2 builds a parameter list based on the parameters coded in the SQL call and the information specified when the stored procedure is initially defined. One of the options on the CREATE PROCEDURE statement is PARAMETER STYLE, which specifies whether or not NULL values can be passed as parameters. This is discussed in detail in 9.1, “CREATE or ALTER PROCEDURE parameters” on page 92. When NULL values are permitted, the stored procedure and the calling program must take some additional steps. This is discussed in 11.6, “Handling NULL values in parameters” on page 161. In this section we use the parameter style GENERAL that does not permit NULL values.

Both parameter style SQL and DBINFO are valid for C stored procedures. However, GENERAL and GENERAL WITH NULLS are most commonly used, hence they will be discussed in this chapter.

For a C stored procedure retrieving information about a specific employee, the parameter list is specified when defining the stored procedure. Example 11-1 shows the CREATE PROCEDURE statement for the C example.

Example 11-1 CREATE PROCEDURE statement for the C example

```
CREATE PROCEDURE DEVL7083.EMPDTL1P
    ( IN EMPNO CHAR(6) CCSID EBCDIC
    , OUT FIRSTNME VARCHAR(12) CCSID EBCDIC
    , OUT MIDINIT CHAR(1) CCSID EBCDIC
    , OUT LASTNAME VARCHAR(15) CCSID EBCDIC
    , OUT WORKDEPT CHAR(3) CCSID EBCDIC
    , OUT HIREDATE DATE
    , OUT SALARY DEC(9,2)
    , OUT RETCODE INTEGER
    , OUT MESSAGE VARCHAR(1331) CCSID EBCDIC
    ) ...
```

This definition specifies whether the parameter is IN (input to the stored procedure), OUT (output from the stored procedure) or INOUT (input to and output from the stored procedure). It also specifies the data type and size of each parameter. This list must be compatible with the parameter declarations in the calling application, which is shown in Example 11-2.

Example 11-2 Parameter definitions of calling application

```
char h_empno[7];
char h_firstnme[13];
char h_midinit[2];
char h_lastname[15];
char h_workdept[4];
char h_hiredate[11];
decimal(9,2) h_salary;
long int h_retcode;
char h_message[1332];
```

Writing a C stored procedure as a main program or subprogram

A stored procedure that runs in a WLM-established address space and uses Language Environment can be either a main program or a subprogram. A stored procedure that runs as a subprogram can perform better because Language Environment does less processing for it.

In general, a subprogram must do the following extra tasks that Language Environment performs for a main program:

- Initialization and cleanup processing
- Allocating and freeing storage
- Closing all open files before exiting

A C program written as *main* contains a main() function. Parameters are passed to it through argc and argv. DB2 sets the value of the first element in the argv array, (argv[0]), to the name of the procedure. The remaining elements of the argv array correspond to the parameters as defined by the PARAMETER STYLE of the procedure

A C program written as *subprogram* is a fetchable function. Parameters are passed to it explicitly; see Example 11-3.

Example 11-3 C stored procedure coded as a subprogram

```
/*********************************************************************/
/*  This C subprogram is a stored procedure that uses linkage        */
/*  convention GENERAL and receives 3 parameters.                    */
/*********************************************************************/
#pragma linkage(cfunc,fetchable)
#include <stdlib.h>
void cfunc(char p1[11],long *p2,short *p3)
{
  /*****************************************************************/
  /*  Declare variables used for SQL operations.  These variables */
  /*  are local to the subprogram and must be copied to and from  */
  /*  the parameter list for the stored procedure call.           */
  /*****************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
  EXEC SQL END DECLARE SECTION;
  /*****************************************************************/
  /*  Receive input parameter values into local variables.     */
  /*****************************************************************/
  strcpy(parm1,p1);
  parm2 = *p2;
  parm3 = *p3;
  /*****************************************************************/
  /*  Perform operations on local variables.                   */
  /*****************************************************************/
  ...
  /*****************************************************************/
  /*  Set values to be passed back to the caller.              */
  /*****************************************************************/
  strcpy(parm1,"SETBYSP");
  parm2 = 100;
  parm3 = 200;
  /*****************************************************************/
  /*  Copy values to output parameters.                        */
  /*****************************************************************/
  strcpy(p1,parm1);
  *p2 = parm2;
  *p3 = parm3;
}
```

All the stored procedures in this chapter are written as main programs.

11.3 Elements of a C stored procedure

Every C stored procedure should contain the following elements:

- Includes and compiler defines
- Constants defines
- Messages defines
- Structures, enums and types defines
- Global variables declarations
- Functions defines
- SQLCA include
- DB2 host variables declarations
- Cursors declarations
- Main routine of the stored procedure
- Helper functions

As in any C program, you have to make all the required defines and include the required header files as documented in the *z/OS V1R9.0 XL C/C++ Run-Time Library Reference,* SA22-7821-09. If you use the GENERAL or GENERAL WITH NULLS parameter style, you have to indicate that the format of the argument list passed to the stored procedure on initialization is in MVS linkage format using the C #pragma runopts(plist(os)). Use the #pragma CSECT directive, if you will be using SMP/E to service your product, and to aid in debugging your program. Example 11-4 shows includes and compiler defines.

Example 11-4 Includes and compiler defines

```
#ifdef DEBUG                          /* File options for debugging */
  #pragma runopts(plist(os),msgfile(OUT1))
  #define OUT stderr
#else
  #pragma runopts(plist(os))
#endif
#include <stdio.h>
#include <string.h>
#include <decimal.h>
#include <ctype.h>
#pragma csect(CODE,"EMPDTL1P")        /* Names code segment         */
#pragma csect(STATIC,"EMPDTL1S")
```

It is good practice to define all constants that you use in the stored procedure before your main function, so that they can be easily changed if required without having to change the actual source code statements. Example 11-5 shows constants defines.

Example 11-5 Constants defines

```
#define RETSEV              12      /* Severe error return code   */
#define RETOK                0      /* No error return code       */
#define MSGROWLEN          121      /* Length of an errmsg line   */
#define DATA_DIM            10      /* Number of message lines    */
#define BLANK              ' '      /* Buffer padding             */
#define NULLCHAR          '\0'      /* Null character             */
#define LINEFEED          0x25      /* Linefeed character         */
#define MIN_EMPNO_LEN        6      /* Minimum empno length       */
#define MAX_EMPNO_LEN        6      /* Maximum empno length       */
```

It is good practice to define all the messages your stored procedure uses before your main function for easier maintenance. Example 11-6 shows the message defines.

Example 11-6 Messages defines

```
 #define INF_COMP             "EMPDTL1P completed successfully..."
 #define ERR_DSNTIAR          "DSNTIAR could not detail the SQL \
error..."
 #define ERR_EMPNO_LEN        "EMPNO length is invalid..."
 #define ERR_QUERY_INFO       "*** SQL error when selecting employee \
data..."
 #define ERR_INVALID_WORKDEPT "Invalid NULL value for WORKDEPT..."
 #define ERR_INVALID_HIREDATE "Invalid NULL value for HIREDATE..."
 #define ERR_INVALID_SALARY   "Invalid NULL value for SALARY..."
```

In our example we gather information about an employee. Therefore, we define a type EMPLOYEE that can hold the values from the database query. Example 11-7 shows structures, enums, and types defined.

Example 11-7 Structures, enums, and types defined

```
 typedef struct
 {
   char empno[7];
   char firstnme[13];
   char midinit[2];
   char lastname[15];
   char workdept[4];
   char hiredate[11];
   decimal(9,2) salary;
 } EMPLOYEE;
```

As in any C program, you should avoid the use of global variables to reduce program complexity and unwanted side-effects. We use only two global variables for error-handling and flow control. rc is the return code that indicates if the stored procedure completed successfully (RC=0) or if there was an error (RC=12). If there was a SQL error, we use DSNTIAR to obtain a formatted form of the SQLCA and a text message based on the SQLCODE field of the SQLCA. If there was a problem using a C runtime library function, we return a formatted error message indicating the location in the program where the error occurred. Both the return code and the message are returned as OUT parameters to the calling program. Example 11-8 shows the global variable declarations.

Example 11-8 Global variables declarations

```
 long int rc;                          /* Return code               */
 char errmsg[DATA_DIM + 1][MSGROWLEN]; /* Error message             */
```

In addition to the main function, our stored procedure contains helper functions for better modularity and to avoid duplicating code. We discuss each function in this chapter. Example 11-9 on page 153 shows the definitions for these functions.

Example 11-9 Functions defines

```
void sql_error(char[]);
char * rtrim(char *);
void query_info(EMPLOYEE *);
```

Our stored procedure contains SQL statements, and must include a definition of the SQLCA and a declaration of all host variables used in SQL statements. Example 11-10 shows the SQLCA include and the DB2 host variable declarations.

Example 11-10 SQLCA include and DB2 host variable declaration

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char h_empno[7];
char h_firstnme[13];
char h_midinit[2];
char h_lastname[15];
char h_workdept[4];
short int i_workdept;
char h_hiredate[11];
short int i_hiredate;
decimal(9,2) h_salary;
short int i_salary;
EXEC SQL END DECLARE SECTION;
```

We have to trim trailing blanks from parameters passed to the stored procedure. Unfortunately, there is no runtime library function to do that, so we have to provide one as shown in Example 11-11.

Example 11-11 Helper function rtrim

```
char * rtrim(char * pstring)
{
  char * pend;

  if (pstring == NULL)
    return NULL;

  pend = pstring + strlen(pstring) - 1; /* Points to last character */
  while (pend > pstring && isblank(*pend)) /* Decrement until non-  */
    pend--;                               /* blank char is encountered*/

  if (pstring == pend && isblank(*pend))/* If the string is empty   */
    *pend = NULLCHAR;
  else
    *(pend + 1) = NULLCHAR;             /* Otherwise add NUL term   */

  return pstring;
}
```

The function sql_error is called explicitly whenever an unexpected SQLCODE is encountered. It accepts a null-terminated location message as parameter. This message is concatenated with the formatted DSNTIAR message, and saved in the global variable errmsg. In addition, `rc` is set to RETSEV(12) to indicate a severe error. See Example 11-12.

Example 11-12 Helper function sql_error

```
void sql_error(char locmsg[])
{
  struct error_struct                 /* DSNTIAR message structure  */
  {
    short int error_len;
    char error_text[DATA_DIM][MSGROWLEN];
  } error_message = {DATA_DIM * MSGROWLEN};
  short int tiar_rc;                  /* DSNTIAR return code        */
  int i, j, k;                        /* Loop control               */
  int lrecl = MSGROWLEN;

  rc = RETSEV;                        /* A fatal error has occured  */
  strcpy(errmsg[0], locmsg);          /* Copy locator message       */
  tiar_rc = dsntiar(&sqlca, &error_message, &lrecl); /* Format msg  */

  if (tiar_rc == 0)                   /* The call was successful    */
  {
    for (i = 0, j = 1; i < DATA_DIM; i++)
    {
      for (k = 0; (error_message.error_text[i][k] == BLANK &&
                k < MSGROWLEN); k++);
        if (k < MSGROWLEN)            /* Do not copy blank lines    */
          strncpy(errmsg[j++],
                  (char *) &error_message.error_text[i][1],
                  MSGROWLEN - 1);
    }
  }
  else                                /* DSNTIAR error occured      */
  {
    strcpy(errmsg[1], ERR_DSNTIAR);
    sprintf(errmsg[2], "*** SQLCODE = %d", sqlca.sqlcode);
    strcpy(errmsg[3], "*** SQLERRM is ");
    for (i = 0; i < sqlca.sqlerrml; i++)
      errmsg[4][i] = sqlca.sqlerrmc[i];
  }
}
```

Next we look at the main function. It is important to initialize all used variables first, and clear all OUT parameters. It is a good idea to check the syntax of the IN parameters first, such as the correct length for a character string or the range for a numerical value, and return an error if the check fails. Example 11-13 shows how to do this.

Example 11-13 Main function initialization and handling IN parameters

```
main(int argc, char *argv[])
{
  EMPLOYEE employee;
  char * pcurbyte;
  int i, j;

  /*********************************************************************/
  /* Initialize variables and OUT parameters.                          */
  /*********************************************************************/
  rc = RETOK;
  memset(errmsg, NULLCHAR, sizeof(errmsg)); /* Clear message buffer */
  memset((void *)&employee, NULLCHAR, sizeof(employee));

  /*********************************************************************/
```

```
/* Check and get IN parameters.                                      */
/*********************************************************************/
strcpy(employee.empno, rtrim((char *)argv[1]));
if (strlen(employee.empno) < MIN_EMPNO_LEN || /* Syntax check     */
    strlen(employee.empno) > MAX_EMPNO_LEN)
{
  strcpy(errmsg[0], ERR_EMPNO_LEN);
  rc = RETSEV;
}
```

After we have verified that the employee number has the required length, we can query the employee information and return the results. Since we have defined the parameter style as GENERAL, we cannot return NULLs, and must return default values instead. It is important that the calling application checks the return code in any case, and does not rely on logic that depends on certain values of the output parameters. Before we return the errmsg lines, we add an ASCII line feed character after each line, so that the message will display nicely in a calling application running on Linux, UNIX, or Windows. See Example 11-14.

Example 11-14 Main function database employee data query and returning results

```
/*********************************************************************/
/* Query information.                                                */
/*********************************************************************/
if (rc < RETSEV)
  query_info(&employee);

/*********************************************************************/
/* Return results.                                                   */
/*********************************************************************/
if (rc < RETSEV)
{
  strcpy((char *)argv[2], employee.firstnme);
  strcpy((char *)argv[3], employee.midinit);
  strcpy((char *)argv[4], employee.lastname);
  strcpy((char *)argv[5], employee.workdept);
  strcpy((char *)argv[6], employee.hiredate);
  *(decimal(9,2) *)argv[7] = employee.salary;
}
else
{
  strcpy((char *)argv[2], "");      /* We cannot return NULL      */
  *(char *)argv[3] = NULLCHAR;
  strcpy((char *)argv[4], "");
  strcpy((char *)argv[5], "");
  strcpy((char *)argv[6], "0001-01-01");
  *(decimal(9,2) *)argv[7] = 0.00;
}

if (rc == RETOK)
  strcpy(errmsg[0], INF_COMP);
*(int *)argv[8] = rc;
if (errmsg[0][0] != BLANK)           /* If error message exists   */
{
  pcurbyte = argv[9];
  for (i = 0; i < DATA_DIM + 1; i++)
  {
    for (j = 0; (errmsg[i][j] != NULLCHAR && j < MSGROWLEN); j++)
      *pcurbyte++ = errmsg[i][j];
    if (j > 0)
```

```
        *pcurbyte++ = LINEFEED;
      }
      *pcurbyte = NULLCHAR;
    }
  }
```

The function query_info has a single parameter, which is a pointer to an EMPLOYEE variable where it expects the employee number empno to be filled in. It then selects the missing information from the database, and sets the other fields of the employee variable. The EMPLOYEE table allows NULL values for WORKDEPT, HIREDATE and SALARY. If we receive a NULL value for any of these fields, we return an error since we cannot pass a valid NULL value back as an output parameter. See Example 11-15.

Example 11-15 Helper function query_info

```
void query_info(EMPLOYEE * pemployee)
 {
   strcpy(h_empno, pemployee->empno);

   EXEC SQL SELECT FIRSTNME, MIDINIT, LASTNAME, WORKDEPT,
                   HIREDATE, SALARY
              INTO :h_firstnme, :h_midinit, :h_lastname,
                   :h_workdept:i_workdept, :h_hiredate:i_hiredate,
                   :h_salary:i_salary
              FROM EMP
             WHERE EMPNO = :h_empno;

   #ifdef DEBUG
       fprintf(OUT,"query_info: select from EMP \
SQLCODE=%ld\n", SQLCODE);
   #endif

   if (SQLCODE != 0)
     sql_error(ERR_QUERY_INFO);
   else
   {
     strcpy(pemployee->firstnme, h_firstnme);
     strcpy(pemployee->midinit, h_midinit);
     strcpy(pemployee->lastname, h_lastname);
     if (i_workdept < 0)
     {
       strcpy(errmsg[0], ERR_INVALID_WORKDEPT);
       rc = RETSEV;
       return;
     }
     else
       strcpy(pemployee->workdept, h_workdept);
     if (i_hiredate < 0)
     {
       strcpy(errmsg[0], ERR_INVALID_HIREDATE);
       rc = RETSEV;
       return;
     }
     else
       strcpy(pemployee->hiredate, h_hiredate);
     if (i_salary < 0)
     {
       strcpy(errmsg[0], ERR_INVALID_SALARY);
       rc = RETSEV;
```

```
      return;
    }
    else
      pemployee->salary = h_salary;
  }
}
```

11.4 Preparing and binding a C stored procedure

Example 11-16 shows the JCL used to prepare, compile, and link-edit the stored procedure and bind the package. Make sure you specify HOST(C) when you prepare your source code module. By default STDSQL(NO) is selected which requires you to explicitly include the definition for the SQLCA using EXEC SQL INCLUDE SQLCA. Make sure you include the search path to all the required include files. Choose the SOURCE compile option to have the precompiled source code printed to the SYSCPRT data set, because if you get a compile error referencing a source code line, you will not be able to locate it in the original source code because of the DB2 precompiler's work. The RENT option specifies that the compiler is to take code that is not naturally reentrant and make it reentrant. Refer to *z/OS V1R9.0 Language Environment Programming Guide,* SA22-7569-02, for a detailed description of reentrancy.

Although CEESTART is the default main entry point for C applications, it is good practice to explicitly specify it in your link-edit SYSIN. You also must link-edit it with the RRSAF language interface module DSNRLI. In our example we also include the DSNTIAR Assembler routine because we use it in our sql_error function.

Example 11-16 JCL to compile EMPDTL1P

```
//PAOLOR1Q JOB (999,POK),'C P/C/L/B',CLASS=A,MSGCLASS=H,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=SC63,L=9999
// JCLLIB ORDER=(DB9AU.PROCLIB)
//*
//JOBLIB  DD  DSN=DB9A9.SDSNEXIT,DISP=SHR
//        DD  DSN=DB9A9.SDSNLOAD,DISP=SHR
//        DD  DSN=CEE.SCEERUN,DISP=SHR
//*-------------------------------------------------------
//* STEP 01: PRE-COMPILE, COMPILE, LINK-EDIT EMPDTL1P
//*          STORED PROCEDURE
//*-------------------------------------------------------
//STEP01   EXEC PROC=DSNHCPP,MEM=EMPDTL1P,USER=DB9AU,
//         PARM.PC=('HOST(C)',CCSID(1047)),
//         PARM.CP='/OPTFILE(DD:COPT)'
//PC.DBRMLIB   DD DSN=SG247083.DEVL.DBRM(EMPDTL1P),
//             DISP=SHR
//PC.SYSLIB    DD DSN=SG247083.C.SOURCE,
//             DISP=SHR
//PC.SYSIN     DD DSN=SG247083.C.SOURCE(EMPDTL1P),
//             DISP=SHR
//CP.COPT    DD  *
   SEARCH('CEE.SCEEH.H')
   SEARCH('CEE.SCEEH.SYS.H')
   MARGINS(1,72)
   SOURCE
   LIST
```

```
   RENT
   DEF(DEBUG)
/*
//LKED.SYSLMOD DD DSN=SG247083.DEVL.LOAD(EMPDTL1P),
//             DISP=SHR
//LKED.SYSIN   DD *
   ORDER CEESTART,EMPDTL1P
   INCLUDE SYSLIB(DSNRLI)
   INCLUDE SYSLIB(DSNTIAR)
   ENTRY CEESTART
   MODE AMODE(31),RMODE(ANY)
/*
//*-------------------------------------------------------
//* STEP 02: BIND EMPDTL1P STORED PROCEDURE
//*-------------------------------------------------------
//STEP02   EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *
   DSN SYSTEM(DB9A)
   BIND PACKAGE(DEVL7083) -
        MEMBER(EMPDTL1P) ACT(REP) ISO(UR) ENCODING(EBCDIC) -
        OWNER(DEVL7083) LIBRARY('SG247083.DEVL.DBRM')
   END
/*
```

If the stored procedure contains SQL statements as in our example, you will get a DBRM that you must bind into a package. It does not require a plan since it runs under the thread for the calling application. The following special processing is needed for *binding* a stored procedure.

Bind the DBRM to DB2 using the command BIND PACKAGE. If you use the ENABLE option of the BIND PACKAGE command to control access to the stored procedure package, you must enable the system connection type of the calling application.

The package for the stored procedure need not be bound with the plan for the program that calls it since it runs under the thread for the calling application.

The owner of the package that contains the SQL CALL must have the EXECUTE authority on the procedure. See Chapter 7, “Security and authorization (Glenn)” on page 57 for details.

The collection ID associated with the stored procedure package must be based on the following rules:

- If you specify NO PACKAGE PATH and NO COLLID when creating the stored procedure, the package must use the same collection ID as the calling program.
- If you specify PACKAGE PATH collection_id1, collection_id2,... when creating the stored procedure, the stored procedure must use one of these collections.
- If you specify NO PACKAGE PATH and COLLID collection_id when creating the stored procedure, the stored procedure must use this collection_id.

Also, see 9.1.8, “Package path” on page 102 for details on the definition of the collection ID and 9.1.10, “Collection ID the stored procedure runs in” on page 103.

Choosing the right isolation level is very important. Many clients may concurrently call a stored procedure, and an incorrectly chosen isolation level can cause serious contention. If

you only have read-only operations in your stored procedure, UNCOMITTED READ (UR) may be a good choice. UR allows your application to read any row that another process has changed, even if the process has not committed the row.

11.5 Actions that the calling application must take

The calling application must initialize all passed parameters declared as INPUT or INOUT before calling the stored procedure. The main function of the calling application is listed in Example 11-17. Notice that the BEGIN/END DECLARE section for the calling application is different from the called application.

Example 11-17 SQL CALL C example

```
/******************************************************************/
 /* Declare DB2 host variables.                                    */
 /******************************************************************/
 EXEC SQL BEGIN DECLARE SECTION;
 char h_empno[7];
 char h_firstnme[13];
 char h_midinit[2];
 char h_lastname[15];
 char h_workdept[4];
 char h_hiredate[11];
 decimal(9,2) h_salary;
 long int h_retcode;
 char h_message[1332];
 EXEC SQL END DECLARE SECTION;

/******************************************************************/
 /* Main routine.                                                  */
 /******************************************************************/
main(int argc, char *argv[])
 {
   int i;
   char line[MAX_LINE_LEN + 2];        /* SYSIN line buffer          */
   char * pline;
   char * ptoken;

   /*****************************************************************/
   /* Initialize variables.                                         */
   /*****************************************************************/
   rc = RETOK;
   memset(errmsg, NULLCHAR, sizeof(errmsg)); /* Clear message buffer */

   /*****************************************************************/
   /* Check and get parameters from SYSIN.                          */
   /*****************************************************************/
   if ((pline = gets(line)) == NULL)
   {
     strcpy(errmsg[0], ERR_READ_STDIN);
     rc = RETSEV;
   }

   if (rc < RETSEV)
   {
     line[MAX_SYSIN_LEN] = NULLCHAR;
     pline = trim(pline);
     if (strlen(pline) < MIN_EMPNO_LEN || /* Check IN param. syntax  */
```

```
      strlen(pline) > MAX_EMPNO_LEN)
    {
      strcpy(errmsg[0], ERR_EMPNO_LEN);
      rc = RETSEV;
    }
    else
      strcpy(h_empno, pline);
  }

  /*****************************************************************/
  /* Call EMPDTL1P.                                                */
  /*****************************************************************/
  if (rc < RETSEV)
  {
EXEC SQL CALL EMPDTL1P(:h_empno, :h_firstnme, :h_midinit,
                       :h_lastname, :h_workdept,
                       :h_hiredate, :h_salary,
                       :h_retcode, :h_message);
    if (SQLCODE != 0)
      sql_error(ERR_CALL_EMPDTL1P);
}

  /*****************************************************************/
  /* Print results.                                                */
  /*****************************************************************/
  if (rc < RETSEV)
  {
    if (h_retcode > RETOK)            /* Check for internal SP err.*/
    {
      rc = h_retcode;
      memcpy(errmsg, h_message, sizeof(h_message));
    }
    else
    {
      printf("**** EMPLOYEE REPORT FOR EMPNO %s ****\n", h_empno);
      printf("       FIRSTNAME: %s\n", h_firstnme);
      printf("  MIDDLE INITIAL: %s\n", h_midinit);
      printf("        LASTNAME: %s\n", h_lastname);
      printf("      DEPARTMENT: %s\n", h_workdept);
      printf("       HIRE DATE: %s\n", h_hiredate);
      printf("          SALARY: $ %D(9,2)\n", h_salary);
    }
  }

  if (rc > RETOK && errmsg[0][0] != BLANK) /* If there was an error */
  {                                        /* print error message   */
    for (i = 0; i < DATA_DIM + 1; i++)
    {
      errmsg[i][MSGROWLEN - 1] = NULLCHAR;
      fprintf(stderr, "%s\n", errmsg[i]);
    }
  }

  return rc;
}
```

After we have verified that the SQLCODE of the SQL CALL statement is 0, we check the OUT return code parameter to determine whether the OUT parameters are valid. We will use them only if the stored procedure completed successfully with RC=0.

11.6 Handling NULL values in parameters

When the number and size of parameters passed is large, or when it makes sense from a semantic point of view, you should consider allowing NULL values. In our example it makes semantic sense because some of the database columns that correspond to the OUT parameters allow NULL values. Example 11-18 lists the new structures. Notice the differences from Example 11-7 on page 152 because of the isxxxNull definitions.

Example 11-18 Structures, enums, and types defines with nulls

```
typedef int BOOL;                    /* Boolean type              */
 typedef struct
 {
   char empno[7];
   char firstnme[13];
   char midinit[2];
   char lastname[15];
   BOOL isWorkdeptNull;
   char workdept[4];
   BOOL isHiredateNull;
   char hiredate[11];
   BOOL isSalaryNull;
   decimal(9,2) salary;
 } EMPLOYEE;
:
```

Example 11-19 contains the main function of our sample application with NULL values allowed. When the parameter style GENERAL WITH NULLS is specified, an array of short indicator variables of the length of the number of parameters is passed as an additional parameter. For easier use, we copy this array to a local array before we check the IN parameters. The check for unexpected NULL values should be included as part of our parameter syntax check. A NULL value as an input parameter may not mean an error, but it can mean that a default value should be used instead.

Example 11-19 Main function initialization and handling IN parameters with NULLS

```
 main(int argc, char *argv[])
 {
   EMPLOYEE employee;
   char * pcurbyte;
   short int locind[9];               /* Indicator variables       */
   short int *pind;                   /* Pointer to indicator vars */
   int i, j;

   /*********************************************************************/
   /* Initialize local variables and OUT parameters.                    */
   /*********************************************************************/
   rc = RETOK;
   memset(errmsg, NULLCHAR, sizeof(errmsg)); /* Clear message buffer */
   memset((void *)&employee, NULLCHAR, sizeof(employee));
   pind = (short int *)argv[10];      /* Locate and recast arg     */
   for (i = 0; i < 9; i++)            /* Copy null-ind array       */
   {
     locind[i] = *pind;
     pind++;
   }

   /*********************************************************************/
   /* Check and get IN parameters.                                      */
```

```
/*********************************************************************/
if (locind[0] < 0)                  /* If parameter is NULL       */
{
  strcpy(errmsg[0], ERR_EMPNO_NULL);
  rc = RETSEV;
}
else
{
  strcpy(employee.empno, rtrim((char *)argv[1]));
  if (strlen(employee.empno) < MIN_EMPNO_LEN ||
      strlen(employee.empno) > MAX_EMPNO_LEN)
  {
    strcpy(errmsg[0], ERR_EMPNO_LEN);
    rc = RETSEV;
  }
}
```

When we return OUT parameters, we need to make sure that we also set the indicator variables accordingly, as shown in Example 11-20.

Example 11-20 Main function database employee data query and returning results

```
/*********************************************************************/
/* Query information.                                                */
/*********************************************************************/
if (rc < RETSEV)
  query_info(&employee);

/*********************************************************************/
/* Return results.                                                   */
/*********************************************************************/
for (i = 0; i < 9; i++)             /* Set all output params NULL */
  locind[i] = -1;

if (rc < RETSEV)
{
  strcpy((char *)argv[2], employee.firstnme);
  locind[1] = 0;
  strcpy((char *)argv[3], employee.midinit);
  locind[2] = 0;
  strcpy((char *)argv[4], employee.lastname);
  locind[3] = 0;
  if (!employee.isWorkdeptNull)
  {
    strcpy((char *)argv[5], employee.workdept);
    locind[4] = 0;
  }
  if (!employee.isHiredateNull)
  {
    strcpy((char *)argv[6], employee.hiredate);
    locind[5] = 0;
  }
  if (!employee.isSalaryNull)
  {
    *(decimal(9,2) *)argv[7] = employee.salary;
    locind[6] = 0;
  }
}
```

```
  if (rc == RETOK)
    strcpy(errmsg[0], INF_COMP);
  *(int *)argv[8] = rc;
  locind[7] = 0;

  if (errmsg[0][0] != BLANK)           /* If error message exists   */
  {
    pcurbyte = argv[9];
    for (i = 0; i < DATA_DIM + 1; i++)
    {
      for (j = 0; (errmsg[i][j] != NULLCHAR && j < MSGROWLEN); j++)
        *pcurbyte++ = errmsg[i][j];
      if (j > 0)
        *pcurbyte++ = LINEFEED;
    }
    *pcurbyte = NULLCHAR;
    locind[8] = 0;
  }

  /* Return indicator variables */
  pind = (short int *)argv[10];        /* Locate and recast arg     */
  for (i = 0; i < 9; i++)              /* Copy over null-ind array  */
  {
    *pind = locind[i];
    pind++;
  }
 }
```

Example 11-21 shows the new query_info definition.

Example 11-21 Helper function query_info with indicators

```
void query_info(EMPLOYEE * pemployee)
 {
   strcpy(h_empno, pemployee->empno);

   EXEC SQL SELECT FIRSTNME, MIDINIT, LASTNAME, WORKDEPT,
                   HIREDATE, SALARY
              INTO :h_firstnme, :h_midinit, :h_lastname,
                   :h_workdept:i_workdept, :h_hiredate:i_hiredate,
                   :h_salary:i_salary
              FROM EMP
             WHERE EMPNO = :h_empno;

   #ifdef DEBUG
       fprintf(OUT,"query_info: select from EMP \
SQLCODE=%ld\n", SQLCODE);
   #endif

   if (SQLCODE != 0)
     sql_error(ERR_QUERY_INFO);
   else
   {
     strcpy(pemployee->firstnme, h_firstnme);
     strcpy(pemployee->midinit, h_midinit);
     strcpy(pemployee->lastname, h_lastname);
     if (i_workdept < 0)
       pemployee->isWorkdeptNull = TRUE;
     else
```

```
      strcpy(pemployee->workdept, h_workdept);
    if (i_hiredate < 0)
      pemployee->isHiredateNull = TRUE;
    else
      strcpy(pemployee->hiredate, h_hiredate);
    if (i_salary < 0)
      pemployee->isSalaryNull = TRUE;
    else
      pemployee->salary = h_salary;
  }
}
```

The calling program needs to include indicator variables in the CALL statement and defensively check whether any output parameters contain an unexpected NULL value as shown in Example 11-22, which also shows the modified DECLARE section, which is different from Example 11-17 on page 159.

Example 11-22 Calling a stored procedure with PARAMETER STYLE GENERAL WITH NULL

```
/*********************************************************************/
 /* Declare DB2 host variables.                                       */
/*********************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
 char h_empno[7];
 short int i_empno;
 char h_firstnme[13];
 short int i_firstnme;
 char h_midinit[2];
 short int i_midinit;
 char h_lastname[15];
 short int i_lastname;
 char h_workdept[4];
 short int i_workdept;
 char h_hiredate[11];
 short int i_hiredate;
 decimal(9,2) h_salary;
 short int i_salary;
 long int h_retcode;
 short int i_retcode;
 char h_message[1332];
 short int i_message;
 EXEC SQL END DECLARE SECTION;
/*********************************************************************/

  if (rc < RETSEV)
  {
    EXEC SQL CALL EMPDTL2P(:h_empno:i_empno,
                           :h_firstnme:i_firstnme,
                           :h_midinit:i_midinit,
                           :h_lastname:i_lastname,
                           :h_workdept:i_workdept,
                           :h_hiredate:i_hiredate,
                           :h_salary:i_salary,
                           :h_retcode:i_retcode,
                           :h_message:i_message);
    if (SQLCODE != 0)
      sql_error(ERR_CALL_EMPDTL2P);
}
```

```
/*****************************************************************/
/* Print results.                                                */
/*****************************************************************/
if (rc < RETSEV)
{
  if (i_retcode < 0)                /* Check for internal SP err.*/
  {
    rc = RETSEV;
    strcpy(errmsg[0], ERR_NULL_RETCODE);
  }
  else if (h_retcode > RETOK)
  {
    rc = h_retcode;
    if (i_message >= 0)
      memcpy(errmsg, h_message, sizeof(h_message));
  }
  else
  {
    printf("**** EMPLOYEE REPORT FOR EMPNO %s ****\n", h_empno);
    printf("       FIRSTNAME: %s\n",
           (i_firstnme < 0) ? "-" : h_firstnme);
    printf("  MIDDLE INITIAL: %s\n",
           (i_midinit < 0) ? "-" : h_midinit);
    printf("        LASTNAME: %s\n",
           (i_lastname < 0) ? "-" : h_lastname);
    printf("      DEPARTMENT: %s\n",
           (i_workdept < 0) ? "-" : h_workdept);
    printf("       HIRE DATE: %s\n",
           (i_hiredate < 0) ? "-" : h_hiredate);
    if (i_salary < 0)
      printf("          SALARY: $ %D(9,2)\n", 0.00);
    else
      printf("          SALARY: $ %D(9,2)\n", h_salary);
  }
}

if (rc > RETOK && errmsg[0][0] != BLANK) /* If there was an error */
{                                        /* print error message   */
  for (i = 0; i < DATA_DIM + 1; i++)
  {
    errmsg[i][MSGROWLEN - 1] = NULLCHAR;
    fprintf(stderr, "%s\n", errmsg[i]);
  }
}

return rc;
}
```

In summary, you must do the following to handle parameters that allow NULL values:

- Make sure the stored procedure definition allows NULL parameters.
- In the calling program, declare indicator variables and set their value to 0 if the parameter is not NULL and -1 if parameter is NULL.
- Include the indicator variables in the CALL statement.
- In the stored procedure, declare the indicator variables.
- In the stored procedure, check for the value of the null indicator to determine whether the parameter is null and take appropriate action.

- If you need to set an OUTPUT or INOUT parameter to null, set its indicator variable to -1.

11.7 Handling result sets in the calling program

If the stored procedure returns a small amount of data that does not contain repeating groups (for example, information about an employee), it is much simpler to avoid result sets altogether, returning the data as parameters as shown in the previous examples. When the stored procedure must return result sets, each consisting of multiple rows (for example, information about all employees in a department), there are two possibilities:

- The number of result sets is fixed, and you know the contents.
- The number of result sets is variable, and you do not know the contents.

Handling the first case is easier to develop, but the second case is more general, and requires minimal modifications if the calling program or stored procedure happens to change. Our sample stored procedure always returns only one result set, and we know the contents.

The following steps are required to handle result sets:

1. When you define the stored procedure to DB2 (see Chapter 9, “Defining stored procedures” on page 91), specify the maximum number of result sets that can be generated by the stored procedure.
2. In the stored procedure, declare and open a cursor for each result set. Note that the stored procedure must not fetch rows from the cursor nor close the cursor. You must declare each cursor using the WITH RETURN clause. If the stored procedure is created using the COMMIT ON RETURN option (see 9.1.17, “Use of commit before returning” on page 106 for details), the cursor must also be declared using the WITH HOLD clause to prevent it from being closed when control returns to the calling program.
3. In the calling program, declare a locator variable for each result set that will be returned. If you do not know how many result sets will be returned, declare enough result set locators for the maximum number possible. An example of a declaration for a single result set locator follows:

```
volatile SQL TYPE IS RESULT_SET_LOCATOR * rs_loc;
```

4. In the calling program, call the stored procedure and check the return code. If the SQLCODE is +466 (SQLSTATE is 0100C), the stored procedure has returned result sets.
5. Link the result set locators to the result sets:

```
EXEC SQL ASSOCIATE LOCATOR (:rs_loc)
                    WITH PROCEDURE EMPRSETP;
```

6. Allocate a cursor for each result set to be processed:

```
EXEC SQL ALLOCATE EMPRSETP_CSR CURSOR
            FOR RESULT SET :rs_loc;
```

7. Fetch and process all rows from the cursors. This process is similar to processing any normal cursor, except that the cursor has already been opened by the stored procedure. If the cursor is declared as scrollable, fetch operations such as FETCH LAST, FETCH RELATIVE n are possible in the calling application.
8. Close the cursor after receiving an SQLCODE of +100 that indicates you have fetched all the rows. Closing the cursor is a good practice to ensure that resources are freed and threads are reusable.

11.8 Handling result sets using Global Temporary Tables

If you need to pass results back that are not the result of an SQL query, such as the contents of a data set or messages from a command execution, and you do not want to store that data permanently, you can return the result set in a created temporary table. See “Special considerations” on page 144 why we recommend usage of created temporary tables (CTT) over declared temporary tables (DTT).

An instance of a created temporary table exists for the lifetime of a unit of work, and only the calling application and the stored procedure can access the instance. An instance is created when a temporary table is first referenced in an OPEN, SELECT, INSERT, or DELETE SQL statement. This eliminates the need for logging and locking, and makes SQL statements that use temporary tables very fast.

In order to use a created temporary table to pass back a result set, you have to define it first. Example 11-23 shows how to define a created temporary table to pass back a list of employees for a specific department.

Example 11-23 Statement to define a created GLOBAL TEMPORARY table

```
CREATE GLOBAL TEMPORARY TABLE DEVL7083.RSETP_TBL_OUT
  ( EMPNO CHAR(6) NOT NULL,
    FIRSTNME VARCHAR(12) NOT NULL,
    MIDINIT CHAR(1) NOT NULL,
    LASTNAME VARCHAR(15) NOT NULL,
    HIREDATE DATE,
    SALARY DEC(9,2))
    CCSID EBCDIC;
```

Example 11-24 shows the function query_dept that queries all the employees from a department and inserts the rows into a created temporary table.

Example 11-24 Helper function query_dept

```
 char * query_dept(char * pdeptno)
 {
   memset(h_deptname, NULLCHAR, sizeof(h_deptname));
   strcpy(h_deptno, pdeptno);

   EXEC SQL SELECT DEPTNAME
              INTO :h_deptname
              FROM DEPT
             WHERE DEPTNO = :h_deptno;
   if (SQLCODE != 0)
   {
     sql_error(ERR_SELECT_DEPTNAME);
     return h_deptname;
   }

   EXEC SQL OPEN DEPT_CSR;
   if (SQLCODE != 0)
   {
     sql_error(ERR_OPEN_DEPT_CSR);
     return h_deptname;
   }

   while (TRUE)
```

```
   {
     EXEC SQL FETCH DEPT_CSR
               INTO :h_empno, :h_firstnme, :h_midinit, :h_lastname,
                    :h_hiredate:i_hiredate, :h_salary:i_salary;
     if (SQLCODE == 0)
     {
EXEC SQL INSERT INTO RSETP_TBL_OUT
                            (EMPNO, FIRSTNME, MIDINIT,
                             LASTNAME, HIREDATE, SALARY)
                     VALUES (:h_empno, :h_firstnme, :h_midinit,
                             :h_lastname, :h_hiredate:i_hiredate,
                             :h_salary:i_salary);
if (SQLCODE != 0)
       {
         sql_error(ERR_INSERT_RSETP_TBL_OUT);
         return h_deptname;
       }
     }
     else if (SQLCODE == 100)
       break;
     else
     {
       sql_error(ERR_FETCH_DEPT_CSR);
       return h_deptname;
     }
   }

   EXEC SQL CLOSE DEPT_CSR;
   if (SQLCODE != 0)
   {
     sql_error(ERR_CLOSE_DEPT_CSR);
     return h_deptname;
   }

   return h_deptname;
 }
```

The result cursor was defined as shown in Example 11-25. The above example can be greatly simplified by just opening a cursor on the department table and not even bothering with a global temporary table. However, in most cases you will process data between fetching the data from the cursor, and inserting rows into the output table, which requires your application to be structured as in the example.

Example 11-25 Cursor declarations

```
EXEC SQL DECLARE OUT_CSR             /* Result set cursor          */
          CURSOR WITH RETURN WITH HOLD FOR
          SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, HIREDATE, SALARY
            FROM RSETP_TBL_OUT
         ORDER BY LASTNAME, FIRSTNME;
EXEC SQL DECLARE DEPT_CSR
          CURSOR FOR
          SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, HIREDATE, SALARY
            FROM EMP
           WHERE WORKDEPT = :h_deptno;
```

After calling the function query_dept to insert rows into the global temporary table, the result set cursor needs to be opened as shown in Example 11-26.

Example 11-26 Returning a result set from the stored procedure

```
  /*****************************************************************/
  /* Query information.                                            */
  /*****************************************************************/
  if (rc < RETSEV)
    pdeptname = query_dept(pdeptno);

  /*****************************************************************/
  /* Return results.                                               */
  /*****************************************************************/
  for (i = 0; i < 4; i++)             /* Set all output params NULL */
    locind[i] = -1;

  if (rc < RETSEV)                    /* Open result set cursor     */
  {
    EXEC SQL OPEN OUT_CSR;
    if (SQLCODE != 0)
      sql_error(ERR_OPEN_OUT_CSR);
  }

  if (rc < RETSEV)                    /* Return department name     */
  {
    strcpy((char *)argv[2], pdeptname);
    locind[1] = 0;
  }

  if (rc == RETOK)                    /* Return return code         */
    strcpy(errmsg[0], INF_COMP);
  *(int *)argv[3] = rc;
  locind[2] = 0;

  if (errmsg[0][0] != BLANK)           /* If error message exists   */
  {
    pcurbyte = argv[4];
    for (i = 0; i < DATA_DIM + 1; i++)
    {
      for (j = 0; (errmsg[i][j] != NULLCHAR && j < MSGROWLEN); j++)
        *pcurbyte++ = errmsg[i][j];
      if (j > 0)
        *pcurbyte++ = LINEFEED;
    }
    *pcurbyte = NULLCHAR;
    locind[3] = 0;
  }

  /* Return indicator variables */
  pind = (short int *)argv[5];         /* Locate and recast arg     */
  for (i = 0; i < 4; i++)              /* Copy over null-ind array  */
  {
    *pind = locind[i];
    pind++;
  }
}
```

11.9 Changing the security context in a C stored procedure

If your stored procedure needs to change its identity to a different identity to access an external resource, the __login function can be used as shown in Example 11-27. Once changed, the process should not revert back to a previous identity.

Stored procedures that use the __login function to switch users require daemon authority. Also, if the BPX.DAEMON facility class is active, the stored procedure loaded into the WLM address space must have been defined to RACF program control. The new user ID also has to have an OMVS segment defined. When you specify __LOGIN_CREATE, a process level security environment is established for the calling process and changed to the user ID and password provided.

Example 11-27 Changing identity

```
#define _OPEN_SYS
#include <unistd.h>

#define     __LOGIN_CREATE         1
#define     __LOGIN_USERID         1

change_user()
{
  int userIDlen = strlen(user_id);
  int pswdlen   = strlen(user_pswd);

  rc =__login(__LOGIN_CREATE
             ,__LOGIN_USERID
             ,userIDlen      /* identity_length    */
             ,user_id        /* identity */
             ,pswdlen        /* pass_length */
             ,user_pswd      /* pass  */
             ,0              /* Not used presently  */
             ,NULL           /* Not used presently  */
             ,0              /* Not used presently  */
             );
  if ( rc != 0)
  {
    strcpy(errmsg[0], ERR_LOGIN);
    sprintf(errmsg[1], " %s", strerror(errno));
    rc = RETSEV;
  }
}
```

You need to provide a secure method of transmitting a user ID and password to the stored procedure. DB2 UDB for z/OS Version 8 supports clients using data stream encryption, which is one way to securely transmit a user ID and password. You can also insert a user ID and password into a control table on the server where your stored procedure is running, and only give certain users SELECT authority on that table. This method is probably a more flexible design choice that ensures that users can interact with only the external resource through the stored procedure.

11.10 Summary

In this chapter, we have discussed when to use C for writing stored procedures. By providing sample code, we have highlighted the following important points:

- The elements that a well written and maintainable C stored procedure should contain
- How to handle parameters that allow NULL values
- How to handle result sets in the stored procedure and in the calling program
- How to use created temporary tables to return result sets

The sample code is available as described in B.1.3, “Sample C programs” on page 889.

12

REXX programming

In this chapter we focus on the development of stored procedures using REXX. REXX is used quite often for a quick application solution, and is the favorite language for system programmers and DBAs. As of DB2 V8, you do not have to order the REXX interface feature; the REXX interface is included in every DB2 license.

The performance of REXX programs accessing DB2 tables is improved in V8 in the case of programs issuing large numbers of SQL statements.

The improvement is due to two changes:

- DB2 V8 avoids loading and chaining control blocks for each REXX API invocation; they are now built only at initialization time and then pointed to for the successive API calls. This has reduced the CPU time.
- The REXX exec points to the DSNREXX code. The change now loads the modules and keeps them in local application address space for the duration of the job whenever they have not been preloaded in the link pack area (LPA). This avoids accessing the modules again from disk.

In this chapter we refer to two simple applications accessing sample tables:

- The first for retrieving employee information for a specific employee number
- The second for retrieving a list of employees for a specific department

If you are not familiar with the REXX/DB2 interface, refer to “Coding SQL statements in a REXX application” in Chapter 9 of *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841 for details.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions can be found in Appendix B, “Additional material” on page 887.

Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 24 to decide what components are applicable to your environment.

This chapter contains the following:

- Verify the REXX environment
- Passing parameters
- Preparing and binding a REXX stored procedure
- Actions that the calling application must take
- Actions that the stored procedure must take
- Handling multiple result sets

12.1 Verify the REXX environment

Before developing the stored procedure, it is important to have a clear understanding of the various steps that must be completed for a stored procedure to execute successfully. These steps are covered in detail in the rest of the book; we will simply list them here for convenience. They are:

1. The WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 39 for details. This environment must allow only one concurrent execution of tasks by specifying NUMTCB=1. If you attempt to run multiple REXX stored procedures in a WLM environment, you will receive a message:

   ```
   +DSNX993I  DSNX9REX CALL TO REXX PROCEDURE WITH EXTERNAL NAME ... FAILED,FUNCTION =
   IRXEXEC RC = 00000064 RSN = 00000000
   ```

   In addition, the calling application receives an error code 00E79106 and an SQLCODE -471.

2. In addition, the JCL must contain a DD statement for ddname SYSEXEC:

   ```
   SYSEXEC  DD  DISP=SHR,DSN=SG247083.DEVL.CLIST
   ```

3. The LE environment must be set up. See Chapter 5, “Language Environment setup” on page 47 for details.

4. The stored procedure must be defined to DB2. Note in particular that if the stored procedure is designed to return result sets, the maximum number of result sets that can be returned is specified in the definition. See Chapter 9, “Defining stored procedures” on page 91 for details.

5. Develop the stored procedure. Note that for REXX, you do not prepare the stored procedure and bind a package even when it has SQL. See 12.3, “Preparing and binding a REXX stored procedure” on page 177 for details.

6. Grant the necessary privileges to the authorization ID of the user that executes the stored procedure. See Chapter 7, “Security and authorization” on page 65 for details.

7. Develop the calling application if needed or use IBM Data Studio to call and test your stored procedure.

Also, see Chapter 16, “Debugging” on page 313 for details on testing and debugging.

12.2 Passing parameters

Example 9-5 on page 109 is our first sample REXX stored procedure. More examples are available in Appendix B, “Additional material” on page 887.

A stored procedure can receive and send back parameters to the calling application. When the calling application issues an SQL CALL to the stored procedure, DB2 builds a parameter list based on the parameters passed in the SQL call, and the information specified when the stored procedure is initially defined. For a REXX stored procedure retrieving information about a specific employee, the parameter list specified when defining the stored procedure is shown in bold in Example 12-1.

Example 12-1 Sample REXX parameter list

```
CREATE PROCEDURE DEVL7083.EMPDTLSR
(
 IN  PEMPNO        CHAR(6)
,OUT PARMOUT       VARCHAR(305)
)
```

```
RESULT SETS 0
EXTERNAL NAME EMPDTLSR
LANGUAGE REXX
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB9AREXX
STAY RESIDENT NO
COLLID DSNREXUR
PROGRAM TYPE MAIN
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF)'
COMMIT ON RETURN NO
```

Restriction: Note that a REXX stored procedure can have at most one parameter defined as OUT or INOUT and the definition below would be *invalid*:

```
CREATE PROCEDURE DEVL7083.EMPDTLSR
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
```

This definition specifies whether the parameter is IN (input to the stored procedure), OUT (output from the stored procedure), or INOUT (input to and output from the stored procedure). It also specifies the data type and size of each parameter. This list must be compatible with the parameter list in the calling application.

Note that the single OUT/INOUT parameter must be the last one in the list.

REXX stored procedures have *no explicit LINKAGE section* and the passing of arguments is accomplished using the normal REXX conventions. For example, an input parameter is received as:

```
PARSE UPPER ARG PEMPNO
```

An output parameter is returned as:

```
RETURN PARMOUT
```

Note also that PARAMETER STYLE SQL is not allowed, and hence neither is DBINFO in REXX stored procedures.

12.3 Preparing and binding a REXX stored procedure

No special processing is needed for preparing a REXX stored procedure except to note the following requirements:

- You cannot execute the ADDRESS DSNREXX CONNECT and ADDRESS DSNREXX DISCONNECT commands. This is because DB2 establishes the connection for you when you execute an SQL statement.
- Just like any other REXX EXEC procedures, no precompile or compile is needed.

REXX stored procedures do not require a package or plan to execute. REXX stored procedures are not precompiled nor does any package have to be bound. They are executed using one of four packages that are bound during the installation of DB2 REXX Language Support. The package that DB2 uses when the stored procedure executes depends on the isolation level at which the stored procedure runs. See Table 12-1.

Table 12-1 REXX packages

Collection name	Isolation level
DSNREXRR	Repeatable read (RR)
DSNREXRS	Read stability (RS)
DSNREXCS	Cursor stability (CS)
DSNREXUR	Uncommitted read (UR)

The isolation level depends on the PACKAGE PATH or COLLID value specified in the CREATE PROCEDURE statement. If NO COLLID is specified then an appropriate DSNREXX package should be included in the collection ID of the caller or you can set the special registers CURRENT PACKAGE PATH or CURRENT PACKAGESET to the collection(s) you want to use.

12.4 Actions that the calling application must take

The calling application must initialize all passed parameters and call the stored procedure. The call is shown in Example 12-2.

Example 12-2 REXX calling application

```
EXECSQL
     CALL EMPDTLSR( :PEMPNO
                   ,:PFIRSTNME
                   ,:PMIDINIT
                   ,:PLASTNAME
                   ,:PWORKDEPT
                   ,:PHIREDATE
                   ,:PSALARY
                   ,:PSQLCODE
                   ,:PSQLSTATE
                   ,:PSQLERRMC
                  )
END-EXEC.
```

Notice that when the REXX stored procedure parameters include a *nullable field*, an indicator variable must be passed. In this case notice that there is no comma between the host variable and the variable indicator.

The statement would look like this:

```
EXECSQL
     CALL EMPWNULL( :PEMPNO INDICATOR :PEMPNOIND,:FIRSTNME, ....
```

12.5 Actions that the stored procedure must take

The stored procedure behaves just like any subprogram, taking action based on input parameters (if any), and setting the values of the output parameters (if any). If the stored procedure must return a result set, additional processing is required, and this is discussed in 12.6, “Handling multiple result sets” on page 178.

If you want to add a statement about debugging a REXX stored procedure, you can either:

- Add the following REXX statement in the stored procedure:

  TRACE R

- Or add SAY statements and then use SDSF to look at the SYSTSPRT output in the stored procedure address space.

12.6 Handling multiple result sets

When the stored procedure returns a small number of parameters, it is much simpler to avoid result sets altogether, returning them as parameters as discussed above. When the stored procedure must return result sets, each consisting of multiple rows, there are two basic alternatives:

- Handling a fixed number of result sets for which you know the contents
- Handling a variable number of result sets, for which you do not know the contents

The first alternative is simpler to develop, but the second alternative is more general and requires minimal modifications if the calling program or stored procedure changes.

The following steps are required to handle result sets:

1. When defining the stored procedure to DB2 (see Chapter 9, “Defining stored procedures” on page 91), specify the maximum number of result sets which can be generated by the stored procedure.
2. In the stored procedure, declare and open a cursor for each result set. Note that the stored procedure must not fetch rows from the cursor nor close the cursor. Each such cursor must be declared using the WITH RETURN clause. In REXX, cursors C1 through C100 are declared with a default attribute of WITH RETURN.
3. In the calling program, process the rows as discussed in 10.2.8, “Handling result sets in the calling program” on page 128.

Example 12-3 provides sample REXX code to process a result set.

Example 12-3 REXX code for result set processing

```
...
ADDRESS DSNREXX "EXECSQL  CALL :PROC"
If SQLCODE <> 0 Then CALL SQLCA

ADDRESS DSNREXX
"EXECSQL  ASSOCIATE LOCATOR (:RESULT) WITH PROCEDURE :PROC"
If SQLCODE <> 0 Then Call SQLCA
```

```
say sqlca
ADDRESS DSNREXX
"EXECSQL ALLOCATE C101 CURSOR FOR RESULT SET :RESULT"
If SQLCODE <> 0 Then Call SQLCA
Do Until(SQLCODE <> 0)
ADDRESS DSNREXX
"EXECSQL FETCH C101 INTO :ZONE"
say zone
End
If SQLCODE <> 0 Then Call SQLCA
ADDRESS DSNREXX
"EXECSQL CLOSE C101”
If SQLCODE <> 0 Then Call SQLCA
RETURN
SQLCA:
TRACE 0
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRM ='SQLERRMC
SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD ='SQLERRD.1','',
            || SQLERRD.2','',
            || SQLERRD.3','',
            || SQLERRD.4','',
            || SQLERRD.5','',
            || SQLERRD.6

SAY 'SQLWARN ='SQLWARN.0','',
            || SQLWARN.1','',
            || SQLWARN.2','',
            || SQLWARN.3','',
            || SQLWARN.4','',
            || SQLWARN.5','',
            || SQLWARN.6','',
            || SQLWARN.7','',
            || SQLWARN.8','',
            || SQLWARN.9','',
            || SQLWARN.10
SAY 'SQLSTATE='SQLSTATE
EXIT
```

13

Java stored procedures

Java and DB2 for z/OS work together in order to allow you to run mission critical applications written in Java.

In this chapter we describe how to set up the environment for Java stored procedures, and the steps to develop and debug them. We assume that you have a basic understanding of the Java language.

In this chapter we will look into:

- Overview of Java stored procedures
- Recent changes for Java stored procedures
- Setting up the environment for Java stored procedures
- Persistent Reusable JVM
- Considerations on static variables

We will then look at the steps for creating a new Java stored procedure:

- Preparing Java stored procedures
- DDL for defining a Java stored procedure

Once created, we can execute the stored procedure. The next sections will assist you in debugging the stored procedure and show a completed Java stored procedure as well.

- Debugging JDBC and SQLJ
- Java sample JDBC stored procedure
- Java sample SQLJ stored procedure

Finally, we will give steps for migrating your Java stored procedures from the Legacy Driver to the IBM Universal driver in:

- Migrating stored procedures to use the new JCC driver

13.1 Overview of Java stored procedures

Java has become a first-class member of the programming language portfolio on the mainframe. Developing in Java for the z/OS platform combines the performance and reliability of the mainframe with the sophisticated development tools available on the workstation. Java is also the ideal development environment for enabling your DB2 system on z/OS for the Internet.

Details on how Java and DB2 for z/OS can work together and form a strong combination that can run your mission critical applications are reported in *DB2 for z/OS and OS/390: Ready for Java,* SG24-6435. That book starts from the basics and covers the new IBM Universal Driver for SQLJ and JDBC, IBM's new JDBC driver implementation, supporting both Type 2 and Type 4 driver connectivity to the members of the DB2 family, including DB2 for z/OS, and DB2 for Linux, UNIX and Windows. Another source of reference information is *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java,* SC18-9842-01. In this book we concentrate on *V9 Java stored procedures*.

DB2 V9 supports the development of *interpreted Java stored procedures*. The IBM Data Studio application development tool only supports creating interpreted Java stored procedures on DB2 V8 and DB2 V9. You can develop interpreted Java stored procedures using both JDBC or SQLJ methods. We recommend getting started by developing stored procedures using JDBC methods. The setup and application preparation process for an SQLJ stored procedure includes more steps than those for JDBC stored procedures. Once you become familiar with JDBC stored procedures you can expand to using SQLJ procedures. SQLJ stored procedures provide better performance since they use static SQL, unlike JDBC stored procedures, which make dynamic SQL calls.

We developed several JDBC and SQLJ Java stored procedures. Refer to Chapter 3, "Our case study" on page 23 for a comprehensive list of JDBC and SQLJ stored procedures.

13.2 Recent changes for Java stored procedures

Since the publication of this book's predecessor, *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083, there have been several changes in both z/OS and JDBC that affect DB2 9 for z/OS Java stored procedure processing. The following list summarizes these changes and the DB2 level affected.

- IBM's zAAP processor can help simplify the server infrastructures and improve operational efficiencies for Java stored procedures (V8 and V9).

  For more information, see:

  `http://www.ibm.com/servers/eserver/zseries/zaap`

- JDBC/SQLJ Driver for OS/390® and z/OS is no longer supported.

  All Java application programs and Java routines that are currently written to work with the JDBC/SQLJ Driver for OS/390 and z/OS need to be modified to work with the IBM DB2 Driver for JDBC and SQLJ (formerly known as the DB2 Universal JDBC Driver and also known as the IBM Data Server Driver for JDBC and SQLJ). The steps for migrating JDBC and SQLJ applications from the legacy JDBC/SQLJ Driver for OS/390 and z/OS to the IBM DB2 Driver for JDBC and SQLJ can be found in *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for JAVA,* SC18-9842.

- Java stored procedures no longer run in resettable Java Virtual Machines.

For more information about resettable JVMs and for a full list of actions that are prevented, see the “Unresettable actions” topic of *Persistent Reusable Java Virtual Machine User’s Guide*, SC34-6201.

> **Note:** This feature is also available for DB2 V8 with APAR PK09213. See 13.4, “Persistent Reusable JVM” on page 193 for more information.

- Java shared classes

  This feature, available with JDK 1.5, offers a transparent and dynamic means of sharing all loaded classes, both system and application classes through a fixed-size area of shared memory called the “class cache”. The feature is available for both V8 and V9. Details about Java shared classes are discussed in the article “Java Technology, IBM Style: Class sharing” available at:

  `http://www.ibm.com/developerworks/java/library/j-ibmjava4/#4`

- Building Java stored procedures using the build utility DSNTJSPP is no longer supported
- Multiple or common jars for an application

  Supporting jars can now be referenced by a Java stored procedure without including their jar paths in the CLASSPATH setting of the Java stored procedure. This exploits using the DB2-supplied stored procedure, SQLJ.ALTER_JAVA_PATH.

- Debugging using Developer Workbench or IBM Data Studio Unified Debugger

  These new tools provide an interactive visual interface for debugging Java stored procedures. See “28.2, “The Unified Debugger” on page 738” for more information about this.

13.3 Setting up the environment for Java stored procedures

In this section, we describe prerequisites and steps that you need to perform in order to set up the environment for running DB2 V9 for z/OS Java stored procedures.

13.3.1 Prerequisite software for Java stored procedures

The major prerequisites are:

- DB2 for z/OS at Version 9
- z/OS Version 1 Release 4
- The IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4.2 or later

> **Important:** The following functions require Java 2 Technology Edition, V5 or later.
>
> - Accessing DB2 tables that include DECFLOAT columns
> - Using Java support for XML schema registration and removal

For information on installing the Java SDK, refer to the following URL:

`http://www-1.ibm.com/servers/eserver/zseries/software/java/`

13.3.2 Ensuring that the Java SDK is at the right level

To ensure that you have the appropriate Software Developers Kit (SDK) release, do the following:

- Determine what the JAVA_HOME setting is. This is typically found in the data set pointed to by the JAVAENV DD statement in the WLM application environment procedure for your Java stored procedures. Depending on your installation, your systems programmer may have installed the Java libraries into a different directory. Typically, JAVA_HOME is set to `/usr/lpp/java/J5.0/.`
- Log on to UNIX System Services (USS). You can either use TSO -> **OMVS** or **telnet** to your z/OS machine from Windows.
- Set the PATH environment variable to the Java bin directory. We do this by issuing the export command to assign the PATH variable to the Java bin directory:

  ```
  => export PATH=/usr/lpp/java/J5.0/bin:$PATH
  ```

- Issue the following command to check the JDK version:

  ```
  => java -version
  ```

You should get the following messages on your terminal.

```
PAOLOR5 @ SC63:/>java -version
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build pmz31devifx-20070801 (SR
5a))
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 z/OS s390-31 j9vmmz3123ifx-20070627
(JIT enabled)
J9VM - 20070614_12948_bHdSMr
JIT  - 20070419_1806_r8
GC   - 200704_19)
JCL  - 20070731a
```

Alternatively, you can check the JDK version from the workstation client by creating and using the following UDF in SPUFI on DB2 for z/OS.

```
CREATE FUNCTION SYSADM.JVMVERS  (VARCHAR(50))
 RETURNS VARCHAR(100)
 FENCED NO SQL
 LANGUAGE JAVA
 SPECIFIC JVMVERS
 EXTERNAL NAME 'java.lang.System.getProperty'
 WLM ENVIRONMENT DB9AWLMJ
 NO EXTERNAL ACTION
 NO FINAL CALL
 PROGRAM TYPE SUB
 PARAMETER STYLE JAVA;
```

Then, issuing the following command in a DB2 9 command line window shows the result; see Example 13-1:

```
SELECT  SYSADM.JVMVERS('java.vm.name') FROM SYSIBM.SYSDUMMY1;
SELECT  SYSADM.JVMVERS('java.version') FROM SYSIBM.SYSDUMMY1;
```

Example 13-1 Verifying the JVM version and name from your workstation

```
C:\>DB2 CONNECT TO DB9A USER PAOLOR5 USING PUP4SALE

   Database Connection Information

 Database server        = DB2 OS/390 9.1.5
 SQL authorization ID   = PAOLOR5
 Local database alias   = DB9A
```

```
C:\>db2 select sysadm.jvmvers('java.vm.name') from sysibm.sysdummy1
------------------------------------------------------------------------

IBM J9 VM
1 record(s) selected.
C:\>db2 select sysadm.jvmvers('java.version') from sysibm.sysdummy1
------------------------------------------------------------------------
1.5.0
1 record(s) selected.
```

13.3.3 Checking the DB2 JDBC and SQLJ libraries for USS

When you install DB2, include the steps for allocating the HFS directory structure and using SMP/E to load the JDBC and SQLJ libraries. See *DB2 9 for z/OS Program Directory,* GI10-8737-00 for information on allocating and loading DB2 data sets. To check for the DB2 libraries, you need to change your working directory to the DB2 home directory, /usr/lpp/db2/db2910, and issue the **list** directory command. In our case the DB2 home directory was /usr/lpp/db2/db9a. However, the JDBC and SQLJ libraries are installed in db2910_jdbc:

```
=> cd /usr/lpp/db2/db9a/db2910_jdbc
=> ls
```

You should see the following directories:

```
IBM       README    bin       classes   lib       samples   userproc
```

13.3.4 Checking the build level of the SQLJ/JDBC driver

- Create the following UDFs to assist you in determining the driver name and version of the IBM Universal driver installed on the DB2 9 for z/OS, as shown in Example 13-2.[1]

Example 13-2 UDFs to determine the driver name and version of the IBM Universal driver

```
CREATE FUNCTION SYSADM.JAVDRVV   ()
 RETURNS VARCHAR(100)
 FENCED NO SQL
 LANGUAGE JAVA
 SPECIFIC JAVDRVV
 EXTERNAL NAME 'com.ibm.db2.jcc.DB2Version.getVersion'
 WLM ENVIRONMENT DB9AWLMJ
 NO EXTERNAL ACTION
 NO FINAL CALL
 PROGRAM TYPE SUB
 PARAMETER STYLE JAVA;

 CREATE FUNCTION SYSADM.JAVDRVN   ()
  RETURNS VARCHAR(100)
  FENCED NO SQL
  LANGUAGE JAVA
  SPECIFIC JAVDRVN
  EXTERNAL NAME 'com.ibm.db2.jcc.DB2Version.getDriverName'
  WLM ENVIRONMENT DB9AWLMJ
  NO EXTERNAL ACTION
  NO FINAL CALL
  PROGRAM TYPE SUB
```

[1] You can also create these UDFs in another platform that uses the IBM Universal driver.

```
  PARAMETER STYLE JAVA;
```

- Then issue these queries to determine the JCC Driver name and version:

```
SELECT  SYSADM.JAVDRVN() FROM SYSIBM.SYSDUMMY1;
SELECT  SYSADM.JAVDRVV() FROM SYSIBM.SYSDUMMY1;
```

Example 13-3 shows the output of the above commands.

Example 13-3 Checking the driver name and version

```
C:\>DB2 CONNECT TO DB9A USER PAOLOR5 USING PUP4SALE

   Database Connection Information

 Database server        = DB2 OS/390 9.1.5
 SQL authorization ID   = PAOLOR5
 Local database alias   = DB9A


C:\>db2 select sysadm.javdrvn() from sysibm.sysdummy1
-------------------------------------------------------
IBM DB2 JDBC Universal Driver Architecture

  1 record(s) selected.

C:\>db2 select sysadm.javdrvv() from sysibm.sysdummy1
-------------------------------------------------------
3.4.77

  1 record(s) selected.
```

13.3.5 DESCSTAT

On DB2 for z/OS, set subsystem parameter DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPF. See Part 2 of the DB2 Installation Guide for information on setting DESCSTAT. This step is necessary for SQLJ support.

13.3.6 Setting up the WLM procedure

You need to have a WLM JCL procedure corresponding to the WLM application environment defined for executing the Java stored procedures. The sample procedure that we used is shown in Example 13-4.

Example 13-4 V9 WLM procedure for running Java stored procedures

```
//*************************************************************
//*    JCL FOR RUNNING THE WLM-ESTABLISHED STORED PROCEDURES
//*    ADDRESS SPACE
//*       RGN     -- THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//*       DB2SSN  -- THE DB2 SUBSYSTEM NAME.
//*       NUMTCB  -- THE NUMBER OF TCBS USED TO PROCESS
//*                  END USER REQUESTS.
//*       APPLENV -- THE MVS WLM APPLICATION ENVIRONMENT
//*                  SUPPORTED BY THIS JCL PROCEDURE.
//*
//*************************************************************
//DB9AWLMJ PROC RGN=0K,APPLENV=XXXXXXXX,DB2SSN=DB9A,NUMTCB=20
```

```
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=DB9AU.RUNLIB.LOAD
//         DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DISP=SHR,DSN=DB9A9.SDSNEXIT
//         DD  DISP=SHR,DSN=DB9A9.SDSNLOAD
//         DD  DISP=SHR,DSN=DB9A9.SDSNLOD2
//JAVAENV DD DISP=SHR,DSN=DB9AU.JSPENV
//JSPDEBUG DD SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//JAVAOUT  DD  PATH='/SC63/sg247083/JAVAOUT.TXT',
// PATHOPTS=(ORDWR,OCREAT,OAPPEND),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
//JAVAERR  DD  PATH='/SC63/sg247083/JAVAERR.TXT',
// PATHOPTS=(ORDWR,OCREAT,OAPPEND),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
```

Be aware of the following points while defining the WLM procedure for Java:

- For every NUMTCB, WLM would create a JVM in the WLM address space. In resettable mode, with JDK 1.4.2, we limited this to a low number, an optimum of 6 or 7 for running Java stored procedures in production. In non-resettable mode with JDK 1.5, we can now set the NUMTCB to a higher number. The recommended number is between 20 and 40.
- The WLM address space for running Java stored procedures should be solely dedicated for Java stored procedures. You should not allow stored procedures of other languages such as COBOL, SQL etc., to use the Java application environment and address space. You want stored procedures with similar performance and resource usage characteristics running in the same WLM environment.
- If the LE runtime SCEERUN library is not included in your system LINKLIST, you need to uncomment the STEPLIB DD for SCEERUN. In this case you might consider putting it in Library Lookaside (LLA) to reduce I/O.
- Apart from including SDSNEXIT, SDSNLOAD libraries in the STEPLIB, do not forget to include the SDSNLOD2 library (it contains the DLLs for the Java drivers). Also, ensure that you have one non APF authorized data set in your STEPLIB. In our case we included CBC.SCBCOMP lib; it can be any library of your choice.
- The JAVAENV DD statement specifies a data set that contains environment variables that define system properties for the execution environment. The presence of this DD statement indicates to DB2 that the WLM environment is for Java stored procedures. For an interpreted Java routine, this data set must contain the environment variable JAVA_HOME. This environment variable indicates to DB2 that the WLM environment is for interpreted Java routines. A detailed discussion of the contents of JAVAENV is in 13.3.7, “Setting up the JAVAENV data set for Java stored procedure execution” on page 188.
- The JAVAOUT and JAVAERR DD statements are optional, and are helpful for debugging Java stored procedures. Specify a data set in your Unix System Services (USS) to capture the output from your SYSOUT and SYSERR. If the file does not exist, USS will create it. Otherwise, the new output will be appended to the existing file. The PATHOPTS and PATHMODE specify the permissions for this data set. More information about these keywords can be found in *z/OS V1R7.0 MVS JCL Reference,* SA22-7597-09.
- JSPDEBUG DD statement specifies a data set into which DB2 puts information that you can use to debug your stored procedures. The information that DB2 collects can be very helpful in debugging setup problems, and also contains key information that you need to

provide when you submit a problem to IBM Service. Example 13-5 shows the sample JSPDEBUG output produced when a stored procedure is invoked.

Important: You should comment out the JSPDEBUG DD statement during production in order not to degrade the performance of your stored procedure. Information regarding each invocation of a Java stored procedure is written to the JSPDEBUG data set by DB2.

Example 13-5 JSPDEBUG output from an invocation of a stored procedure

```
-----------------------------------------------------
Entered PK49236 version at time:  Fri Nov 16 23:50:46 2007

Default EBCDIC encoding is 37; as CCSID char: 'Cp037'
Java method is defined to be stored in a jar.
Generated signature before convert: (Ljava/lang/String;ŶLjava/lang/String;ŶLjava
ang/String;ŶLjava/math/BigDecimal;ŶLjava/sql/Date;ŶLjava/lang/String;)V
Processing IN and INOUT parameters of the Java method
  parm 1 is String: '000100                              ' CCSID: 37
invoking class: EmpDtlsJ, method: GetEmpDtls
Back from Call: Processing time was 0.223500
Processing OUT and INOUT parameters of the Java method
  parm 2 is String(8); 'THEODORE' CCSID: 37
  parm 3 is String(1); 'Q' CCSID: 37
  parm 4 is String(7); 'SPENSER' CCSID: 37
  parm 5 is String(3); 'E21' CCSID: 37
  parm 6 is BigDecimal (DECFLOAT(34)): 26150.00
  parm 7 is Date string: '1980-06-19' CCSID: 37
  parm 8 is String(1); ' ' CCSID: 37
Number of result sets is 0
Return Status: Execution=0, Debug=0
-----------------------------------------------------
```

- For debugging purposes, or in general for gathering information from the stack trace in the event of unhandled Java exceptions, specify data sets for JAVAOUT and the JAVAERR DD cards. These data sets are explained in 13.9, “Debugging JDBC and SQLJ” on page 209. They are only required if you plan to debug your Java stored procedures using the System.out.println method.

13.3.7 Setting up the JAVAENV data set for Java stored procedure execution

The WLM procedure where Java stored procedures execute requires a JAVAENV DD. This data set defines the Java environment variables that will be used.

This JAVAENV data set should have the characteristics shown in Table 13-1.

Table 13-1 JAVAENV definition

JAVAENV data set characteristics		
LRECL	255	This maximum is limited by LE. 245 bytes usable. If more than 245 bytes included, unpredictable results will occur.
RECFM	VB	
ORGANIZATION	PS	

The contents of the JAVAENV data set that was used in our lab is shown in Example 13-6.

Example 13-6 Contents of JAVAENV - DB9AU.JAVAENV file

```
ENVAR("JAVA_HOME=/usr/lpp/java/J5.0",
"JCC_HOME=/usr/lpp/db2/db9a/db2910_jdbc",
"CLASSPATH=/usr/lpp/db2/db9a/db2910_jdbc/userproc",
"DB2_BASE=/usr/lpp/db2/db9a/db2910_base",
"RESET_FREQ=-1"),
XPLINK(ON)
```

All the environment variables need to be included in this file. Ensure that the total length of all the entries does not exceed 245 bytes (exclude the blanks)[2]. In case your entries exceed the 245 byte limit, you need to take a different approach, as shown in Example 13-7. Here we show an alternate form of JAVAENV definitions.

Example 13-7 Contents of JAVAENV having _CEE_ENVFILE variable

```
ENVAR("_CEE_ENVFILE=/usr/lpp/db2/db9a/evnfile.txt",
"JAVA_HOME=/usr/lpp/java/J5.0",
"DB2_BASE=/usr/lpp/db2/db9a/db2910_base",
"RESET_FREQ=-1"),
XPLINK(ON)
```

The _CEE_ENVFILE variable points to an HFS file that contains most of the environment variables, because this file has no limitation of size. The JAVA_HOME variable must be defined in the JAVAENV data set, and not in the HFS file corresponding to _CEE_ENVFILE. The contents of _CEE_ENVFILE file are shown in Example 13-8. This is a standard UNIX file where each line must start in column 1 and the continuation character is a \.

Example 13-8 Contents of the _CEE_ENVFILE - /usr/lpp/db2/db9a/envfile.txt

```
JCC_HOME=/usr/lpp/db2/db9a/db2910_jdbc
CLASSPATH=/usr/lpp/db2/db9a/db2910_jdbc/userproc
```

You can use _CEE_ENVFILE for overcoming the 245 limit when specifying other environmental variables that tend to be long or transitory in nature, such as JITC_COMPILING and JITC_COMPILEOPT.

13.3.8 Environment variables in the JAVAENV data set

A description of the various environment variables that need to be defined in the JAVAENV data set are mentioned in Table 13-2.

Table 13-2 Contents of a JAVAENV data set

Environment variable	Description
JCC_HOME[a]	This environment variable is set to the location of the JCC driver. For example: JCC_HOME=/usr/lpp/db2/db9a/db2910_jdbc
JAVA_HOME	This environment variable indicates to DB2 that the WLM environment is for Java routines. The value of JAVA_HOME is the highest-level directory in the set of directories that contain the Java SDK. For example: JAVA_HOME=/usr/lpp/java/IBM/J15
DB2_BASE	The DB2 base directory defaults to /usr/lpp/db2910_base. If you did not use the default location, set this environment variable to the directory where your DB2 base directory is installed.

[2] The 245 limit is a restriction of the Language Environment.

Environment variable	Description
CLASSPATH	The directory where you place your compiled stored procedures. A detailed discussion of CLASSPATH can be found in 13.7, “Making the stored procedure class files available to DB2” on page 201
JVMPROPS	You can optionally specify here, the name of a USS file which contains JVM startup options. See “JVMPROPS” on page 190.
RESET_FREQ	Specify a value of -1 to indicate that the JVM is to be started in its non-resettable mode, and is never reset.
HEAP	When debugging with JDK 1.4[a], set this to: (8M,2M,ANYWHERE,KEEP). Otherwise, you can code (NONE) or remove this variable.
WORK_DIR	Optional: If you do not code a JAVAOUT or JAVAERR DD card in your WLM, you can set this environment variable to a valid HFS directory. This will be the default directory for STDOUT and STDERR. The default file names will be, `server_stdout.txt` and `server_stderr.txt`.

a. If the DB2_HOME environment variable is coded in the JAVAENV data set, DB2 for z/OS V8 ignores this. However, in DB2 9 for z/OS, this will cause an error when the WLM is started.

Important: When using JVM 1.4.2, the default native heap size is insufficient for debugging Java stored procedures with the IBM Data Studio. The JVM 1.5 default native heap size is sufficient. JDK1.4.2 heap size therefore has to be increased to (8M,2M,ANYWHERE,KEEP). Do not put this environment variable in the _CEE_ENVFILE. For example, in your JAVAENV file, you can code:

```
MSGFILE(JSPDEBUG,,,,ENQ),
XPLINK(ON),
HEAP(8M,2M,ANYWHERE,KEEP),
ENVAR("_CEE_ENVFILE=/u/oeusr05/CEEOPTIONS.txt")
```

DB2 for z/OS V9 flags the presence of the environment variable DB2_HOME as an error. DB2 for z/OS V8 ignores this variable.

JVMPROPS

JVMPROPS is the environment variable that specifies the name of a z/OS UNIX System Services file that contains startup options for the JVM in which the stored procedure runs. JVMPROPS is the Java stored procedures environment mechanism to set the -Xoptionsfile option.

Example 13-9 shows the contents of the HFS file.

Example 13-9 Contents of the JVMPROPS file

```
# Properties file for JVM for Java stored procedures
# Sets the initial size of middleware heap within non-system heap
-Xms64M
# Sets the maximum size of nonsystem heap
-Xmx128M
#initial size of system heap
-Xinitsh512K
```

For information about JVM startup options, see *“IBM 31-bit and 64-bit SDKs for z/OS, Java 2 Technology Edition, Version 5 SDK and Runtime Environment User Guide”,* available at:

http://www.ibm.com/servers/eserver/zseries/software/java/pdf/sdkguide.zos.pdf

Note: To enable class sharing in JDK 1.5, code the -Xshareclasses option in this file.

13.3.9 Binding the JDBC packages

The IBM Universal Driver requires a set of packages to be bound at the target DB2 system. To bind these packages, run the DB2Binder utility. This utility binds the packages that are used at the database server by the IBM Universal Driver, and grants EXECUTE authority on the packages to PUBLIC.

DB2Binder utility

You can run the DB2Binder from the USS shell or from a workstation DB2 command line prompt connected to the target server.

Before running the utility, ensure that you have the file db2jcc.jar defined in your CLASSPATH. For example, in our environment, we have /usr/lpp/db2/db9a/ db2910_jdbc/classes/db2jcc.jar defined to the CLASSPATH.

Issue the command shown in the USS shell:

```
java com.ibm.db2.jcc.DB2Binder
 -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A
-user PAOLOR5 -password PUP4SALE
-collection DEVL7083
```

The URL options are: -url jdbc:db2://server_name:port_number/database_name.

You can use the backslash (\ ) continuation character at the end of the first line in the above command. When you press Enter, the command line is cleared so that you can continue typing. The line you typed prior to the backslash is displayed in the output area, and the shell prompt changes to > beneath it to indicate that you are continuing a command.

If you have a DB2 client installed, you can code the DB2Binder command in a .bat file and call it from the workstation through a DB2 command window as shown in Example 13-10.

Example 13-10 DB2Binder command from a DB2 command window

```
C:\$WorkDocuments\ZOS_Related>db2binder.bat

C:\$WorkDocuments\ZOS_Related>set CLASSPATH=c:\sqllib\java\db2jcc.jar;c:\sqllib\
java\db2jcc_license_cisuz.jar;c:\sqllib\java\db2jcc_license_cu.jar;

C:\$WorkDocuments\ZOS_Related>java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://v14
ec039.svl.ibm.com:446/STLEC1 -user admf001 -password  n1cetest -collection MYCOLID
...
```

Collection name

The binder binds the packages into the collection specified. If you do not specify the collection name, the binder puts the packages in the NULLID collection. When you define the stored procedure (DDL), the collection name that you specify should have the Universal Driver JDBC packages defined in it.

Note: The collection ID specified in the DB2Binder command needs to be the same ID under which your stored procedure will execute. Furthermore, the DB2Binder command needs to be issued for each collection that contains a package for your Java stored procedure that will be executed.

Example 13-11 shows an alternate way to bind the JDBC packages.

Example 13-11 Sample Job to bind the packages for JCC

```
//JCCSETUP JOB (999,POK),'JAVA COMP',CLASS=A,MSGCLASS=A,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
//JOBLIB   DD DSN=CEE.SCEERUN,DISP=SHR
//JCOMP   EXEC PGM=AOPBATCH,PARM='sh -L'
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDIN    DD *
 java com.ibm.db2.jcc.DB2Binder  \
 -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A  \
-user PAOLOR5 -password PUP4SALE \
-collection DEVL7083
/*
```

13.3.10 Install the DB2-provided metadata stored procedures

Before you can use certain functions in the DB2 Universal JDBC Driver, you need to install the following DB2-provided stored procedures and create a set of tables.

```
SQLCOLPRIVILEGES
SQLCOLUMNS
SQLFOREIGNKEYS
SQLFUNCTIONCOLUMNS
SQLFUNCTIONS
SQLGETTYPEINFO
SQLPRIMARYKEYS
SQLPROCEDURECOLS
SQLPROCEDURES
SQLSPECIALCOLUMNS
SQLSTATISTICS
SQLTABLEPRIVILEGES
SQLTABLES
SQLUDTS
SQLCAMESSAGE
```

The tables you need to create are:

```
SYSIBM.SYSDUMMYU
SYSIBM.SYSDUMMYA
SYSIBM.SYSDUMMYE
```

These tables ensure that character conversion does not occur when Unicode data is stored in DBCLOB or CLOB columns.

The installation job DSNTIJSG creates these stored procedures and tables as part of a new installation or migration, or by job DSNTIJMS, for installations that were installed or migrated

before the procedures were introduced into Version 8. These jobs must be customized before execution, as described in the job prologs.

Prior to running these jobs, you should set the subsystem parameter named DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPF.

See Chapter 7, “Installing the IBM DB2 Driver for JDBC and SQLJ” in *DB2 Version 9.1 for z/OS Installation Guide,* GC18-9846-01, for more information.

13.4 Persistent Reusable JVM

Every Java stored procedure requires a Java Virtual Machine (JVM) to execute. If you have *n* number of stored procedures concurrently executing in a WLM Address Space, you require *n* instances of JVM. To improve the performance for transaction processing, IBM implements Persistent Reusable JVM.

Persistent Reusable Java Virtual Machines speed up the processing of Java applications in transaction processing environments on z/OS systems. Transaction processing in a z/OS environment is characterized by short, repetitive transactions that run in subsystems such as CICS Transaction Servers or DB2 Database Management Systems.

The Persistent Reusable JVM improves transaction processing throughput in such environments by providing the ability to run multiple JVMs within a z/OS address space, providing increased scalability between transactions processed, and JVM resources used. Hundreds or even thousands of transactions can be processed in a JVM. This has the effect of distributing the cost of starting that JVM over all of the transactions processed by the JVM.

To ensure isolation between transactions, each JVM processes only one transaction at a time, and each JVM is created in its own Language Environment (LE) enclave to ensure isolation between JVMs running in parallel. The set of JVMs within an address space is called a JVMSet.

The model of one transaction per JVM implies the recycling of the JVM; that is, create a JVM, run the transaction, and destroy the JVM. However, the startup overhead for a traditional JVM is very high; high-volume transaction processing requires a model that allows serial reuse of a JVM by many transactions, and that destroys and creates a new JVM only when absolutely necessary.

13.4.1 Resettable JVMs

A resettable JVM is defined as one that can be reset to a known state between application programs. Once the JVM has been reset, the next application program that runs is unable to determine whether it is running in a new JVM or a JVM that has been reset. As a result, the program cannot be affected by any actions of a previous program. With this approach, DB2 does not have to recreate a JVM for every transaction before it starts.

However, executing Java routines in a JVM that is started in resettable mode has its limitations, namely:

- The Java stored procedures were limited in what they can do so as not to corrupt the JVM for subsequent users.
- The number of stored procedures you can run in an address space is limited.
- Once every 10 JVM resets, a request is made to the JVM to perform Garbage Collection. This impacts performance.

- Java routines that ran in other platforms required additional porting to run on DB2 for z/OS.

The Java environment variable, RESET_FREQ, determines if the JVM is started in a resettable or non-resettable mode.

- A value greater than zero specifies that the JVM is started in resettable mode and the value this is set to determines the frequency of JVM resets.
- A value of zero indicates that the JVM is run in resettable mode with the default reset frequency. By default, the JVM is run in resettable mode and resets every 256 stored procedure invocations.
- A value less than zero indicates that the JVM is started in non-resettable mode and never reset.

13.4.2 Non-resettable JVMs

With DB2 9, JVMs can be started in non-resettable mode. In this mode,

- There is no need to set the TMSUFFIX environment variable to specify classes that define a tidy-up method to execute each time the JVM is reset.
- The periodic request to perform garbage collection is not needed. Instead, JVMPROPS can be used to specify JVM startup options that influence how garbage collection is done.
- While a resettable JVM prevents extra threads from being left behind after a Java routine executes, this task can be done by WLM, which prevents DB2 from leaving the WLM enclave and requesting more work.

The Java environment variable, RESET_FREQ, determines whether the JVM is started in a resettable or non-resettable mode. When RESET_FREQ specifies a value less than zero, that will indicate that the JVM is to be started in its non-resettable mode, and never reset.

13.5 Considerations on static variables

We advise against using static and non-final variables in Java stored procedures and UDFs. Just do not confuse using *static variables* with using *static routines*; it is required that a Java stored procedure or UDF be defined as a static routine.

The advice against using static variables is given for the following reasons:

- Supporting the use of static variables is explicitly not required by the applicable ANSI/ISO standard.
- It is difficult to guarantee that a sequence of CALLs will be processed by the same JVM.

  For example, suppose that two stored procedures, INITIALIZE and PROCESS, use the same static variable, sv1. INITIALIZE sets the value of sv1, and PROCESS depends on the value of sv1.

  - CALL INITIALIZE. This runs in one JVM and sets the static variable, `sv1`, to a known state or value.
  - CALL PROCESS goes to a different JVM and finds the value of the static variable to be un-initialized.
- While for Java applications, the static variables are initialized or reset whenever the class is loaded, for Java stored procedures and UDFs the programmer can't control which JVM a subsequent stored procedure or UDF will run in. So invoking a stored procedure twice does not guarantee that static variables will have the value set from the previous invocation.

For more information on static variables and other Java programming tips, refer to Chapter 5 of *DB2 Version 9.1 for z/OS, Application Programming Guide and Reference for Java*, SC18-9842-01.

13.6 Preparing Java stored procedures

There are three methods for preparing Java stored procedures:

- Prepare the stored procedure without a JAR file. In here, you simply generate the .class files and ensure that the CLASSPATH contains the location of this class file.
- Prepare the stored procedure to run from a JAR file. Running Java stored procedures from JAR files is recommended.
- Use IBM Data Studio (see Chapter 27, “The IBM Data Studio” on page 643) to prepare the stored procedure. The tooling will prepare the stored procedure to run from a JAR file as well but will perform all the tasks automatically.

In this section we limit our discussion to preparing Java stored procedures without the IBM Data Studio tooling. We also limit our discussion to using UNIX System Services (USS). You need USS because that is where the Java SDK and JDBC drivers reside.

13.6.1 Profile data set

Before you can prepare your SQLJ/JDBC stored procedures, you need to set the environment variables in the profile data set. Every UNIX system has a global HFS profile file named /etc/profile. Environment variables set in this file are available to all UNIX users. On the other hand if you want the environment variables to be visible only to a specific user, then you need to update the user’s “.profile” data set. The user’s profile data set can be found in the user directory. For example, the profile data set for user ID PAOLOR5 would be /u/paolor5/.profile. Example 13-12 shows the contents of user profile data set /u/paolor5/.profile.

Example 13-12 /u/paolor5/.profile data set

```
PATH=/usr/lpp/java/J5.0/bin:$PATH
PATH=/usr/lpp/db2/db9a/db2910_jdbc/bin:$PATH
export PATH
LIBPATH=/usr/lpp/db2/db9a/jcc/lib:$LIBPATH
export LIBPATH
LD_LIBPATH_PATH=/usr/lpp/db2/db9a/jcc/lib:$LD_LIBPATH_PATH
export LD_LIBPATH_PATH
STEPLIB=DB9A9.SDSNEXIT:DB9A9.SDSNLOAD:DB9A9.SDSNLOD2:$STEPLIB
STEPLIB=CEE.SCEERUN:$STEPLIB
export STEPLIB
CLASSPATH=/usr/lpp/db2/db9a/db2910_jdbc/classes/db2jcc.jar:$CLASSPATH
CLASSPATH=/usr/lpp/db2/db9a/db2910_jdbc/classes/db2jcc_javax.jar:$CLASSPATH
CLASSPATH=/usr/lpp/db2/db9a/db2910_jdbc/classes/sqlj.zip:$CLASSPATH
CLASSPATH=/usr/lpp/db2/db9a/db2910_jdbc/classes/db2jcc_license_cisuz.jar:$CLASSPATH
export
CLASSPATH
```

The names of the directory path could vary across installations. Refer to Table 13-3 for a general description of various environment variables. Be aware that the contents of the profile data set are used while preparing a stored procedure or running a Java DB2 application in USS. The profile data set is not used for setting up the stored procedure runtime environment and properties. The JAVAENV data set controls the runtime environment and behavior of Java stored procedures.

Table 13-3 Environment variables

Variable	Typical value	Value used at the ITSO site
LIBPATH	/usr/lpp/db2910_jdbc/lib	/usr/lpp/db2/db9a/jcc/lib:/lib:/usr/lib:.
CLASSPATH	/usr/lpp/db2910_jdbc/classes/	/SC63/sg247083/spjava:/usr/lpp/db2/db9a/db2910_jdbc/classes/db2jcc_license_cisuz.jar:/usr/lpp/db2/db9a/db2910_jdbc/classes/sqlj.zip:/usr/lpp/db2/db9a/db2910_jdbc/classes/db2jcc_javax.jar:/usr/lpp/db2/db9a/db2910_jdbc/classes/db2jcc.jar:
LD_LIBRARY_PATH	/usr/lpp/db2/db9a/jcc/lib	same as LIBPATH
PATH	It needs to be set to the bin sub directory in Java and DB2 directories /usr/lpp/db2/db2910_jdbc/bin	/usr/lpp/db2/db9a/db2910_jdbc/bin: /usr/lpp/java/J5.0/bin: /usr/lpp/java/J1.4/bin:/bin:.

13.6.2 Deciding whether to use JDBC or SQLJ

Before you start coding your Java stored procedure, you may want to decide whether you want to use JDBC or SQLJ or both. Here we present a summary of the differences between them to help you with your decision.

- Portability
  - JDBC is a Java data interface standard that provides access to a wide range of relational databases. Virtually all database vendors have adapted the JDBC specification into their database products. Some IDEs still do not support SQLJ. There is no SQLJ support for most of the common persistence frameworks such as *Hibernate*.
  - SQLJ requires a pre-processing step and bind processing. JDBC does not require these steps.
- Maintainability
  - SQLJ programs require fewer lines of code than JDBC programs. Hence, they are shorter and easier to debug.
  - SQLJ allows you to imbed host expressions in your SQL statements and takes care of the binding for you. JDBC requires you to write separate method calls to bind each input parameter and retrieve each select list item.
- Security
  - SQLJ performs statement syntax checking, data type checking and authorization checking at compile time while JDBC values are passed to and from SQL without checking at compile time.
  - SQLJ provides for the separation of the package owner and package runner as a result of the bind processing. You can therefore assign different user privileges to each. JDBC only checks user privileges at runtime.
  - JDBC provides finer-grained control over the execution of SQL statements and offers true dynamic SQL capability (e.g. discovery of database or instance metadata at runtime).

You can also combine SQLJ and JDBC in the same stored procedure. Chapter 4 of *DB2 Version 9.1 for z/OS, Application Programming Guide and Reference for Java*, SC18-9842-01, discusses how you do this.

13.6.3 Preparing stored procedures with only JDBC Methods

If your stored procedure uses only JDBC Methods then all you need to do is to compile the stored procedure using the **javac** command. The **javac** command can either be invoked as a foreground command or submitted as a batch job.

Using javac in foreground

Example 13-13 shows the foreground invocation of the **javac** command.

Example 13-13 Using the javac command

```
/SC63/sg247083/spjava:>javac EmpDtlsJ.java
```

Using javac as a batch command

The AOPBATCH utility, provided by Infoprint Server, also runs as z/OS UNIX shell commands or executables. BPXBATCH sends output to the HFS files defined in the JCL STDOUT and STDERR DD statements. AOPBATCH, on the other hand, sends the output to your JES2 output queues directly. Then you can control the output using SDSF. The commands can be directly entered in STDIN. Example 13-14 shows the JCL for compiling the Java program with AOPBATCH.

Example 13-14 Compiling the Java program using AOPBATCH

```
//JAVACOMP JOB (999,POK),'JAVA COMP',CLASS=A,MSGCLASS=A,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
//JOBLIB   DD DSN=CEE.SCEERUN,DISP=SHR
//JCOMP   EXEC PGM=AOPBATCH,PARM='sh -L'
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDIN    DD *
  cd /SC63/sg247083/spjava
  javac EmpDtlsJ.java
/*
```

Tip: The Java Diagnostics Guide recommends a minimum region size of 128 MB. This gives enough storage to accommodate a (default) 64 MB maximum heap size, the 40+ MB for the JIT, and leaves 24 MB for application and system storage requirements.

13.6.4 Preparing SQLJ stored procedures

Figure 13-1 on page 198 shows the preparation process of an SQLJ stored procedure. There are a number of steps involved in preparing an SQLJ stored procedure. You need to translate, compile, customize, and bind your SQLJ stored procedure as part of its preparation process. These steps are now discussed.

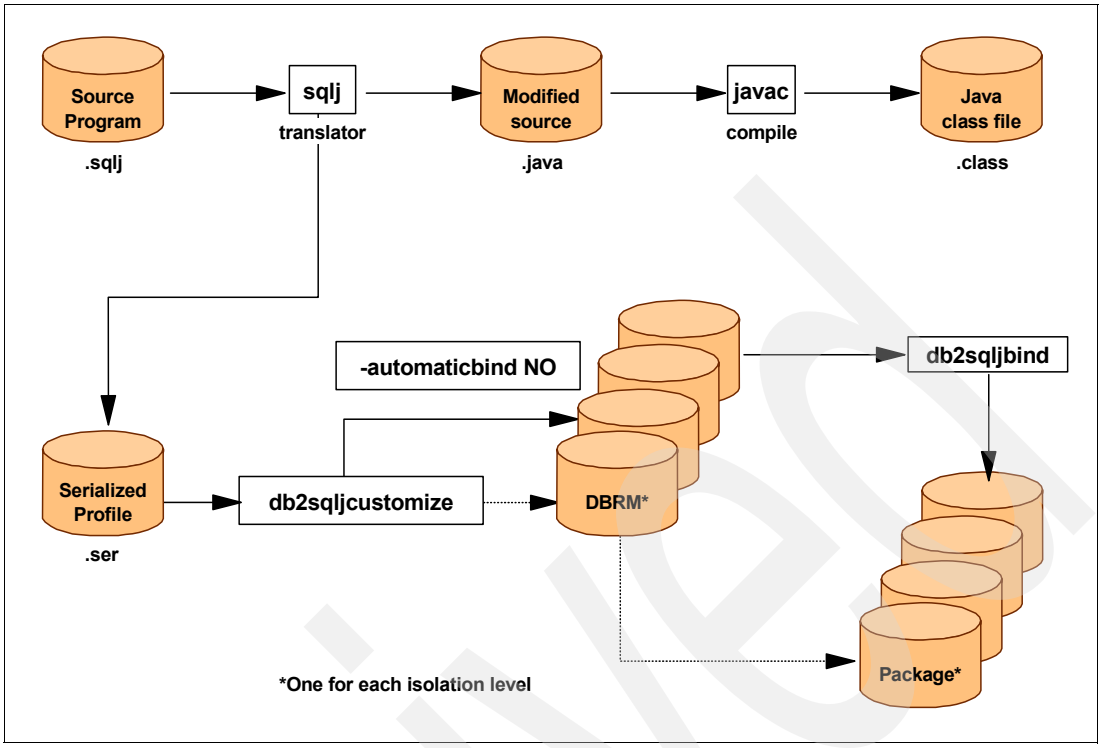

Figure 13-1 SQLJ preparation process

Translation and compilation

All SQLJ stored procedure source files should end with a .sqlj extension. In our example, the stored procedure EmpDtl1J.sqlj resides in /SC63/sg247083/spjava.

Issue the following command to translate and compile the sqlj stored procedure:

```
/SC63/sg247083/spjava:>sqlj -compile=true EmpDtl1J.sqlj
```

You can issue the **sqlj** command from any directory as long as the $PATH variable is defined to include the path where the sqlj executable resides (that is, /usr/lpp/db2/db9a/jcc/lib).

When you issue the **sqlj** command, the translator creates a modified Java source code, EmpDtl1J.java. The compile=true option forces the translator to compile the modified Java code into bytecode and produce corresponding class files.

A number of files are produced as a result of SQLJ program preparation. They are shown in Example 13-15.

Example 13-15 File produced by SQLJ preparation

```
EmpDtl1J.sqlj                    sqlj source code for the stored procedure.

EmpDtl1J.java                    The translator step modifies the sqlj source
                                 code and creates a corresponding Java source
                                 file.

EmpDtl1J.class                   Stored procedure class file produced as a
                                 result of -compile=true option.

EmpDtl1J_Ctx.class               Connection Context class.
                                 The name of the class is the same as that
```

	mentioned in the sqlj source code. The no. of Connection context classes produced depends upon the number of context classes defined in the sqlj source code.
EmpDtl1J_SJProfile0.ser	For every connection context class the translator creates a serialized profile. If the Java routine defines n context classes, the translator produces n serialized profiles. These profiles need to be customized further by using the db2profc command (discussed in the next section).
EmpDtl1J_SJProfileKeys.class	

Customizing the profile

Example 13-16 shows the command used for customizing the server profile. The **db2sqljcustomize** command produces four DBRMs (one for each transaction isolation) by default and also updates the serialized profiles with DB2-specific information obtained from:

- `-url`, specifies the connection URL for the target database server.
- `-user` and `-password`, specifies the login userid and password to this server.
- `-rootpkgname`, specifies the common part of the names of the four DBRMs and/or packages that the SQLJ customizer generates. The package names must be seven or fewer characters in length, unless the `-longpkgname` option is also specified.[3]
- `-qualifier`, the qualifier that is to be used for unqualified objects in the SQLJ program during online checking. This value is not used as the qualifier when the packages are bound.
- -collection, the collection ID.
- -bindoptions, the bind options.

The default behavior of **db2sqljcustomize** when the connection URL is specified is to perform online checking of data types in the SQLJ stored procedure; and to automatically bind the packages created.

If the **db2sqljcustomize** fails, or if you want to create identical packages in another server, you can issue **db2sqljbind** to bind the packages manually.

Example 13-16 Sample db2sqljcustomize command

```
/SC63/sg247083/spjava:>db2sqljcustomize -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A
-user paolor5 -password pup4sale -rootPkgName EMPDTL1 -qualifier DEVL7083 -collection
DEVL7083 -bindoptions "CURRENTDATA NO QUALIFIER DEVL7083" EmpDTL1J_SJProfile0.ser
```

Note: You may opt to create a script file that contains the db2sqljcustomize command because in OMVS, you may not be able to enter all options in one line.

Example 13-17 shows the output of the **db2sqljcustomize** command. Notice the names of the four DBRM members that the **db2sqljcustomize** command produces. For more information about db2sqljcustomize, consult the section “Commands for SQLJ Preparation” in *DB2 for z/OS V9.1, Application and Programming Guide and Reference for Java,* SC18-9842-01.

[3] the -longpkgname option allows you to specify package names of up to 127 characters.

Example 13-17 Output of the db2sqljcustomize command

```
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Begin Customization
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Set qualifier for online checking to SCHEMA: DEVL7083
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Loading profile: EmpDtl1J_SJProfile0
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Customization complete for profile EmpDtl1J_SJProfile0.ser
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Begin Bind
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Loading profile: EmpDtl1J_SJProfile0
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ User bind options: CURRENTDATA NO QUALIFIER DEVL7083
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Driver defaults(user may override): BLOCKING ALL VALIDATE BIND
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Fixed driver options: DATETIME ISO DYNAMICRULES BIND
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL11 at isolation level UR
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL12 at isolation level CS
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL13 at isolation level RS
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL14 at isolation level RR
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Bind complete for EmpDtl1J_SJProfile0
                                                                RUNNING
```

Binding the DBRMs into packages

If the **db2sqljcustomize** fails to bind your packages, or if you wish to bind the packages to another server, then you can do so using the **db2sqljbind** command.

In Example 13-18 we show the command to bind the packages created in DB9A to server DB9B.

Example 13-18 Binding the DBRM packages for SQLJ stored procedure. using db2sqljbind

```
db2sqljbind -url jdbc:db2://wtsc63.itso.ibm.com:12350/DB9B \
   -user paolor5 -password pup4sale \
  -bindoptions "QUALIFIER DEVL7083" \
EmpDtl1J_SJProfile0.ser
```

Example 13-19 shows the output of the **db2sqljbind** command.

Example 13-19 Output of the db2sqljbind command

```
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Begin Bind
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Loading profile: EmpDtl1J_SJProfile0
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ User bind options: QUALIFIER DEVL7083
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Driver defaults(user may override): BLOCKING ALL VALIDATE BIND
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Fixed driver options: DATETIME ISO DYNAMICRULES BIND
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL11 at isolation level UR
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL12 at isolation level CS
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL13 at isolation level RS
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Binding package EMPDTL14 at isolation level RR
Ýibm¨Ýdb2¨Ýjcc¨Ýsqlj¨ Bind complete for EmpDtl1J_SJProfile0
```

Using the batch SQLJ preparation job

The SQLJ preparation steps discussed above can all be done by submitting a batch job as shown in Example 13-20. Make sure that the user who submits the job has the appropriate profile data set as discussed in 13.6.1, “Profile data set” on page 195.

Example 13-20 Sample Job to prepare an SQLJ stored procedure

```
//SQLJCOMP JOB (999,POK),'COBOL C/L/B/E',CLASS=A,MSGCLASS=T,           JOB05064
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=SC63,L=9999
//JOBLIB   DD DSN=CEE.SCEERUN,DISP=SHR
```

```
//JCOMP   EXEC PGM=AOPBATCH,PARM='sh -L'
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDIN    DD *
cd /SC63/sg247083/spjava
db2sqljcustomize -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A \
   -user paolor5 -password pup4sale \
   -rootPkgName EMPDTL1 -qualifier DEVL7083 -collection DEVL7083
   -bindoptions "CURRENTDATA NO QUALIFIER DEVL7083" \
EmpDtl1J_SJProfile0.ser
/*
```

13.7 Making the stored procedure class files available to DB2

When a Java stored procedure is called, DB2 checks the external name for the Java stored procedure and determines whether the Java stored procedure is installed in the DB2 catalog as a jar, or is associated with a class file in the CLASSPATH.

From our additional materials link, download and browse the following files:

```
EmpDtljJ.java
EmpDtlsJ.ddl
EmpDtl1J.sqlj
EmpDtl1J.ddl
```

13.7.1 Without jars

If you are not using jars, you need to ensure that the zFS or HFS directory that contains the stored procedure class files is in the CLASSPATH for the WLM-established stored procedure address space. The CLASSPATH is specified in the JAVAENV data set of your WLM procedure. See 13.3.7, “Setting up the JAVAENV data set for Java stored procedure execution” on page 188 for details on how to set up the CLASSPATH. If you need to modify the CLASSPATH environment variable in the JAVAENV data set to include the directory for the Java routine’s classes, you must restart the WLM address space to make it use the modified CLASSPATH.

If your stored procedure is using JDBC methods only, you only need to place the **.**class files in the CLASSPATH directory. In case of SQLJ stored procedures, you also need to place the **.**ser files and the context classes in the CLASSPATH directory.

Table 13-4 shows the relationship between classpath and the location of the class files. It has two examples, one each for JDBC and SQLJ stored procedures.

Table 13-4 Relation between CLASSPATH and the location of the class files

JDBC Stored Procedure: EMPDTLSJ	
Stored Procedure Source Code	EmpDtlsJ.java
Stored Procedure Class File	EmpDtlsJ.class
CLASSPATH (In the JAVAENV data set)	/SC63/sg247083/spjava

JDBC Stored Procedure: EMPDTLSJ	
LOCATION of the class file.	/SC63/sg247083/spjava/ Notice that the location of the class file is the same as that of the CLASSPATH directory. If there was a package statement “package abc” in the Java source code, then you need to place the class file in a directory known as /SC63/sg247083/spjava/abc/ The CLASSPATH value need not change and it should not include the abc subdirectory.
EXTERNAL NAME in the DDL	‘EmpDtlsJ.GetEmpDtls’
SQLJ Stored Procedure: EMPDTL1J	
Stored Procedure Source Code	EmpDtl1J.sqlj
CLASSPATH (In the JAVAENV data set)	/SC63/sg247083/spjava
LOCATION for the Class files	The class files along with the .ser files should be placed in directory /SC63/sg247083/spjava/. Notice that the location of the class files is the same as that of the CLASSPATH directory. If there was a package statement “package abc” in the Java source code, then you need to place the class file in a directory known as /SC63/sg247083/spjava/abc/ The CLASSPATH would continue to remain the same and it should not include the abc subdirectory.
EXTERNAL NAME in the DDL	‘EmpDtl1J.GetEmpDtls’ Notice that the context classes and serialized profiles are not mentioned in the “external name clause.”.

13.7.2 With jars

You also have an option to create a jar file containing all your class files, and then define the jar file to DB2 using the IBM-supplied stored procedure SQLJ.DB2_INSTALL_JAR. While recommended, you do not have to define the jar file to DB2. You could include the directory of the jar file in the CLASSPATH.

If you define the jar to DB2, you should not have the class files in the CLASSPATH. In case the class files appear in both places, the DB2 catalog and the CLASSPATH directory, you may get unpredictable results.

Example 13-21 shows the commands for creating the jar file for the SQLJ stored procedure. Notice that the jar file contains the classes, context classes, and serialized profiles corresponding to the stored procedure. You create a jar file for a JDBC stored procedure in the same manner, listing the classes, and other jar files if needed.

Example 13-21 Employee.jar containing files for sqlj stored procedure EmpDtl1J

```
/u/paolor7:>cd /SC63/sg247083/spjava
/SC63/sg247083/spjava:>jar -cvf Employee.jar EmpDtl1*.class EmpDtl1*.ser
added manifest
```

```
adding: EmpDtl1J.class(in = 2481) (out= 1358)(deflated 45%)
adding: EmpDtl1J_Ctx.class(in = 2044) (out= 807)(deflated 60%)
adding: EmpDtl1J_SJProfileKeys.class(in = 991) (out= 560)(deflated 43%)
adding: EmpDtl1J_SJProfile0.ser(in = 3309) (out= 1442)(deflated 56%)
/SC63/sg247083/spjava:>
```

13.7.3 Defining jars to DB2

Once you have created the jar file, you can define it to DB2. DB2 provides you with the DB2_INSTALL_JAR stored procedure to load the jar file to DB2. You could invoke the DB2_INSTALL_JAR stored procedure from the IBM Data Studio or from an application program:

```
CALL SQLJ.DB2_INSTALL_JAR(empdtlBlob, DEVL7083.EMPDTL,0)
```

The first argument, known as AJAR, specifies a Large Binary object or BLOB.

The second argument is the SCHEMANAME.JARNAME. The JARNAME that you specify can be anything.

The third argument is always 0.

Using IBM Data Studio to define jars to DB2

- In the Database Explorer, expand your connection to **Schemas** → **SQLJ** → **Stored Procedures.**
- Right-click **DB2_INSTALL_JAR** and click **Run.**
- In the Specify Parameter Values dialog, click the ellipsis for the field AJAR.
- In the Specify Value - AJAR, click **Browse**.
- Point your file browser to the location where your jar is. IBM Data Studio will convert the jar file contents into a BLOB.
- Click **OK** twice to exit both dialogs.

Figure 13-2 on page 204 shows the two dialogs when running SQLJ.DB2_INSTALL_JAR from IBM Data Studio.

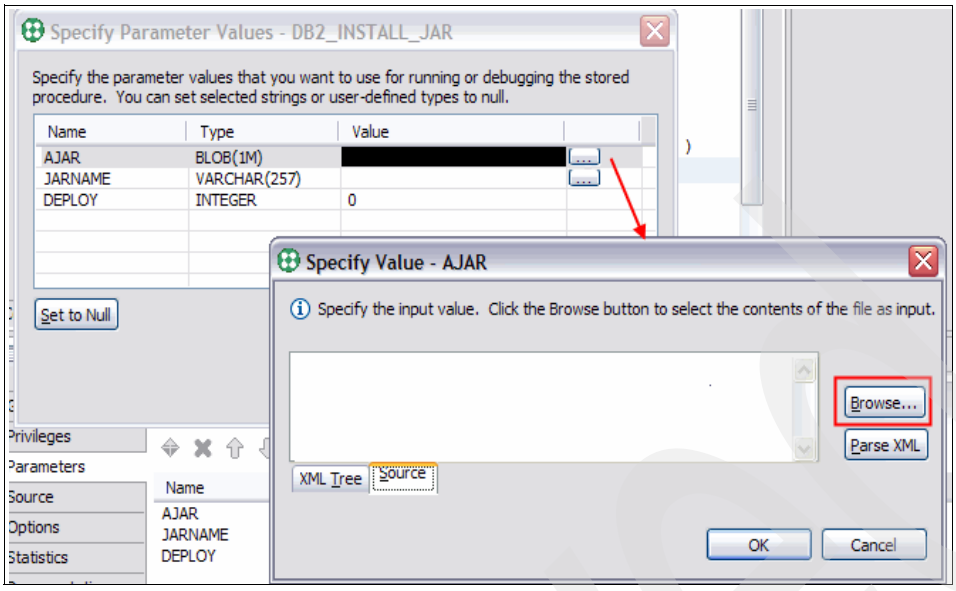

Figure 13-2 Invoking SQLJ.DB2_INSTALL_JAR from IBM Data Studio → Database Explorer

Calling SQLJ.DB2_INSTALL_JAR from an application

Example 13-22 is an excerpt from a sample Java application that defines a jar to DB2 by calling SQLJ.DB2_INSTALL_JAR. The complete code is available in the additional materials provided with the book.

Example 13-22 SimpleInstallJar code (simplified)

```
import java.sql.*;              // JDBC classes
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;

class SimpleInstallJar
{
   public static void main (String argv[])
   {
   // *** change this to your target URL ***
   String url = "jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A";
   //***  change this to the jarname you want defined in DB2 ***
   String jarname = "PAOLOR5.EMPLJAR";
   //***   change this to the Java SP's classname ***
   String classid = "EmpDtlsJ";
   // *** change this ***
   String jarfile = "C:\\SG247083\\SG247083_01-JAVA\\Employee.jar";
   // *** optional: change this ***
   String jarsourcefile = "C:\\SG247083\\SG247083_01-JAVA\\EmpDtlsJ.java";
   try
   {
      Class.forName ("com.ibm.db2.jcc.DB2Driver").newInstance ();
      // *** change this ***
      Connection con = DriverManager.getConnection(url, "PAOLOR5", "PUP4SALE");

      File aFile = new File(jarfile);
      FileInputStream inputStream = new FileInputStream(aFile);
      //System.out.println("Input Stream = " + inputStream);
      CallableStatement stmt;
      String sql = "Call sqlj.db2_install_jar(?, ?, ?) ";
```

```
      stmt = con.prepareCall (sql);
      stmt.setBinaryStream(1,inputStream, (int)aFile.length());
      stmt.setString( 2, jarname );
      stmt.setInt( 3, 0 );
      boolean isrs = stmt.execute();
      stmt.close ();
      System.out.println("install jar completed");

      con.commit();
      con.close ();
   }
   catch (Exception e)
   {
      System.out.println("install jar failed");
      e.printStackTrace ();
   }
}
}
```

If the user who creates the stored procedure is different from the user who defines the jar to DB2, you need to give authority to use the jar:

```
GRANT USAGE ON JAR DEVL7083.EMPDTL TO PUBLIC
```

After defining the jar to DB2, you can create a stored procedure that can reference the jar file as:

```
EXTERNAL NAME 'DEVL7083.EMPDTL:EmpDtl1J.GetEmpDtls'
```

Where:

- DEVL7083 is the schema name.
- EMPDTL is the jar name as defined to DB2 (it is not the jar file!).
- EmpDtl1J is the stored procedure class name.
- GetEmpDtls is the method name.

There is no package name since we did not code package name in the sqlj source code.

13.8 DDL for defining a Java stored procedure

Example 13-23 shows a sample DDL definition used for defining the JDBC-based Java stored procedure.

Example 13-23 Sample DDL for registering the stored procedure EmpDtlsJ

```
CREATE PROCEDURE DEVL7083.EMPDTLSJ
( IN  EMPNO      CHARACTER(6),
  OUT FIRSTNAME   VARCHAR(12),
  OUT MIDINIT        CHAR(1),
  OUT LASTNAME    VARCHAR(15),
  OUT WORKDEPT       CHAR(3),
  OUT SALARY      DECIMAL(9,2),
  OUT HIREDATE       DATE,
  OUT OUTPUTMESSAGE VARCHAR(250))
EXTERNAL NAME 'EmpDtlsJ.GetEmpDtls'
LANGUAGE JAVA
PARAMETER STYLE JAVA
COLLID DSNJDBC
PROGRAM TYPE SUB
```

```
WLM ENVIRONMENT DB9AWLMJ
```

Refer to Table 13-5 for a detailed discussion on the various options when defining a DB2 stored procedure.

Table 13-5 DDL parameters for Java stored procedure definition

Parameter	Description
LANGUAGE	It must always be Java.
PARAMETER STYLE	The only parameter style supported is Java. A discussion about the parameters can be found in 13.8.1, "INPUT/OUTPUT parameters" on page 206.
COLLID	Collection name that has the JDBC packages bound in it. In case of SQLJ stored procedures the collection should also have the stored procedure packages bound in it.
EXTERNAL NAME	Specifies the Java classname.methodname of the Java stored procedure. Refer to 13.8.2, "EXTERNAL NAME" on page 207 for further details.
PROGRAM TYPE	SUB. For Java stored procedures, PROGRAM TYPE should always be SUB. See "Why is the program type SUB for a Java stored procedure".
WLM ENVIRONMENT	Java stored procedures can only run in a WLM Environment. We used a WLM Environment DB9AWLMJ. Steps for defining the environment can be found in Chapter 4, "Setting up and managing Workload Manager" on page 39.

Note: Using JDK 1.4.1 and up JVM requires adding XPLINK(ON) to JAVAENV

Why is the program type SUB for a Java stored procedure

Java stored procedures are considered subroutines or subprograms in DB2, because:

- The parameters are processed as separate arguments and not as an array of arguments.
- The top level method in a Java stored procedure is a `public static` method as opposed to a `public static void main`.
- The stored procedure can be called and return execution to the calling application when the stored procedure processing is done.
- Unlike Java applications, Java stored procedures' JDBC connection or SQLJ connection context can use the connection to the data source that processes the CALL.

13.8.1 INPUT/OUTPUT parameters

A Java routine must be defined with PARAMETER STYLE JAVA. PARAMETER STYLE JAVA specifies that the routine uses a parameter-passing convention that conforms to the Java language and SQLJ specifications. The Java method that is invoked as a stored procedure must be a `public static` method with a `void` return type. Parameters sent to a Java stored procedure are declared using the default JDBC mapping of SQL data types to Java data types. In addition, DB2 passes INOUT and OUT parameters as single-entry arrays. This means that in your Java routine, you must declare OUT or INOUT parameters as arrays of the Java type, so that the Java program can set them for return to the calling application. In

Table 13-6, FIRSTNAME is defined as an output variable, and the Java program declares it as a String[] array.

Table 13-6 Input/output parameter handling in stored procedures

Parameter declaration in stored procedure DDL	Arguments defined in the Java Method
CREATE PROCEDURE DEVL7083.EMPDTLSJ (IN EMPNO CHARACTER(6), OUT FIRSTNAME VARCHAR(12), OUT MIDINIT CHAR(1), OUT LASTNAME VARCHAR(15), OUT WORKDEPT CHAR(3), OUT SALARY DECIMAL(9,2), OUT HIREDATE DATE, OUT OUTPUTMESSAGE VARCHAR(250)) EXTERNAL NAME 'EmpDtlsJ.GetEmpDtls' LANGUAGE JAVA PARAMETER STYLE JAVA COLLID DSNJDBC PROGRAM TYPE SUB WLM ENVIRONMENT DB2GWEJ1	public static void GetEmpDtls(String empno, String[] firstName, String[] midInit, String[] lastName, String[] workDept, java.math.BigDecimal[] salary, java.sql.Date[] hireDate, String[] outputMessage)

Things are a bit different when the stored procedure returns a result set. For each result set, include an object of type java.sql.ResultSet[] in the parameter list for the stored procedure method. Table 13-7 shows a stored procedure returning a result set.

Table 13-7 Stored procedure returning a result set

Parameter declaration in stored procedure DDL	Arguments defined in the Java Method
CREATE PROCEDURE DEVL7083.EMPRSETJ (IN WORKDEPT CHARACTER(3), OUT OUTPUTMESSAGE VARCHAR(250)) EXTERNAL NAME 'EmpRsetJ.GetEmpResult' LANGUAGE JAVA PARAMETER STYLE JAVA COLLID DSNJDBC PROGRAM TYPE SUB **DYNAMIC RESULT SETS 1** WLM ENVIRONMENT DB9AWLMJ	public static void GetEmpResult((String workDept, String[] outputMessage, **ResultSet[] rs**)

13.8.2 EXTERNAL NAME

The external name specifies the location and name of the Java class name and method name that needs to be invoked when the call to a stored procedure is made. You can specify the external name in numerous ways. We examine and discuss the various options and combinations.

Case A: Simple case - no jars and no packages

Java stored procedure EmpDtlsJ.java has been compiled into EmpDtlsJ.class. This stored procedure makes JDBC calls.

There are no package statements specified in the Java code. The name of the static method is GetEmpDtls. The CLASSPATH variable in the JAVAENV data set has been set to /SC63/sg247083/spjava/. The CREATE PROCEDURE DDL contains the clause,

EXTERNAL NAME 'EmpDtlsJ.GetEmpDtls'

In USS, the LOCATION for EmpDtlsJ.class file should be in the same directory as defined by the CLASSPATH, such as /SC63/sg247083/spjava/.

Case B: Dealing with packages

We modify the Java stored procedure EmpDtlsJ.java and added the package statement `com.ibm.itso` in the first line of the Java code. Then we saved this to EmpDtls2.java, and compiled this into EmpDtls2.class. The name of the static method inside the stored procedure is still GetEmpDtls. The CLASSPATH variable in the JAVAENV data set has been set to /SC63/sg247083/spjava/. When we code the CREATE PROCEDURE DDL for this stored procedure, we set

EXTERNAL NAME ‘com.ibm.itso.EmpDtlsJ.GetEmpDtls’

In USS, the LOCATION for EmpDtlsJ.class file should be in a directory named /SC63/sg247083/spjava/com/ibm/itso/. Note that this directory is *not* in the CLASSPATH.

Notice that we need to create the subdirectories com, ibm, and itso and place the stored procedure class file in the lowermost subdirectory in the hierarchy, and the EXTERNAL NAME clause has a mention of the package name.

Case C: External jar files not defined to DB2

We modify Java stored procedure EmpDtlsJ.java and added the package statement `abc/pqr/xyz` in the first line of the Java code. We saved this to EmpDtls3.java and compiled it into EmpDtls3.class. The EmpDtls3.class resides in directory /SC63/abc/pqr/xyz/. The package statement `package abc.pqr.xyz` is specified in the first line of the Java code. The jar file Employee.jar is created to hold the EmpDtlsJ.class by issuing the commands shown in Example 13-24. In the CREATE PROCEDURE DDL, we set

EXTERNAL NAME ‘abc.pqr.xyz.EmpDtlsJ.GetEmpDtls’

The Employee.jar file needs to be added to the CLASSPATH. The CLASSPATH environment variable needs to be set to CLASSPATH=/SC63/Employee.jar.

Notice that the EXTERNAL NAME clause has no mention of the jar file. You need to mention the jar file only if you define the jar to DB2. At runtime, the class file EmpDtlsJ.class is no longer required. Instead, the Employee.jar file is used by the system to pick up the relevant class files.

Example 13-24 Commands to create the Employee.jar file

```
=>cd /SC63
/SC63:>jar -cvf Employee.jar abc
added manifest
adding: abc/(in = 0) (out= 0)(stored 0%)
adding: abc/pqr/(in = 0) (out= 0)(stored 0%)
adding: abc/pqr/xyz/(in = 0) (out= 0)(stored 0%)
adding: abc/pqr/xyz/EmpDtlsJ.class(in = 1720) (out= 953)(deflated 44%)
/SC63:>
```

Case D: Jar files defined to DB2

Java stored procedure EmpDtls3.java has been recompiled into EmpDtlsJ.class, which resides in directory /SC63/abc/pqr/xyz/. The package statement `package abc.pqr.xyz` is specified in the first line of the Java code. The jar file Employee.jar is created to hold EmpDtlsJ.class by issuing the commands shown in Example 13-24. We registered the Employee.jar file to DB2 using the IBM-supplied stored procedure SQLJ.DB2_INSTALL_JAR. 13.7.3, “Defining jars to DB2” on page 203 discussed the two ways to define the jars to DB2.

Once the jar file is registered to DB2, you must remove the jar file from the CLASSPATH Environment variable. Failure to do so may cause unpredictable errors.

In the CREATE PROCEDURE DDL, the external name clause should be defined as follows:

```
EXTERNAL NAME 'DEVL7083.EMPLJAR:abc.pqr.xyz.EmpDtlsJ.GetEmpDtls'
```

Where:

- DEVL7083 is the jar schema name.
- EMPLJAR is the jar name as defined to DB2 (it is not the jar file!).
- abc.pqr.xyz is the package name.
- EmpDtlsJ is the stored procedure class name.
- GetEmpDtls is the method name.

The CLASSPATH variable can be set to blank.

13.9 Debugging JDBC and SQLJ

With DB2 9 for z/OS and DB2 V8 with APAR PK41138 / PTF UK25860 applied, Java stored procedures, with can be debugged using the IBM Data Studio Unified Debugger feature. Debugging using Data Studio is discussed in Chapter 28, “Tools for debugging DB2 stored procedures” on page 735.

Another way to debug a Java stored procedure is to convert it into a Java application and then use an Integrated Development Environment such as Eclipse to debug the Java application.

The steps of converting a Java stored procedure to a Java application with minimal effort are documented in 13.9.1, “Changing Java stored procedure to enable debugging in Eclipse” on page 209. This methodology leverages the Java Debugger within Eclipse. The Eclipse Debug Perspective is the same perspective used by Data Studio's Unified Debugger. See 28.3.4, “Defining the EMPDTLSS SQL case study for debugging” on page 749 for details about the Debug Perspective.

13.9.1 Changing Java stored procedure to enable debugging in Eclipse

Table 13-8 on page 210 shows the conversion.

Table 13-8 Converting the stored procedure method to a main method

Stored procedure code	Converted Java application
Changes to the Stored Procedure Method	

public static void GetEmpDtls(
String empno,
String[] firstName,
String[] midInit,
String[] lastName,
String[] workDept,
java.math.BigDecima[] salary,
String[] outputMessage)
	public static void main (String args[])
	{
	String empno;
	empno=args[0];
	String[] firstName = new String[1];
	String[] midInit = new String[1];
	String[] lastName = new String[1];
	String[] workDept = new String[1];
	java.math.BigDecimal[] salary = new java.math.BigDecimal[1];
	String[] outputMessage = new String[1];
	
	Points to note:
	
	All the output parameters are defined as an array of one element.
	
	GetEmpDtls Method is changed to aMain Method.
	
	Input parameters need not be defined as arrays, the input variables need to be populated by values passed by the command line argument.
	
	You can debug the Java application from the RAD or any Java Development tool.
	The Java application can be invoked from the command line:
	
	Java EmpDtlsJ '000010'

Changes to the connection String	
Connection conndb2 = null; conndb2 = DriverManager.getConnection("jdbc:default:connection");	The connection statements in the stored procedure need to be changed, depending on where the Java application needs to run. In either case it accesses data from DB2 for z/OS: Java application running on Host Connection conndb2 = null; Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver"); conndb2 = DriverManager.getConnection("jdbc:db2os390sqlj:DB2G"); Java application running on Windows and accessing data on Z/OS. Notice that you need to mention the userid and password on the connection string. Connection conndb2 = null; Class.forName("COM.ibm.db2.jdbc.app.DB2Driver"); conndb2 = DriverManager.getConnection("jdbc:db2:DB2G","userid","password");

Once the above changes are made to your application, the Java application code needs to be copied into a Java project in Eclipse. You can obtain a free copy of Eclipse from the web site:

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/galileo/SR1/eclipse-java-galileo-SR1-win32.zip

The site includes several versions of Eclipse. The version that the current release of Data Studio uses is Eclipse 3.4.

Once you've downloaded Eclipse, you can launch the Eclipse Help Contents, which will walk you through:

- Opening the Java Perspective,
- Creating a Java Project,
- Importing your converted Java stored procedure into the Java project,
- Creating a Java application runtime configuration
- Debugging the Java application

You can code System.out.println and System.err.println lines in your Java code. The output is directed to JAVAOUT and JAVAERR DD cards in the WLM address space.

Example 13-25 shows the DD cards that you need to include in your WLM address space.

Example 13-25 DD cards for Java in WLM procedure

```
//JAVAOUT  DD    PATH='/SC63/sg247083/JAVAOUT.TXT',
// PATHOPTS=(ORDWR,OCREAT,OAPPEND),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
//JAVAERR  DD    PATH='/SC63/sg247083/JAVAERR.TXT',
// PATHOPTS=(ORDWR,OCREAT,OAPPEND),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
```

JAVAOUT maps to STDOUT and JAVAERR maps to STDERR. This example uses the data sets in an append fashion. This data set should be deleted occasionally to keep it from growing without bounds.

13.10 Java sample JDBC stored procedure

In this section we show how to implement JDBC routines.

13.10.1 Sample Java stored procedure code: EmpDtlsJ using JDBC

The sample Java stored procedure shown in Example 13-26 illustrates a JDBC Java stored procedure:

1. All the out parameters need to be defined as arrays.
2. The connection string in a stored procedure should always have:

   `"jdbc:default:connection". The stored procedure should always use an existing connection.`
3. While passing the parameters back to the caller, you need to populate the first element of the array. That is:

   ```
   hireDate[0]  = rs.getDate("HIREDATE");
   ```
4. The Java statements (SQL + language code) in the stored procedure should be included in the try block. The catch block should be coded to handle any SQL exceptions or any Java exceptions.

Example 13-26 EmpDtlsJ - Using JDBC

```
import java.sql.*;
import java.io.*;
import java.math.*;

public class EmpDtlsJ {

public static void GetEmpDtls(
       String  empno,
       String[] firstName,
       String[] midInit,
       String[] lastName,
       String[] workDept,
       java.math.BigDecimal[] salary,
       java.sql.Date[] hireDate,
       String[] outputMessage)                                              1
  {
       Connection conndb2 = null;
       int rc ;
       String sql = " ";
       outputMessage[0] = " ";
        try {
            // Use an existing connection to DB2

            conndb2 = DriverManager.getConnection("jdbc:default:connection");  2
            Statement stmtdb2 = conndb2.createStatement();

            sql = "SELECT * FROM DEVL7083.EMP "
                + " WHERE EMPNO = '"+ empno + "'";
            ResultSet rs =  stmtdb2.executeQuery(sql) ;
            if (rs.next())
              {
               empno     = rs.getString("EMPNO");
               firstName[0] = rs.getString("FIRSTNME");
               midInit[0]   = rs.getString("MIDINIT");
               lastName[0]  = rs.getString("LASTNAME");
               workDept[0]  = rs.getString("WORKDEPT");
               salary[0]    = rs.getBigDecimal("SALARY");
               hireDate[0]  = rs.getDate("HIREDATE");                       3
              }
        }
    catch (SQLException e)                                                  4
    {
           outputMessage[0]  = "SQLException raised, SQLState = "
           + e.getSQLState() + " SQLCODE = " + e.getErrorCode()
           + " :" + e.getMessage();
    }
        catch (Exception e) {
            outputMessage[0] = e.toString();
        }
    }
}
```

13.10.2 DDL for Java stored procedure EmpDtlsJ

Example 13-27 shows the DDL for creating the EMPDTLSJ stored procedure.

Example 13-27 DDL for EMPDTLSJ

```
CREATE PROCEDURE DEVL7083.EMPDTLSJ
 ( IN  EMPNO      CHARACTER(6),
```

```
OUT FIRSTNAME   VARCHAR(12),
OUT MIDINIT        CHAR(1),
OUT LASTNAME    VARCHAR(15),
OUT WORKDEPT       CHAR(3),
OUT SALARY      DECIMAL(9,2),
OUT HIREDATE    DATE,
OUT OUTPUTMESSAGE VARCHAR(250))
 EXTERNAL NAME 'EmpDtlsJ.GetEmpDtls'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  COLLID DSNJDBC
  PROGRAM TYPE SUB
  WLM ENVIRONMENT DB2GWEJ1
```

13.10.3 Deploying JDBC stored procedures on z/OS

The way you deploy your Java stored procedures depends upon where you developed them. If you use a Java development environment on a workstation, you have to transfer the application to the zSeries® machine first. We used the File Transfer Protocol (FTP) statements shown in Example 13-28.

Example 13-28 FTP the Java source code

```
R:\SG24-7604\addmat\SG247083_01-JAVA>ftp wtsc63.itso.ibm.com
Connected to wtsc63.itso.ibm.com.
220-FTP Server (user 'marichu@us.ibm.com')
220
User (wtsc63.itso.ibm.com:(none)): paolor5
331-Password:
331
Password:
230-220-FTPMVS1 IBM FTP CS V1R9 at wtsc63.itso.ibm.com, 23:52:03 on 2007-11-02.
230-PAOLOR5 is logged on.  Working directory is "PAOLOR5.".
230
ftp> cd /SC63/sg247083/spjava
250 HFS directory /SC63/sg247083/spjava is the current working directory.
ftp> bin
ftp> put EmpDtlsJ.java
200 Port request OK.
125 Storing data set /SC63/sg247083/spjava/EmpDtlsJ.java
250 Transfer completed successfully.
ftp: 1615 bytes sent in 0.00Seconds 1615000.00Kbytes/sec.
ftp>
```

13.10.4 Sample Java stored procedure returning a result set - EmpRsetJ

Example 13-29 shows the DDL for Java stored procedure EmpRsetJ. Notice that the definition of the stored procedure mentions a number of Dynamic Result Sets that the stored procedure returns.

Example 13-29 DDL for Java stored procedure EmpRsetJ

```
CREATE PROCEDURE DEVL7083.EMPRSETJ ( IN  WORKDEPT CHARACTER(3),
OUT OUTPUTMESSAGE VARCHAR(250))
EXTERNAL NAME 'EmpRsetJ.GetEmpResult'
LANGUAGE JAVA
PARAMETER STYLE JAVA
COLLID DSNJDBC
```

```
PROGRAM TYPE SUB
DYNAMIC RESULT SETS 1
WLM ENVIRONMENT DB9AWLMJ
```

Example 13-30 shows the code for Java stored procedure EmpRsetJ. Notice that a result set argument is included in the method signature for GetEmpResult.

Example 13-30 Sample code for Java stored procedure EmpRsetJ

```
import java.sql.*;
import java.math.*;

public class EmpRsetJ {

public static void GetEmpResult( String workDept,String[] outputMessage,ResultSet[] rs)
       {
         Connection conndb2 = null;
         String sql = " ";
         outputMessage[0] = " ";
         Statement stmtdb2 = null;
         try {

             conndb2 = DriverManager.getConnection("jdbc:default:connection");
             sql = "SELECT * FROM DSN8710.EMP "
                   + " WHERE WORKDEPT     = "
                   + "'" + workDept + "'" ;

              stmtdb2 = conndb2.createStatement();
               rs[0] =  stmtdb2.executeQuery(sql) ;
       }
    catch (SQLException e)
    {
           outputMessage[0]  = "SQLException raised, SQLState = "
           + e.getSQLState() + " SQLCODE = " + e.getErrorCode()
           + " :" + e.getMessage();
    }
        catch (Exception e) {
            outputMessage[0] = e.toString();
        }
```

13.10.5 Calling the Java stored procedure

In the Additional Materials for this book we provide a Java application, CALDTLSJ.java, that calls our sample JDBC stored procedure. For your convenience, the code for this application is shown in Example 13-31.

Example 13-31 Sample Java application that calls DB2

```
import java.sql.*;
import java.math.*;

public class CalDtlsJ {

public static void main(String[] args) {
   CallableStatement cstmt = null;
   String url = "jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A";

try {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
```

```
            Connection conndb2 =
               DriverManager.getConnection(
               url,
               "PAOLOR5",
               "PUP4SALE");
            cstmt =
               conndb2.prepareCall("CALL DEVL7083.EMPDTLSJ(?,?,?,?,?,?,?,?)");
            cstmt.setString(1, args[0]);
            cstmt.registerOutParameter(2, Types.VARCHAR);
            cstmt.registerOutParameter(3, Types.CHAR);
            cstmt.registerOutParameter(4, Types.VARCHAR);
            cstmt.registerOutParameter(5, Types.CHAR);
            cstmt.registerOutParameter(6, Types.DECIMAL);
            cstmt.registerOutParameter(7, Types.DATE);
            cstmt.registerOutParameter(8, Types.VARCHAR);

            cstmt.execute();

            System.out.println("First Name " + cstmt.getString(2));
            System.out.println("Mid   Name " + cstmt.getString(3));
            System.out.println("Last  Name " + cstmt.getString(4));
            System.out.println("Work Dept " + cstmt.getString(5));
            System.out.println("Salary " + cstmt.getBigDecimal(6));
            System.out.println("Hire Date" + cstmt.getDate(7));
            System.out.println("OutMessgae " + cstmt.getString(8));

            cstmt.close();
            conndb2.commit();
            conndb2.close();
            System.out.println("I am done");
         } catch (SQLException e) {
            System.out.println(
               "SQLException raised, SQLState = "
               + e.getSQLState()
               + " SQLCODE = "
               + e.getErrorCode()
               + " :"
               + e.getMessage());
         } catch (Exception e) {
            System.out.println("Error" + e.toString());
         }
      }
      }
```

13.11 Java sample SQLJ stored procedure

In this section we show how to implement SQLJ routines.

13.11.1 Sample code for SQLJ stored procedure - EmpDtl1J.sqlj

Example 13-34 shows an SQLJ stored procedure, EmpDtl1J.sqlj. We now discuss various aspects of the code.

Establishing a connection

There are many ways by which an SQLJ program or a stored procedure can connect to a data source. You can follow the steps described here to connect to a data source from an SQLJ stored procedure:

1. Execute an SQLJ connection declaration clause:

   ```
   #sql context EmpDtl1J_Ctx;
   ```

   An SQLJ program requires a connection context class to be defined. In our example we declare the context class EmpDtl1J_Ctx. At the time of compilation the sqlj translator creates a new Java class, EmpDtl1J_Ctx.class.

2. Invoke the JDBC DriverManager.getConnection method:

   ```
   conndb2 = DriverManager.getConnection("jdbc:default:connection")
   ```

3. Invoke the constructor for the connection context class that you created in step 1.

   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. In our example we create the context object myConCtx. For every SQL statement that you execute in your SQLJ program, you need to prefix the statement with the context object:

   ```
   EmpDtl1J_Ctx myConCtx  = null;
   myConCtx = new EmpDtl1J_Ctx(conndb2);
   ```

Host variables

In SQLJ stored procedures or applications, you need to use host variables just as in any other 3-GL language such as COBOL, C, etc. You need to declare your host variables before you can use them in an SQL statement. Example 13-32 shows the host variable declarations that were used in the example.

Example 13-32 Host variable declarations

```
String hfirstName;
String hmidInit;
String hlastName;
String hworkDept;
java.math.BigDecimal hsalary;
```

You can directly select the contents of a DB2 column into a host variable. Example 13-33 shows a sample SQL statement that was used.

Example 13-33 SQL statement with host variables

```
#sql [myConCtx] { SELECT FIRSTNME,MIDINIT,LASTNAME,
                         WORKDEPT,SALARY
                         INTO :hfirstName,:hmidInit,:hlastName,
                              :hworkDept,:hsalary
                         FROM DEVL7083.EMP
                         WHERE EMPNO = :empno };
```

Example 13-34 shows the sample code for an SQLJ stored procedure.

Example 13-34 EmpDtl1J.sqlj

```
import java.sql.*;
import java.math.*;
import sqlj.runtime.*;
#sql context EmpDtl1J_Ctx ;
public class EmpDtl1J {
```

```
public static void GetEmpDtls(
       String  empno,
       String[] firstName,
       String[] midInit,
       String[] lastName,
       String[] workDept,
       java.math.BigDecimal[] salary,
       String[] outputMessage)
{
       String hfirstName;
       String hmidInit;
       String hlastName;
       String hworkDept;
       java.math.BigDecimal hsalary;
       Connection conndb2 = null;
       outputMessage[0] = " ";
       EmpDtl1J_Ctx myConCtx  = null;
        try {
            // Use an existing connection to DB2

            conndb2 = DriverManager.getConnection("jdbc:default:connection");
            myConCtx = new EmpDtl1J_Ctx(conndb2);

            #sql [myConCtx] { SELECT FIRSTNME,MIDINIT,LASTNAME,
                              WORKDEPT,SALARY
                              INTO :hfirstName,:hmidInit,:hlastName,
                                   :hworkDept,:hsalary
                              FROM DEVL7083.EMP
                              WHERE EMPNO = :empno };
             firstName[0] = hfirstName ;
             midInit[0]   = hmidInit ;
             lastName[0]  = hlastName;
             workDept[0]  = hworkDept;
             salary[0]    = hsalary ;
           }
    catch (SQLException e)
           {
            outputMessage[0]  = "SQLException raised, SQLState = "
            + e.getSQLState() + " SQLCODE = " + e.getErrorCode()
            + " :" + e.getMessage();
           }
        catch (Exception e) {
            outputMessage[0] = e.toString();

           }
    }
}
```

13.11.2 Result sets and position updates in SQLJ stored procedures

An equivalent of a cursor in an SQLJ application or a stored procedure is a result set iterator. Like a cursor, a result set iterator can be non-scrollable or scrollable. It is a Java object that you use to retrieve rows from a result table.

There are two types of iterators: positioned iterators and named iterators. Positioned iterators identify the columns of a result table by their position in the result table. Named iterators identify the columns of the result table by table column names.

Apart from retrieving rows from a result set, an iterator can also be used to perform a positioned update. As in DB2 applications in other languages, performing positioned UPDATEs and DELETEs is an extension of retrieving rows from a result table.

Example 13-36 shows a stored procedure that does a positioned update. The stored procedure receives two input parameters: Department Code and Salary-Increase-Factor. We open cursor and fetch records belonging to a specified department. While fetching each record we update the salary of the employee by the given factor. In our example we use a positioned iterator.

The basic steps in using a result set iterator are:

1. Declare the iterator, which results in an iterator class

   Declaring an iterator is very similar to declaring a cursor in other languages. In case you plan to use an iterator for updating data, you need to declare the iterator in a separate file. In case you use an iterator for only selecting data, then you need not declare the iterator in a separate file. Example 13-35 shows the external file that contains the iterator declaration, EmpTst2J_UpdByPos. The iterator specifies that you intend to update the SALARY column.

   Example 13-35 EmpRst2J_UpdByPos.sqlj file - external file declaration

```
import java.math.*;
#sql public iterator EmpRst2J_UpdByPos implements sqlj.runtime.ForUpdate
with(updateColumns="SALARY")  (String ,BigDecimal );
```

2. Define an instance of the iterator class:

```
EmpRst2J_UpdByPos upditer;
```

3. Assign the result table of a SELECT to an instance of the iterator. Notice that you do not mention the `FOR UPDATE CLAUSE in the SQL statement`. The update clause appears in the iterator declaration file:

```
#sql [myConCtx] upditer  = { SELECT EMPNO,SALARY
FROM DEVL7083.EMP WHERE WORKDEPT = :workDept } ;
```

4. Retrieve rows:

   a. Execute a FETCH statement in an executable clause to obtain the current row:

```
#sql {FETCH :upditer INTO :hempno ,:hsalary};
```

   b. Test whether the iterator is pointing to a row of the result table by invoking the PositionedIterator.endFetch method:

```
while (!upditer.endFetch())
```

   c. If the iterator is pointing to a row of the result table, execute an SQL UPDATE...WHERE CURRENT OF :iterator-object statement in an executable clause to update the columns in the current row. Execute an SQL DELETE... WHERE CURRENT OF :iterator-object statement in an executable clause to delete the current row:

```
#sql [myConCtx] upditer  = { SELECT EMPNO,SALARY
FROM DEVL7083.EMP WHERE WORKDEPT = :workDept } ;
```

5. Close the iterator:

```
upditer.close();
```

The stored procedure sample code in Example 13-36 illustrates the use of a positioned update.

Example 13-36 EmpRst2J.sqlj - Sample stored procedure - updating using positioned iterator

```
                                        //this stored proc takes two input parameters
                                        //workdept and salary-increase-factor and
                                        //updates the salary of all employees
                                        //belonging to the dept  by using a positioned
                                        //iterator.
import java.sql.*;
import java.math.*;
import sqlj.runtime.*;
import EmpRst2J_UpdByPos;               //Import the generated iterator class that was
                                        //created by the iterator declaration clause
                                        //Make make sure that the external file
EmpRst2J_UpdByPos.sqlj                  //is defined and translated using the sqlj
                                        //translator.
#sql context EmpRst2Ctx;                //Create the connection context
public class EmpRst2J {

public static void GetEmpResult( String workDept,BigDecimal factor , String[]
outputMessage)
                                      //input parameters are  workdepartment and
                                      //salary-increase-factor
       {
         Connection conndb2 = null;
         outputMessage[0] = " ";
         EmpRst2Ctx  myConCtx = null;
         EmpRst2J_UpdByPos upditer;
         String hempno = " ";
         BigDecimal hsalary = null;
         try {
            conndb2 = DriverManager.getConnection("jdbc:default:connection");
            conndb2.setAutoCommit(false);                    //turn autocommit off
            myConCtx = new EmpRst2Ctx(conndb2);

            #sql [myConCtx] upditer  = { SELECT EMPNO,SALARY
                                       FROM DEVL7083.EMP WHERE WORKDEPT = :workDept } ;
            #sql {FETCH :upditer INTO :hempno ,:hsalary};     //fetch the first recd.
            while (!upditer.endFetch())                       //look for eof cursor
                                                              //condition.
            {
              #sql [myConCtx] {UPDATE DEVL7083.EMP SET SALARY = SALARY * :factor
                               WHERE CURRENT OF :upditer};    //positioned update is
                                                    //achieved by using the upditer
              #sql {FETCH :upditer INTO :hempno ,:hsalary};    //fetch next recd
            }
            upditer.close();                                  //close the update iterator
         }
    catch (SQLException e)


    {
           outputMessage[0]  = "SQLException raised, SQLState = "
           + e.getSQLState() + " SQLCODE = " + e.getErrorCode()
           + " :" + e.getMessage();
    }
        catch (Exception e) {
            outputMessage[0] = e.toString();
        }
    }
}
```

Preparation JCL

Example 13-37 shows the JCL used for preparing the SQLJ stored procedure. Notice that we also need to translate and compile the iterator declaration file, EmpRst2J_UpdByPos.sqlj.

Example 13-37 Sample JCL for preparing the application

```
//SQLJCOMP JOB (999,POK),'COBOL C/L/B/E',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=SC63,L=9999
//JOBLIB   DD DSN=CEE.SCEERUN,DISP=SHR
//JCOMP   EXEC PGM=AOPBATCH,PARM='sh -L'
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDIN    DD *
   cd /SC63/sg247083/spjava
   sqlj EmpRst2J_UpdByPos.sqlj
   sqlj EmpRst2J.sqlj
db2sqljcustomize -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A \
   -user paolor5 -password pup4sale \
   -rootPkgName EMPRST2 -qualifier DEVL7083 -collection DEVL7083 \
   -bindoptions "CURRENTDATA NO QUALIFIER DEVL7083" \
EmpRst2J_SJProfile0.ser
//*
```

DDL definition for EMPRST2J

Example 13-38 shows the DDL used for the definition of the stored procedure.

Example 13-38 DDL definition for the stored procedure

```
CREATE PROCEDURE DEVL7083.EMPRST2J ( IN  WORKDEPT   CHARACTER(3),
                                     IN  FACTOR     DECIMAL(3,2),
                                    OUT OUTPUTMESSAGE VARCHAR(250))
    EXTERNAL NAME 'EmpRst2J.GetEmpResult'
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    COLLID DEVL7083
    PROGRAM TYPE SUB
    COMMIT ON RETURN YES
    WLM ENVIRONMENT DB2GWEJ1
```

13.12 Migrating stored procedures to use the new JCC driver

Assuming that you currently have Java stored procedures running on your DB2 V8 system using the JDBC/SQLJ Legacy Driver, and you need to migrate them to use the new JCC driver, you need to follow the steps[4] described below.

13.12.1 Migrating JDBC stored procedures

The steps are:

[4] The same procedure can be used if you were running your stored procedures in DB2 V7, and wish to migrate to JCC driver in DB2 V7.

1. Create a new WLM application environment for the JCC environment. The JAVAENV data set should have the JCC_HOME variable pointing to the JCC directory. You should not have any reference to the DB2_HOME directory. Refer to 13.3.7, “Setting up the JAVAENV data set for Java stored procedure execution” on page 188 for further details on setting up the WLM procedure for JCC support.
2. It is advisable to create a separate directory to keep your stored procedure class files that will run under the new JCC driver.

   For stored procedures making JDBC calls, the migration is simple. You need to alter the stored procedure to the new WLM environment and copy the stored procedure class files to the new CLASSPATH directory (as specified in the JAVAENV of the new JCC WLM address space). Our existing stored procedures running the Legacy Driver were residing in the directory /SC63/sg247083/DB8AU/spjava; we created a new directory named /SC63/sg247083/DB8AU/spjava/jcc. Table 13-9 shows the various setup steps that you need to perform to move to the JCC driver.

Table 13-9 DB2 V8 migrating JDBC stored procedure from Legacy Driver to JCC

	DB2 V8 using Legacy Driver	**DB2 V8 using Universal Driver (JCC)**
JAVAENV Contents	CLASSPATH=/SC63/sg247083/DB8AU/spjava DB2_HOME=/usr/lpp/db2/db8a JAVA_HOME=/usr/lpp/java/IBM/J1.4 The above names could be different at each site, depending upon where the HFS libraries for Java and DB2 are loaded by the system programer. Typically, the Java libraries are in: /usr/lpp/java/IBM/J1.4 and the db2 libraries in /usr/lpp/db2/db2810. All the stored procedure class files that are developed by programers are kept in the CLASSPATH Library: /SC63/sg247083/DB8AU/spjava	CLASSPATH=/SC63/sg247083/DB8AU/jcc/spjava JCC_HOME=/usr/lpp/db2/db8a/jcc JAVA_HOME=/usr/lpp/java/IBM/J1.4 Notice that the DB2_HOME does not appear, instead the JCC_HOME is mentioned. The JCC_HOME has a jcc subdirectory. Typically, the Java libraries are in: /usr/lpp/java/IBM/J1.4 and the db2 libraries in /usr/lpp/db2/jcc/db2810/jcc. All the stored procedure class files need to be copied to the new CLASSPATH directory: /SC63/sg247083/DB8AU/jcc/spjava
WLMENV	DB8ADJ1	DB8ADJC2
DDL	CREATEPROCEDUREDEVL7083.EMPDTLSJ (IN EMPNO CHARACTER(6), OUT FIRSTNAME VARCHAR(12), OUT MIDINIT CHAR(1), OUT LASTNAME VARCHAR(15), OUT WORKDEPT CHAR(3), OUT SALARY DECIMAL(9,2), OUT OUTPUTMESSAGE VARCHAR(250)) EXTERNAL NAME 'EmpDtlsJ.GetEmpDtls' LANGUAGE JAVA PARAMETER STYLE JAVA COLLID DSNJDBC PROGRAM TYPE SUB WLM ENVIRONMENT DB8ADJ1 WLM environment is set to DB8ADJ1 COLLID DSNJDBC has the 4 JDBC packages bound in it - Refer to JCL DSNTJJCL	In order to use the driver alter the stored procedure to the new WLM ENVIRONMENT. ALTER PROCEDURE DEVL7083.EMPDTLSJ WLM ENVIRONMENT DB8ADJC2; After the ALTER copy the class file EmpDtlsJ.class to the new directory: /SC63/sg247083/jcc/spjava COLLID is no longer DSNJDBC, probably NULLID or what was defined at BIND.

	DB2 V8 using Legacy Driver	DB2 V8 using Universal Driver (JCC)
Profile	Depending on what environment you are running, JCC or Legacy Driver, you need to have an appropriate profile. A profile comes into play whenever you prepare a Java stored procedure or application. When you run any Java applications in USS, the profile is used to set the appropriate environment. For Java stored procedures, the runtime environment is not controlled by the contents of the profile data set, instead it is controlled by the contents of JAVAENV. However, when you prepare a Java stored procedure, the profile contents are used.	
Sample Profile	JAVA_HOME=/usr/lpp/java/IBM/J1.4 PATH=/usr/lpp/java/IBM/J1.4/bin:$PATH PATH=/usr/lpp/db2/db8a/bin:$PATH export PATH LIBPATH=/usr/lpp/db2/db8a/lib:$LIBPATH export LIBPATH LD_LIBRARY_PATH=/usr/lpp/db2/db8a/lib export LD_LIBPATH_PATH CLASSPATH=/usr/lpp/db2/db8a/classes/db2j2classes.zip:. export CLASSPATH STEPLIB=DB8A8.SDSNEXIT:DB8A8.SDSNLOAD:DB8A8.SDSNLOD2:$STEPLIB STEPLIB=CEE.SCEERUN:$STEPLIB export STEPLIB export DB2SQLJPROPERTIES=/SC63/sg247083/DB8AU/db2sqljjdbc.properties Notice that there is no mention of the jcc directories.	JAVA_HOME=/usr/lpp/java/IBM/J1.4 PATH=/usr/lpp/java/IBM/J1.4/bin:$PATH PATH=/usr/lpp/db2/db8a/jcc/bin:$PATH export PATH LIBPATH=/usr/lpp/db2/db8a/jcc/lib:$LIBPATH export LIBPATH LD_LIBRARY_PATH=/usr/lpp/db2/db8a/jcc/lib export LD_LIBPATH_PATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc.jar:$CLASSPATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc_javax.jar:$CLASSPATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/sqlj.zip:$CLASSPATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc_license_cisuz.jar:$CLASSP export CLASSPATH STEPLIB=DB8A8.SDSNEXIT:DB8A8.SDSNLOAD:DB8A8.SDSNLOD2:$STEPLIB STEPLIB=CEE.SCEERUN:$STEPLIB export STEPLIB

13.12.2 Migrating SQLJ stored procedures

Migrating sqlj stored procedures to use the new JCC driver is a bit more tricky. When you compile and prepare your sqlj stored procedures (even an sqlj application) you get a set of executables: A class file, a set of context files, customized serialized profiles (.ser files), and finally a set of packages. In case you plan to run the sqlj stored procedure in a new JCC environment, you need to upgrade only the serialized profiles .ser files. The rest of the executables remain unchanged. IBM provides you with a utility to upgrade the serialized profile.

1. Create a new WLM application environment for the JCC environment. The JAVAENV data set should have the JCC_HOME variable pointing to the JCC directory. You should not have any reference to the DB2_HOME directory. Refer to 13.3.6, “Setting up the WLM procedure” on page 186, for further details on setting up the WLM procedure for JCC support.
2. Copy all the executable files (class files and .ser files) to the new CLASSPATH directory as specified in the JAVAENV data set.
3. Run the db2sqljupgrade utility to upgrade the serialized profiles (.ser). The upgrade utility takes the .ser files created by the Legacy Driver and updates them for the JCC environment. It leaves the DB2 packages and the rest of the class files completely untouched.

   Example 13-39 shows the JCL required to upgrade the serialized profile:

   1,2,3 - We set the CLASSPATH variable to the required JCC libraries.

4 - We need to assign the classes of the Legacy Driver to the CLASSPATH. The JCC driver classes should always be ahead of the classes belonging to the Legacy Driver.

5,6 - Set the PATH variable so that we could execute the db2sqljupgrade utility.

7 - Change to the directory that has the serialized profile.

8 - Command to run the db2sqljupgrade utility. Notice that the upgrade utility only upgrades the .ser files. It does not touch any other class files or DB2 packages.

4. If you do not upgrade the .ser file and try to execute the sqlj stored procedure in a JCC environment, your application will fail with the message shown in Example 13-41 on page 225.

Example 13-39 db2sqljupgrade utility

```
//JAVACOMP JOB (999,POK),'JAVA COMP',CLASS=A,MSGCLASS=A,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
//JOBLIB   DD DSN=CEE.SCEERUN,DISP=SHR
//JCOMP   EXEC PGM=AOPBATCH,PARM='sh -L'
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDIN    DD *
CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc.jar:$CLASSPATH..................... 1
CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/sqlj.zip:$CLASSPATH.....................2
CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc_license_cisuz.jar:$CLASSPATH.....3
CLASSPATH=$CLASSPATH:/usr/lpp/db2/db8a/classes/db2j2classes.zip ..................4
  PATH=/usr/lpp/java/IBM/J1.4/bin.................................................5
  PATH=/usr/lpp/db2/db8a/jcc/bin:$PATH  ..........................................6
  cd /SC63/sg247083/DB8AU/jcc/spjava/.............................................7
  db2sqljupgrade EmpDtl1J_SJProfile0.ser..........................................8
/*
```

Example 13-40 shows the output listing of the upgrade utility. The upgrade utility renames the existing .ser profile as _old.ser and creates a new one with the original name.

Example 13-40 Output listing of the upgrade utility

```
Saving the copy of profile as EmpDtl1J_SJProfile0_old.ser
Customizing Profile
Obtaining information from old profile
Upgrade Successful
```

Table 13-10 summarizes the changes for migrating to the JCC driver.

Table 13-10 DB2 V8 migrating SQLJ stored procedure from legacy driver to JCC

	DB2 V8 using Legacy Driver	DB2 V8 using Universal Driver (JCC)
JAVAENV Contents	CLASSPATH=/SC63/sg247083/DB8AU/spjava DB2_HOME=/usr/lpp/db2/db8a JAVA_HOME=/usr/lpp/java/IBM/J1.4 The above names could be different at each site, depending upon where the HFS libraries for Java and DB2 are loaded by the system programer. Typically, the Java libraries would be in: /usr/lpp/java/IBM/J1.4 and the db2 libraries in /usr/lpp/db2/db2810. All the stored procedure class files that are developed are kept in the CLASSPATH Library: /SC63/sg247083/DB8AU/spjava	CLASSPATH=/SC63/sg247083/DB8AU/jcc/spjava JCC_HOME=/usr/lpp/db2/db8a/jcc JAVA_HOME=/usr/lpp/java/IBM/J1.4 Notice that the DB2_HOME does not appear, instead the JCC_HOME is mentioned. The JCC_HOME has a jcc subdirectory. Typically, the Java libraries would be in: /usr/lpp/java/IBM/J1.4 and the db2 libraries in /usr/lpp/db2/jcc/db2810/jcc. All the stored procedure class files need to be copied to the new CLASSPATH directory: /SC63/sg247083/DB8AU/jcc/spjava
WLMENV	DB8ADJ1	DB8ADJC2
DDL	CREATE PROCEDURE DEVL7083.EMPDTL1J (IN EMPNO CHARACTER(6), OUT FIRSTNAME VARCHAR(12), OUT MIDINIT CHAR(1), OUT LASTNAME VARCHAR(15), OUT WORKDEPT CHAR(3), OUT SALARY DECIMAL(9,2), OUT OUTPUTMESSAGE VARCHAR(250)) EXTERNAL NAME 'EmpDtl1J.GetEmpDtls' LANGUAGE JAVA PARAMETER STYLE JAVA COLLID DEVL7083 PROGRAM TYPE SUB WLM ENVIRONMENT DB8ADJ1	In order to use the driver alter the stored procedure to the new WLM ENVIRONMENT. a) ALTER PROCEDURE DEVL7083.EMPDTL1J WLM ENVIRONMENT DB8ADJC2; b) After the ALTER DDL , copy the following class files to the newCLASSPATH directory: /SC63/sg247083/jcc/spjava EmpDtl1J.class EmpDtl1J.java EmpDtl1J.sqlj EmpDtl1J_Ctx.class EmpDtl1J_SJProfile0.ser EmpDtl1J_SJProfileKeys.class
Upgrade		Run the db2sqljupgrade utility to customize the .ser files as shown in Example 13-39.
Profile	Depending on what environment you are running, JCC or legacy driver, you need to have an appropriate profile. A profile comes into play whenever you prepare a Java stored procedure or application. When you run any Java applications in USS, the profile is used to set the appropriate environment. For Java stored procedures, the runtime environment is not controlled by the contents of the profile data set, instead it is controlled by the contents of JAVAENV. However, when you prepare a Java stored procedure, the profile contents are used.	

	DB2 V8 using Legacy Driver	DB2 V8 using Universal Driver (JCC)
Sample profile	JAVA_HOME=/usr/lpp/java/IBM/J1.4 PATH=/usr/lpp/java/IBM/J1.4/bin:$PATH PATH=/usr/lpp/db2/db8a/bin:$PATH export PATH LIBPATH=/usr/lpp/db2/db8a/lib:$LIBPATH export LIBPATH LD_LIBRARY_PATH=/usr/lpp/db2/db8a/lib export LD_LIBPATH_PATH CLASSPATH=/usr/lpp/db2/db8a/classes/db2j2classes.zip:. export CLASSPATH STEPLIB=DB8A8.SDSNEXIT:DB8A8.SDSNLOAD:D8A8.SDSNLOD2:$STEPLIB STEPLIB=CEE.SCEERUN:$STEPLIB export STEPLIB export DB2SQLJPROPERTIES=/SC63/sg247083/DB8AU/db2sqljjdbc.properties Notice that there is no mention of the jcc directories.	JAVA_HOME=/usr/lpp/java/IBM/J1.4 PATH=/usr/lpp/java/IBM/J1.4/bin:$PATH PATH=/usr/lpp/db2/db8a/jcc/bin:$PATH export PATH LIBPATH=/usr/lpp/db2/db8a/jcc/lib:$LIBPATH export LIBPATH LD_LIBRARY_PATH=/usr/lpp/db2/db8a/jcc/lib export LD_LIBPATH_PATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc.jar:$CLASSPATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc_javax.jar:$CLASSPATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/sqlj.zip:$CLASSPATH CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc_license_cisuz.jar:$CLASSP export CLASSPATH STEPLIB=DB8A8.SDSNEXIT:DB8A8.SDSNLOAD:DB8A8.SDSNLOD2:$STEPLIB STEPLIB=CEE.SCEERUN:$STEPLIB export STEPLIB

The error message is shown in Example 13-41 on page 225.

Example 13-41 Error listing - Trying to run a sqlj stored procedure without upgrade

```
'SQLException raised, SQLState = 46130 SQLCODE = 0 :profile EmpDtl1J_SJProfile0 not found:
java.lang.ClassNotFoundException: COM.ibm.db2os390.sqlj.custom.DB2SQLJProfile' CCSID: 37
```

13.12.3 Extracting a .ser file from a jar file defined to DB2

There are times when the sqlj runtime files are packaged into a jar file. Let us assume that we have the stored procedure EmpRst1J.sqlj.

1. The runtime files for EmpRst1J.sqlj are in a jar defined to DB2:

   ```
   EXTERNAL NAME 'DEVL7083.EMPLJAR:EmpRst1J.GetEmpResult'
   ```

2. We need to unload the jar file from DB2 into an HFS file. DB2 stores the JAR as a BLOB object in its catalog table:

   ```
   SELECT JAR_DATA from SYSIBM.SYSJAROBJECTS where JAR_ID ='EMPLJAR' and JARSCHEMA =
   'DEVL7083'
   ```

   We wrote a Java application to extract the BLOB into an HFS file. Example 13-42 shows a Java application, which extracts the BLOB into an HFS file. It takes three arguments: first is the schema name, second the jar ID, and third the output file.

   Example 13-42 Java application ExtractJar to extract a BLOB

   ```
   import java.sql.*;
   import java.io.*;

   public class ExtractJar {

    public static void main(String[] args) {
     String owner;
       Blob jarBlob = null;
       String sql = null;
   ```

```
    String schemaName = args[0] ;
    System.out.println("Sehema Name is " + schemaName);
    String jarID     =  args[1];
    System.out.println("Jar ID      is " + jarID);
    String fileName  = args[2];
    System.out.println("fileName    is " + fileName);
    String sqltxt;
    InputStream inpStream = null;;
    int nread;
    byte[] byteArray = new  byte[1024];

    try {
     FileOutputStream outFile = new FileOutputStream(fileName);
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     Connection con =
      DriverManager.getConnection(
       "jdbc:db2://wtsc63.itso.ibm.com:12345/DB8A",
       "paolor7",
       "bhas11");
     Statement stmt = con.createStatement();
                    sqltxt = "SELECT JAR_DATA FROM SYSIBM.SYSJAROBJECTS WHERE JAR_ID = "
                           + "'" + jarID + "'"
                           + "and JARSCHEMA = '" + schemaName + "'" ;


     ResultSet rs =
      stmt.executeQuery(sqltxt);
     if (rs.next())
     {
      jarBlob = rs.getBlob("JAR_DATA") ;
      inpStream = jarBlob.getBinaryStream() ;
     }

                   while ((nread = inpStream.read(byteArray)) > 0)
                     outFile.write(byteArray,0,nread);
                   outFile.close() ;
      File fn = new File(fileName);
      System.out.println("Extracted jar is  "+ fn.getAbsolutePath());
    } catch (SQLException e) {
     System.out.println(
      "SQLException raised, SQLState = "
       + e.getSQLState()
       + " SQLCODE = "
       + e.getErrorCode()
       + " :"
       + e.getMessage());
    } catch (Exception e) {
     System.out.println("Error found" + e.toString());

    }
  }
}
```

Example 13-43 shows the sample JCL to invoke the Java application ExtractJar. Before compiling and running the Java JDBC application, ensure that the profile is set to point to the JCC setup.

We also developed a stored procedure, EXTRACT_JAR, that extracts a jar file from DB2 and writes the file to an HFS file. A detailed description of the Extract_jar stored procedure is in 25.4.4, “Handling large BLOB columns” on page 614.

The Java application shown in Example 13-42 used a JCC Type 4 Driver. Notice the connection string; we specify the domain name and port number of the target DB2 subsystem.

Example 13-43 Command to execute the ExtractJar java application

```
//EXTRACTJ JOB (999,POK),'COBOL C/L/B/E',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=SC63,L=9999
//JOBLIB   DD DSN=CEE.SCEERUN,DISP=SHR
//JCOMP   EXEC PGM=AOPBATCH,PARM='sh -L'
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDIN    DD *
  cd /SC63/sg247083/DB8AU/jcc/spjava
  java ExtractJar DEVL7083  EMPLJAR  Employee.jar
/*
//* DEVL7083 - Schema Name EMPLJAR - JARID Employee.jar - output file
```

3. Once we download the BLOB to an HFS file, Employee.jar, we need to extract the .ser file from the jar. Issue the following commands to do the extraction:

```
jar -tf Employee.jar            This command lists the contents of the jar file

jar -xvf  Employee.jar  EmpRst1J_SJProfile0.ser     This command extracts the .ser
profile
```

4. The next step is to upgrade the EmpRst1J_SJProfile0.ser using the db2sqljupgrade utility. Example 13-39 shows the use of the upgrade utility; supply the .ser file as an argument to the utility.
5. Once the .ser file is upgraded, you need to put it back into the jar file:

```
jar -uvf  Employee.jar  EmpRst1J_SJProfile0.ser
```

6. Now you need to replace the jar file that resides in DB2. You can use the IBM-supplied DB2_REPLACE_JAR stored procedure to replace the existing jar in DB2:

```
REPLACE_JAR(file:/SC63/sg247083/DB8AU/jcc/spjava/Employee.jar,DEVL7083.EMPLJAR)
```

7. ALTER the stored procedure to point to the new JCC WLM environment.

13.13 Common problems

In this section we list common problems that the Java stored procedure developer can encounter during development.

13.13.1 WLM-related errors

Error when executing a stored procedure

If the stored procedure encounters an error, WLM may stop. The following set of operator commands can be issued to display, resume, or force WLM to stop:

Display WLM status	/DIS WLM,APPLENV=DB9AWLMJ
Resume WLM	/V WLM,APPLENV=DB9AWLMJ,RESUME
Stop WLM	/V WLM,APPLENV=DB9AWLMJ,QUIESCE

The reason you may want to quiesce and then resume WLM would be if you want to make changes to the WLM application environment proc.

JAVAENV data set has wrong length

The following error message, displayed in your WLM's joblog,

```
"IEC143I 213-3C,IGGO191A,DB9AWLMJ,DB9AWLMJ,JAVAENV,C779,ASP109,IDDED.APPL.UTIL"
```

indicates that your JAVENV data set has the wrong record length. See 13.3.7, "Setting up the JAVAENV data set for Java stored procedure execution" on page 188, for the DCB information of the JAVAENV data set. WLM will stop, and will need to be resumed.

Missing DB2_HOME or JCC_HOME environment variable

The following error message, displayed in your WLM's joblog,

```
DSNX961I DSNX9WLJ ATTEMPT TO PERFORM JNI FUNCTION CreateJavaVM 894
FAILED FOR STORED PROCEDURE . . SSN= DB9A PROC= DB9AWLMJ ASID=
03B9 CLASS= METHOD= ERROR INFO= JAVAENV FILE DOES NOT SPECIFY
DB2_HOME OR JCC_HOME.
```

indicates that your ENVAR has wrong specifications. In DB2 9 for z/OS only one parameter is allowed. It should be JCC_HOME. See 13.3.8, "Environment variables in the JAVAENV data set" on page 189 for details on how to set up these environment variables. WLM will stop and will need to be restarted.

Abend U4038

When the XPLINK(ON) parameter is not included in the JAVAENV data set, the WLM address space will not initialize, and you may get the following Abend in the console log and WLM job log:

```
IDIO001I Fault Analyzer V7R1M0 (HAC4710 2006/09/18) invoked by IDIXDCAP using
SYS1.IFA.PARMLIB(IDICNF00)
IDIO044I Current fault is a duplicate of fault ID F00197 in history file SYSU.IFA.HIST -
the duplicate
count is 5
IDIO053I Fault history file entry suppressed
IDIO002I Module DSNX9JVM, program DSNX9JVM, offset X'1CF8': Abend U4038
IEA995I SYMPTOM DUMP OUTPUT 414
USER COMPLETION CODE=4039 REASON CODE=00000000.
```

The XPLINK(ON) parameter is required with JDK 1.4.1 and up. The importance of this parameter is also discussed in 27.2.7, "Java SDKs used by IBM Data Studio" on page 661.

13.13.2 Runtime problems

DSNX961I DSNX9WLJ ATTEMPT TO PERFORM JNI FUNCTION

This message can appear in different circumstances, which may hide the real reason for an abnormal termination of WLM. The problems could be any of the following:

User does not have EXECUTE authority on the package

The following message is displayed in the WLM job log:

```
…FindClass 796 FAILED FOR STORED PROCEDURE VGWSFD.TZCONVERT . SSN= DSNW
PROC= DSNWWAID ASID= 03B9 CLASS= tzconvert/TZConvert METHOD= GetUtcOffset
ERROR INFO=
tzconvert.TZConvert VGWSFD.TZCONVERT java.sql.SQLException: SQLCODE: -551
SQLERRMC=DB2XSFA VGWSFD EXECUTE PACKAGE DSNJAR.DSNX9LDJ, Get release info
```

Make sure that you grant the necessary authorizations to execute the procedure and package to the authorization ID calling the stored procedure. See Chapter 10 of *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java,* SC18-9842-01 for details about authorization.

Invalid reference to DB2Driver

```
- FindClass
- FAILED FOR ROUTINE . .  SSN= DB9A PROC= DB9AWMLJ ASID= 003A
- CLASS=  METHOD=  ERROR INFO= java.lang.NoClassDefFoundError:
- com/ibm/db2/jcc/DB2Driver
```

indicates that your ENVAR's JCC_HOME is pointing to an invalid directory. Verify the directory where the IBM DB2 Universal Driver files are installed. It could also mean that your JCC_HOME is pointing to an old version of the JDBC Drivers rather than the Universal Driver.

Incompatible JDK levels

When a Java stored procedure is built using JDK™ 1.5, and the DB2 for z/OS server is using JDK 1.4, the stored procedure execution fails with the following message:

```
- FindClass
- FAILED FOR STORED PROCEDURE ADMF001.J_V82JARS .  SSN= V81A     PROC=
- V81AWMJU ASID= 003D  CLASS= PKG60424024051890/J_v82jars METHOD=
- j_v82jars  ERROR INFO= PKG60424024051890.J_v82jars

-   ADMF001.SQL6052402405858 java.lang.UnsupportedClassVersionError: -
PKG60424024051890/J_v82jars (Unsupported major.minor version 49.0)
```

Either build your Java stored procedure using JDK 1.4, or upgrade the server JDK to 1.5. The server JDK is set up in the JAVAENV variable, JAVA_HOME. See 13.3.2, "Ensuring that the Java SDK is at the right level" on page 183 for details on setting up this variable.

ClassNotFoundException when executing a stored procedure

This message occurs in SYSOUT of the WLM address space:

```
…Exception in thread "main"
toString string from error: 'java.lang.ClassNotFoundException: tzconvert.TZConvert
.VGWSFD.TZCONVERT'
generated error string: 'java.lang.ClassNotFoundException: tzconvert.TZConvert
.VGWSFD.TZCONVERT.(none)'
** class not found error: class 'tzconvert/TZConvert'
```

Determine whether the jar file is complete or not corrupt. It may not have been uploaded in binary mode. Use the TSO OMVS **jar** command to expand the jar and see its contents:

```
jar -tvf <jar filename>"
```

If nothing is displayed, the jar file may be corrupt.

Install_jar failed -443: FOPEN() ERROR: 111

This message occurs in the SYSOUT of the job that calls the DB2 stored procedure SQLJ.INSTALL_JAR to install the user-defined stored procedure from the OMVS path:

```
Error Statement CALL SQLJ.INSTALL_JAR(:url,:jar_name,:zero)
SQLSTATE= 38502
SQLWARN =
SQLWARN.1 =
SQLWARN.2 =
SQLWARN.3 =
SQLWARN.4 =
```

```
SQLWARN.5 =
SQLWARN.6 =
SQLWARN.7 =
SQLWARN.8 =
SQLWARN.9 =
SQLWARN.10 =
SQLERRD.1 = -818
SQLERRD.2 = 0
SQLERRD.3 = 0
SQLERRD.4 = -1
SQLERRD.5 = 0
SQLERRD.6 = 0
SQLERRP = DSNXRRTN
SQLERRMC = INSTALL_JAR.INSTALL_JAR.M1O6 FOPEN() ERROR: 111.
SQLCODE = -443
READY
```

The jar file in OMVS is read protected. Change the attributes to 755 using the following command:

```
chmod 755 <jar filename>
```

The call to SQLJ.INSTALL_JAR should now run without any error.

No result sets returned to the calling program

In the SYSOUT of the WLM job, the following message is shown:

```
Number of result sets is 0
Return parm is 0
```

The problem may be resolved by the following changes to your Java code:

- Use .equals instead of == for example (Country.equals("668"))
- Use != null instead of != " " for example (timeZoneID != null)

-450

The length of the variable assigned to an output parameter in the CREATE PROCEDURE statement is not sufficient to receive the data from DB2. Make sure the lengths are compatible. Consult Table 18 in *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java,* SC18-9842-01, for more information on data type and SQL type mappings.

-805

A -805 is usually indicative of an inconsistency between the COLLID used in the CREATE PROCEDURE statement and the COLLID used when the Java package was bound.

A message similar to the following is shown in the WLM log:

```
-8O5 DBRM OR PACKAGE NAME location-name.collection-id.dbrm-name.consistency -token NOT
FOUND IN PLAN plan-name. REASON reason
parm 6 is var String(248); ' SQL exception raised, SQLState = 51OO2 SQLCode = -8O5 :
ÄIBM/DB2ÜÄT2zos/2.1O.59ÜT2zosConnection.flowConnect:execConnect:13O1:DB2 engine SQL
error, SQLCODE = -8O5, SQLSTTATE = 51OO2, error tokens =
BOE2_DSNW.VGWSFD_21.SYSSTAT.5359534C564C3O31;RGECON' CCSID: 5OO
```

To solve this problem, check the routine definitions in the DB2 catalog and make sure that the COLLID in the CREATE PROCEDURE statement matches the COLLID you specified in the DB2Binder utility.

If the COLLID in the CREATE PROCEDURE statement is blank, the collection ID of the stored procedure is the same as the collection ID of the program that calls it. Redefine the routine or the calling program so that the COLLIDs match.

For more details on this, see Chapter 9 of *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java,* SC18-9842-01, including the discussion on how to use the CURRENT PACKAGESET special register for setting the COLLID.

-20204 during execution

Care should be taken when coding the EXTERNAL NAME clause in the CREATE PROCEDURE statement, as this can cause the following error message:

```
DEVL7083.EMPDTLSJ - Exception occurred while running:
A database manager error occurred.SQLCODE: -20204, SQLSTATE: 46008 - THE USER-DEFINED
FUNCTION OR PROCEDURE DEVL7083.EMPDTLSJ WAS UNABLE TO MAP TO A SINGLE JAVA METHOD.
SQLCODE=-20204, SQLSTATE=46008, DRIVER=3.50.152
```

Ensure that the EXTERNAL NAME follows the syntax for using jars and packages as explained in 13.7, “Making the stored procedure class files available to DB2” on page 201.

-204 name IS AN UNDEFINED NAME

Setting different values for the CURRENT SQLID and CURRENT SCHEMA can cause a -204 when executing a stored procedure, as typified by the following error message:

```
parm 6 is var String(248); ' SQL exception raised, SQLState = 42704 SQLCode = -204 :
ÄIBM/DB2ÜÄT2zos/2.10.59ÜT2zosStatement.readPrepareDescribeOutput_:nativePrepareInto:
1377:DB2 engine SQL error, SQLCODE = -204, SQLSTATE = 42704, error tokens =
MINSK03.RGEVTZCS ' CCSID: 500
```

The problem is that the user's authorization ID (MINSK03) is used as implicit qualifier for the table (RGEVTZCS) and not the schema (VGWSFD) under which the table was created and the plan of the calling program was bound.

To resolve this problem:

- Define the JDBC properties file in OMVS and include a -Ddb2.jcc.currentSchema=<schema name>. For example, in OMVS, the file will look like this:

```
# Set default schemaname
-Ddb2.jcc.currentSchema=VGWSFD
```

- Include the data set into JAVAENV via the ENVAR parameter as follows:

```
XPLINK(ON),
ENVAR("JCC_HOME=/usr/lpp/db2810/jcc",
"JAVA_HOME=/usr/lpp/java/J1.4",
"JVMPROPS=/u/de59887/jdbcprop")
```

See JVMPROPS in 13.3.8, “Environment variables in the JAVAENV data set” on page 189 for other applications of the JVMPROPS environment variable.

14

External SQL procedures

In this chapter we focus on the development of external stored procedures using the SQL Procedures language. We refer to two simple applications accessing sample tables: the first for retrieving employee information for a specific employee number, and the second for retrieving a list of employees for a specific department.

If you are not familiar with the SQL Procedures language capabilities, refer to Appendix D. “SQL control statements for external SQL procedures” in *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854 for details.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions are in Appendix B, “Additional material” on page 887.

Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 24 to decide what components are applicable to your environment.

SQL stored procedures were introduced to DB2 with Version 5. Before V9, only one kind of SQL stored procedure existed. DB2 9 for z/OS introduced a new kind of SQL stored procedures, the *Native SQL stored procedures*. Refer to Chapter 15, “Native SQL procedures” on page 253 if you want to learn more about the new kind of SQL procedures in DB2 for z/OS.

The way SQL stored procedures have been implemented before V9 is today referred to as *External SQL stored procedures*.

Tip: If you have not yet started using SQL stored procedures, we recommend that you start with Native stored procedures provide more features and are easier to handle than external stored procedures. If you are already using SQL stored procedures, you might want to migrate them to Native stored procedures in order to be able to make use of the enhancements that Native stored procedures provide in comparison to External SQL stored procedures.

This chapter contains the following:

- Verifying the environment

- Defining an SQL procedure
- Handling error conditions
- Migrating to native SQL stored procedures

14.1 Verifying the environment

Before developing the stored procedure, it is important to have a clear understanding of the various steps that must be completed for a stored procedure to execute successfully. These steps are covered in detail in the rest of the book and we simply list them here for convenience. They are:

1. The WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 39 for details.
2. The LE environment must be set up. See Chapter 5, “Language Environment setup” on page 47 for details.
3. The stored procedure must be defined to DB2. Note in particular that if the stored procedure is designed to return result sets, the maximum number of result sets that can be returned is specified in the definition. See Chapter 9, “Defining stored procedures” on page 91 for details.
4. Develop the stored procedure. See 14.2.1, “Preparing and binding an SQL procedure” on page 236 for details.
5. Grant the necessary privileges to the authorization ID of the user that executes the stored procedure. See Chapter 7, “Security and authorization” on page 65 for details.
6. Develop the calling application if needed.
7. Also see Chapter 16, “Debugging” on page 313 for details on testing and debugging.

14.1.1 What is different about an SQL procedure?

The main difference between an external stored procedure and an SQL procedure is the location of the processing logic. For an external stored procedure, the definition of the stored procedure specifies the parameters associated with it, as discussed in Chapter 9, “Defining stored procedures” on page 91. The EXTERNAL parameter specifies the load module associated with the stored procedure, and the source code corresponding to that load module is located in a source library external to DB2 (for example, a COBOL source would be a program source library). For an SQL procedure, the processing logic is embedded within the CREATE PROCEDURE statement itself. For example, an external procedure that accepts an employee number and a raise percent to update the employee salary looks like this:

```
CREATE PROCEDURE GIVRAISE
( IN PEMPNO CHAR(6)
, IN PRAISEPCT DEC(6,2)
)
LANGUAGE COBOL
EXTERNAL NAME PGM00;
```

The equivalent statement for an SQL procedure is:

```
CREATE PROCEDURE GIVRAISE
( IN PEMPNO CHAR(6)
, IN PRAISEPCT DEC(6,2)
)
LANGUAGE SQL
UPDATE EMP
SET SALARY = SALARY * (1 + PRAISEPCT/100)
WHERE EMPNO = PEMPNO;
```

There is one important difference between external SQL stored procedures and other external stored procedures, which is the manner in which errors are handled. In general, DB2 automatically returns the SQL conditions through the SQLCA for SQL procedures. This

requires additional work for external stored procedures, and is sometimes not possible at all. See 14.3, “Handling error conditions” on page 245 for error handling in SQL procedures as well as 16.1.5, “Unhandled SQL errors to CALL statements” on page 323 for error handling in external stored procedures.

14.2 Defining an SQL procedure

In this section we show the steps and the parameters used for defining an SQL language stored procedure.

14.2.1 Preparing and binding an SQL procedure

Note that the CREATE PROCEDURE statement is input to the stored procedure definition process (traditionally the DDL) as well as to the preparation process (traditionally the source) as pointed out in 14.1.1, “What is different about an SQL procedure?” on page 235. The relevant source code must therefore be extracted from the CREATE PROCEDURE statement.

When you use the client-based IBM DB2 Developer Workbench or newer Data Studio to define the SQL procedure to DB2, the preparing and binding is automatically done for you by the SQL procedure build utility. If you do not use the Developer Workbench or Data Studio, you must do the following:

- Use the DB2 SQL precompiler to convert the SQL procedure source statements into a C language program.
- Process the resulting C program as if they are in any other SQL application program through the DB2 precompiler or the SQL statement coprocessor.
- Bind the resulting DBRM into a package.

For the first two steps mentioned above the installation process provides you with JCL procedure DSNHSQL.

14.2.2 Handling terminator defaults

Each CREATE PROCEDURE statement consists of the procedure header and the procedure body. The procedure body always contains embedded semicolons at the end of each statement.

As a result, you need to change statement terminators in DSNTIAD, DSNTEP2, DSNTEP4 and SPUFI when processing a CREATE statement for an SQL procedure, because DSNTIAD, DSNTEP2, DSNTEP4, and SPUFI default to semicolon for their statement terminator symbol.

If you decide to change the SQL statement terminator, for example to the percent sign (%), you can do this as follows for each of the four mentioned SQL user interfaces:

- For DSNTEP2: Add PARMS('/SQLTERM(%)') keyword of DSN RUN command.
- For DSNTEP4: Add PARMS(’/SQLTERM(%)’) keyword of DSN RUN command.
- For DSNTIAD: Add PARMS('SQLTERM(%)') keyword of DSN RUN command.

Important: Note that DSNTEP2 and DSNTEP4 are PL/I programs. For PL/I programs you specify the parameter list of the form A/B, where A represents a list of runtime options and B represents a list of parameters for the PL/I application program. If runtime options are not needed, write the list in the form /B.

Options such as SQLTERM belong to the parameters for the PL/I application program. As a consequence you must make sure that they are specified after the slash (/).

DSNTIAD is an assembler program. There are no runtime parameters for assembler programs. As a consequence there are no two sections in the list of parameters and therefore you must omit the slash here.

- For SPUFI: Choose **Change defaults** in the SPUFI input panel, and change the SQL terminator to some special character other than semicolon (;), such as percent (%).

You can also change the SQL terminator by specifying --#SET TERMINATOR *symbol* just before your SQL statement. So if you intended to change the symbol to ampersand (&), you could do so by coding the following statement in your SPUFI editor:

```
--#SET TERMINATOR &
SELECT * FROM SYSIBM.SYSROUTINES &
CREATE PROCEDURE .....
```

Refer to Table 14-1 for a list of invalid characters for use as SQL statement terminator.

Table 14-1 Invalid special characters for SQL statement terminators

Name	Character	Hexadecimal representation
blank		X'40'
comma	,	X'5E'
double quote	“	X'7F'
left parenthesis	(	X'4D'
right parenthesis	)	X'5D'
single quote	’	X'7D'
underscore	_	X'6D'

14.2.3 Handling comment lines

When you do not use the Developer Workbench or Data Studio, some language elements that are valid in one SQL statement interface may be invalid in another. Some language elements that are valid for the SQL precompiler (such as /*.... */ as comment lines) are invalid in SPUFI. For instance, the comment lines in bold in Example 14-1 should be deleted in order to be able to submit the statement in SPUFI.

Example 14-1 Comment lines not allowed in SPUFI

```
/*********************************************************/
/*  description of stored procedure                      */
/*********************************************************/
DROP   PROCEDURE DEVL7083.EMPVER8S
#
CREATE PROCEDURE DEVL7083.EMPVER8S
(
```

```
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
...
)
...
```

14.2.4 Statements in an SQL procedure

In this section, we discuss and show examples of statements that can be included in the body of an SQL procedure.

Assignment

The assignment statement assigns a value to an output parameter or to an SQL variable. See Example 14-2.

Example 14-2 Assignment statement

```
SET I = 1 ;
SET PSQLERRMC = 'SEVERE ERROR OCCURED - SEE LOG FOR DETAILS';
```

CALL

The CALL statement invokes a stored procedure. This procedure can be an authorized procedure written in any language (for example, COBOL, Java, etc.). See Example 14-3.

Example 14-3 CALL statement

```
CALL EMPDTLSC( PEMPNO
              ,PFIRSTNME
              ,PMIDINIT
              ,PLASTNAME
              ,PWORKDEPT
              ,PHIREDATE
              ,PSALARY
              ,PSQLCODE
              ,PSQLSTATE
              ,PSQLERRMC
             );
```

CASE

The CASE statement selects an execution path based on the evaluation of one or more conditions. See Example 14-4.

Example 14-4 CASE statement

```
CASE  TMPVAR
      WHEN 1 THEN
         SELECT SUM(SALARY)
         INTO   TOTSAL
         FROM   EMP
         WHERE  SALARY > 50000;
      WHEN 2  THEN
         SELECT SUM(SALARY)
         INTO   TOTSAL
         FROM   EMP
         WHERE  SALARY BETWEEN 30000 AND 50000;
      ELSE
         SELECT SUM(SALARY)
```

```
        INTO   TOTSAL
        FROM   EMP
        WHERE  SALARY < 30000;
END CASE;
```

GOTO

The GOTO statement causes a branch to a user-defined label within an SQL procedure. See Example 14-5.

Example 14-5 GOTO statement

```
DOIT: ...

IF  TEMPVAR = 100  THEN GOTO DOIT;
                   ELSE SET TEMPVAR = TEMPVAR + 1;
END IF;
```

IF

The IF statement selects an execution path based on the evaluation of a condition. See Example 14-6.

Example 14-6 IF statement

```
IF  TEMPVAR = 100  THEN GOTO DOIT;
                   ELSE SET TEMPVAR = TEMPVAR + 1;
END IF;
```

LEAVE

The LEAVE statement transfers program control out of a loop or a compound statement. See Example 14-7.

Example 14-7 LEAVE statement

```
OPEN C2;
GETEACH: LOOP
   FETCH C2 INTO MYEMPNO, MYSALARY ;
   IF  SQLCODE = 100 THEN LEAVE GETEACH;
   END IF;
END LOOP;
```

LOOP

The LOOP statement executes a statement or a group of statements multiple times. See Example 14-8.

Example 14-8 LOOP statement

```
OPEN C2;
GETEACH: LOOP
   FETCH C2 INTO MYEMPNO, MYSALARY;
   IF  SQLCODE = 100 THEN LEAVE GETEACH;
   END IF;
END LOOP;
```

REPEAT

The REPEAT statement executes a statement or a group of statements until a search condition is true. See Example 14-9.

Example 14-9 REPEAT statement

```
SET I = 1 ;
DOIT: REPEAT
UPDATE    EMP
SET       SALARY = SALARY + 0.01
WHERE     WORKDEPT = PDEPTNO;
SET I = I + 1 ;
UNTIL I > 5
END REPEAT DOIT;
```

WHILE

The WHILE statement executes a statement or a group of statements while a search condition is true. See Example 14-10.

Example 14-10 WHILE statement

```
SET I = 1 ;
WHILE I < 6
DO
UPDATE    EMP
SET       SALARY = SALARY + 0.01
WHERE     WORKDEPT = PDEPTNO;
SET I = I + 1 ;
END WHILE;
```

Compound statement (BEGIN... END)

A compound statement contains one or more of any of the other types of statements in this list. In addition, a compound statement contains a group of statements and declarations for SQL variables, cursors, and condition handlers. The order in which they can appear is:

1. SQL variables, condition declarations, and return code declarations
2. Cursor declarations
3. Handler declarations
4. SQL procedure statements

Example 14-11 shows an example.

Example 14-11 Compound statement

```
BEGIN
DECLARE C2 CHAR(30);
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
DECLARE c1 CURSOR FOR
SELECT deptno, deptname, admrdept
FROM department
ORDER BY deptno;
SELECT FIRSTNME , MIDINIT , SALARY
INTO   PFIRSTNME , PMIDINIT , PSALARY
FROM   EMP
WHERE  EMPNO = PEMPNO ;
SELECT SQLCODE, SQLSTATE INTO PSQLCODE, PSQLSTATE FROM SYSIBM.SYSDUMMY1;
SET PSQLERRMC  = 'ADIOS';
```

```
END
```

GET DIAGNOSTICS

The GET DIAGNOSTICS statement obtains information about the execution status of the previous SQL statement that was executed. See Example 14-12.

Example 14-12 GET DIAGNOSTICS statement

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
GET DIAGNOSTICS CONDITION
    1 PSQLERRMC = MESSAGE_TEXT
    , PSQLCODE  = DB2_RETURNED_SQLCODE
    , PSQLSTATE = RETURNED_SQLSTATE;
```

ITERATE

The ITERATE statement causes the flow of control to return to the beginning of a loop. ITERATE is only allowed in looping statements such as LOOP, REPEAT, WHILE. See Example 14-13.

Example 14-13 ITERATE statement

```
DECLARE C2 CURSOR WITH RETURN FOR
SELECT    EMPNO, SALARY
FROM      EMP
WHERE     WORKDEPT = PDEPTNO
ORDER BY  EMPNO;
...
OPEN C2;
GETIT: LOOP
   FETCH C2 INTO MYEMPNO, MYSALARY ;
   IF  SQLCODE = 100 THEN LEAVE GETIT;
   END IF;
   ITERATE GETIT;
END LOOP;
```

SIGNAL

The SIGNAL statement is used to return an error or warning condition to the calling program. It causes an error or warning to be returned with the specified SQLSTATE along with an optional message text. See Example 14-14.

Example 14-14 SIGNAL statement

```
DECLARE EXIT     HANDLER FOR SQLSTATE VALUE '57011'
 SIGNAL SQLSTATE '75001'
       SET MESSAGE_TEXT =
          'CANNOT GET TO EMP, TRY AGAIN AND THEN CALL DBA';
```

In this case, the calling program receives:

```
SQL0438N  Application raised error with diagnostic text: "CANNOT GET TO EMP, TRY AGAIN
AND THEN CALL DBA".  SQLSTATE=75001
```

RESIGNAL

Like the SIGNAL statement, RESIGNAL is used to return an error or warning condition to the calling program. It causes an error or warning to be returned with the specified SQLSTATE along with an optional message text. This statement is valid within a handler only. Note that RESIGNAL, unlike SIGNAL, can be issued with no SQLSTATE operand to re-raise the condition that caused the handler to be invoked. See Example 14-15.

Example 14-15 RESIGNAL statement

```
DECLARE EXIT     HANDLER FOR SQLSTATE VALUE '22003'
RESIGNAL SQLSTATE '75002'
      SET MESSAGE_TEXT =
         'ATTEMPT TO DIVIDE BY ZERO - CORRECT AND TRY AGAIN';
```

In this case, the calling program receives:

```
SQL0438N  Application raised error with diagnostic text: "ATTEMPT TO DIVIDE BY ZERO -
CORRECT AND TRY AGAIN".  SQLSTATE=75002
```

RETURN

The RETURN statement is used to return from the routine. Optionally, it can return an integer status value (the return code).

See Example 14-16.

Example 14-16 RETURN statement

```
DECLARE ANY_ERRORS CHAR(1);
...
IF  ANY_ERRORS = 'Y' THEN RETURN 16;
                     ELSE RETURN 0;
END IF;
```

Note: SQL statements in SQL procedures

A subset of the SQL statements that are described in Chapter 5 of *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854 can be specified in an SQL procedure. Note that certain statements are valid in a compound statement but not valid if the statement is the only statement in the procedure body. This is shown in Appendix A of the same manual.

14.2.5 Declaring and using variables

You can declare SQL variables that you use only within an SQL procedure. SQL variables are like host variables in external stored procedures. They can have the same data types and lengths as SQL procedure parameters such as CHAR, DECIMAL, INTEGER, and so on.

Note the following restrictions on SQL variables:

- SQL variable names can be up to 128 bytes in length and can include alphanumeric characters and the underscore character.
- Although lowercase characters are allowed in variable names, DB2 converts all variable names to uppercase. Due to that fact two SQL variables such as myvar and MYVAR are identical to DB2 and therefore not allowed within one procedure.
- You cannot declare an SQL variable with the same name as a parameter name. In V8, this restriction is lifted: you can have an SQL variable with the same name as a parameter.
- When you use an SQL variable in an SQL statement, do not precede it with a colon.

- When you use a parameter in an assignment statement, all parameters, not just those declared as OUT or INOUT, can be modified.
- The restriction not to use an SQL reserved word as an SQL variable name was removed in DB2 V8.I

Variable names are implicitly or explicitly qualified and we suggest the following guidelines:

- When you use an SQL procedure parameter, variable or column name in the procedure body, qualify them with the procedure name as shown in Example 14-17 in the cases where they might be ambiguous. For instance, a column name with the same name as a variable.

Example 14-17 Qualifying a parameter

```
CREATE PROCEDURE DEVL7083.EMPRSETS
(
 IN  PDEPTNO       CHAR(3)
,OUT PDEPTNAME     VARCHAR(36)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
) ...
SET EMPRSETS.PSQLCODE   = SQLCODE ;
```

- Specify a label for each compound statement and qualify SQL variable names in the compound statement with that label as shown in Example 14-18.

Example 14-18 Qualifying a SQL variable

```
P1: BEGIN
DECLARE I       INTEGER;
SET P1.I = 1;
```

- Qualify column names with the associated table or view names as shown in Example 14-19.

Example 14-19 Qualifying a column name

```
DECLARE C1 CURSOR WITH RETURN FOR
SELECT EMP.EMPNO, EMP.FIRSTNME
, EMP.MIDINIT, EMP.LASTNAME, EMP.HIREDATE, EMP.SALARY
FROM EMP
WHERE EMP.WORKDEPT = PDEPTNO
ORDER BY EMP.SALARY DESC;
```

14.2.6 Passing parameters

Example 9-7 on page 110 shows our first sample SQL language stored procedure. More examples are available as described in Appendix B, “Additional material” on page 887.

A stored procedure can receive and send back parameters to the calling application. When the calling application issues an SQL CALL to the stored procedure, DB2 builds a parameter list based on the parameters passed in the SQL call and the information specified when the stored procedure is initially defined. For an SQL procedure retrieving information about a specific employee, the parameter list specified when defining the stored procedure is shown in Example 14-20.

Example 14-20 Parameter list

```
CREATE PROCEDURE DEVL7083.EMPDTLSS
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
) ...
```

This definition specifies whether the parameter is IN (input to the stored procedure), OUT (output from the stored procedure), or INOUT (input to and output from the stored procedure). It also specifies the data type and size of each parameter. This list must be compatible with the parameter list in the calling application.

SQL procedures have no explicit LINKAGE section and the passing of arguments is accomplished using the normal conventions. For example, an input parameter is received based on the call sequence when it is invoked.

An output parameter value is returned as set by the stored procedure. There is no explicit RETURN statement needed.

14.2.7 Actions for the calling application

The calling application must initialize all passed parameters and call the stored procedure. The call is shown in Example 14-21.

Example 14-21 Calling application

```
EXEC SQL
     CALL EMPDTLSS( :PEMPNO
                   ,:PFIRSTNME
                   ,:PMIDINIT
                   ,:PLASTNAME
                   ,:PWORKDEPT
                   ,:PHIREDATE
                   ,:PSALARY
                   ,:PSQLCODE
                   ,:PSQLSTATE
                   ,:PSQLERRMC
                  )
END-EXEC.
```

If the stored procedure must return a result set, additional processing is required. This is discussed in 14.2.9, “Handling result sets” on page 245.

14.2.8 Actions that the stored procedure must take

The stored procedure behaves just like any subprogram, taking action based on input parameters (if any) and setting the values of the output parameters (if any). If the stored procedure must return a result set, additional processing is required. This is discussed in 14.2.9, “Handling result sets” on page 245.

14.2.9 Handling result sets

When the stored procedure returns a small number of parameters, it is much simpler to avoid result sets altogether, returning them as parameters as discussed above. When the stored procedure must return result sets, each consisting of multiple rows, there are two alternatives:

- Handling a fixed number of result sets for which you know the contents
- Handling a variable number of result sets, for which you do not know the contents

The first alternative is simpler to develop, but the second alternative is more general and requires minimal modifications if the calling program or stored procedure changes. We will now discuss each of these alternatives in detail.

The following steps are required to handle result sets:

1. When defining the stored procedure to DB2 (see Chapter 9, “Defining stored procedures” on page 91), specify the maximum number of result sets that could be generated by the stored procedure.
2. In the stored procedure, declare and open one cursor for each result set. Note that the stored procedure must not fetch rows from the cursor nor close the cursor. Each such cursor must be declared using the WITH RETURN clause.
3. In the calling program, process the rows as discussed in 10.2.8, “Handling result sets in the calling program” on page 128.

14.2.10 Redeploying SQL procedures

DB2 builds external SQL procedures as external C load modules, but the resulting load modules *must not* ever be defined to DB2 as a LANGUAGE C stored procedure. Some customers have made the mistake of doing this as part of a redeployment of the SQL procedure. The invocation of those stored procedures fails.

You can redeploy the SQL procedures without carrying over and redefining the possibly large source SQL by moving the C module and DBRM, but the stored procedure must continue to be defined to DB2 as LANGUAGE SQL on the CREATE PROCEDURE statement, which should be invoked as any other DDL statement. The procedure body on this CREATE PROCEDURE... LANGUAGE SQL can be any valid simple SQL statement, such as SET CURRENT DEGREE = 'ANY', as DB2 will not use this procedure body when the re-deployed SQL procedure referencing the C load module is invoked.

14.3 Handling error conditions

You can detect and pass back to the calling program SQL errors and SQL warnings by using a combination of various techniques. We discuss these in the following sections:

- You can include specific statements, the so-called handlers to trap the error or warning conditions. We discuss this in 14.3.1, “Using handlers in an SQL procedure” on page 246.
- You can return a condition code to the calling application. We discuss this in 14.3.2, “Using the RETURN statement for the SQL procedure status” on page 249.
- You can use SIGNAL or RESIGNAL to raise a specific SQLSTATE and a text message. We discuss this in 14.3.3, “Using SIGNAL and RESIGNAL to raise a condition” on page 250.
- When called by a trigger, you may have a need to force a negative SQL code so that the trigger fails. We discuss this in 14.3.4, “Forcing errors in an SQL procedure when called by a trigger” on page 250.

14.3.1 Using handlers in an SQL procedure

If an SQL error occurs when the SQL procedure executes, the SQL procedure terminates unless you include statements called *handlers* to tell the procedure to take some other action.

Handlers are similar to WHENEVER statements in an external SQL application program. You can handle the following:

- SQL errors
- SQL warnings
- Not found/no more rows conditions
- Specific SQLSTATEs

The general form of the handler declaration is:

```
DECLARE handler-type HANDLER FOR condition SQL-procedure-statement;
```

When a situation occurs that matches the `condition`, the `SQL-procedure-statement` executes. The action specified in a condition handler can be any SQL statement.

Important: In external stored procedures the SQL-procedure-statement can only be a single SQL statement. Compound statements in condition handlers are only allowed in Native SQL stored procedures. Multiple statements can be defined in a handler body in an external SQL procedure by using a control statement other than a compound-statement, for example:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
  L1: LOOP
    stmt1;
    stmt2;
    ...
    LEAVE L1;
  END LOOP L1;
```

After the statement completes, DB2 performs the action indicated by the *handler-type* (CONTINUE or EXIT). For CONTINUE, the execution continues with the statement after the statement that caused the handler to be activated. For EXIT, execution skips to the end of the compound statement that contains the handler.

CONTINUE handler examples

In this section we show you some possible scenarios and behaviors for CONTINUE handlers.

Example 1

The sample in Figure 14-1 on page 247 shows a CONTINUE handler that is looking for SQLWARNING conditions. If an SQLWARNING occurs, the handler is fired (1). The result of statement2 in this example is SQLCODE +562. The action defined in the handler (SET CODE = SQLCODE) sets variable CODE to 562 (2). Since this is a CONTINUE handler, the execution of the procedure continues with the next statement (3) and finally stops its execution after it reaches the END statement.

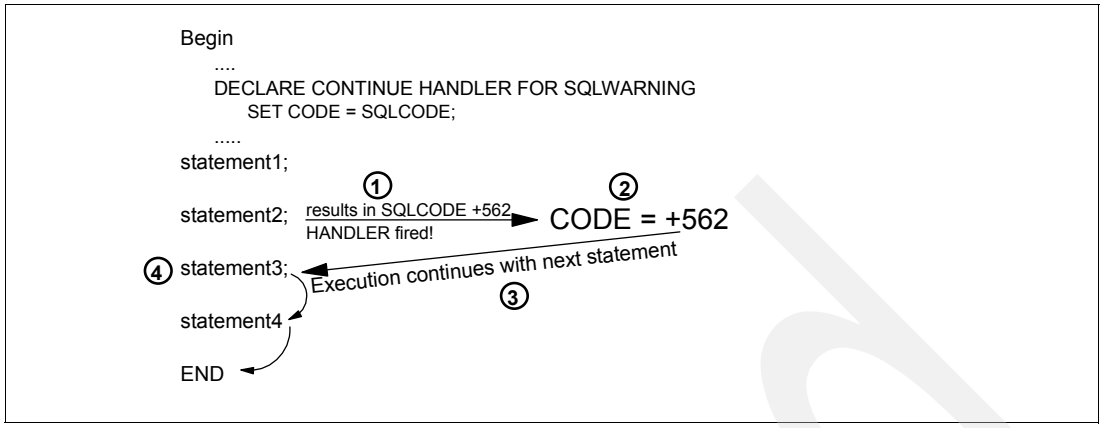

Figure 14-1 CONTINUE handler 1

Example 2

If the error that raised the exception is a FOR, IF, CASE, WHILE, or REPEAT statement, then control returns to the statement that follows the END FOR, END IF, END CASE, END WHILE, or END REPEAT.

This means that if the evaluation of the search-condition in an IF statement raises a condition, and the condition invokes a continue handler, then processing will continue with the statement following the IF statement. However, if a statement within the IF statement raises a condition which invokes a continue handler, processing continues with the next statement following the one that raised the condition.

In Example 14-2, the procedure does not continue after the END IF although the CONTINUE handler is fired, because the error occurred inside the IF ... END IF block. (Actually, it works exactly the same way as it does in the sample described in Figure 14-1.)

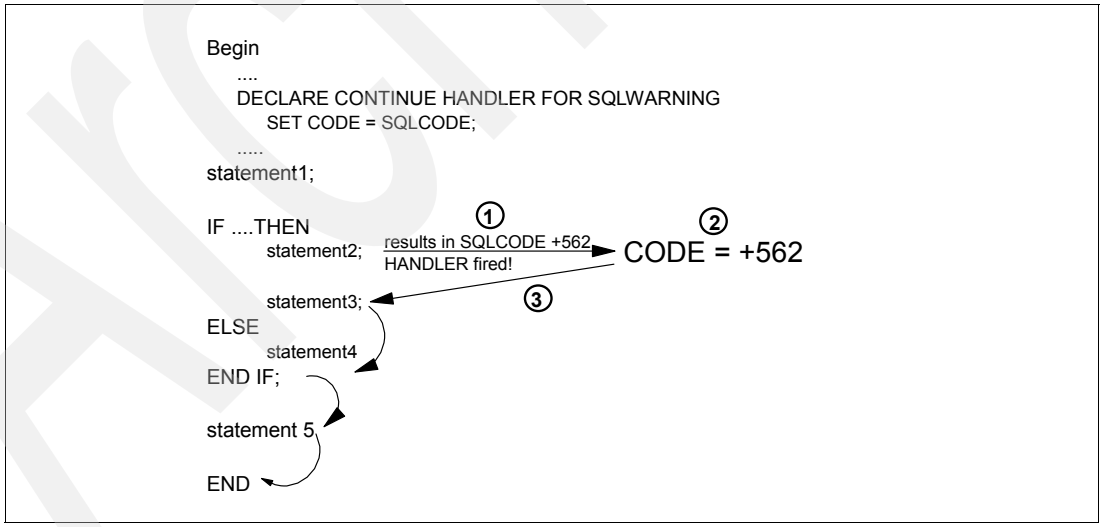

Figure 14-2 CONTINUE handler 2

Example 3

If the error occurs in the IF statement as shown in Figure 14-3, after assigning nnn to variable CODE (2), the procedure continues its processing with statement 5, which is the first statement after the END IF (3).

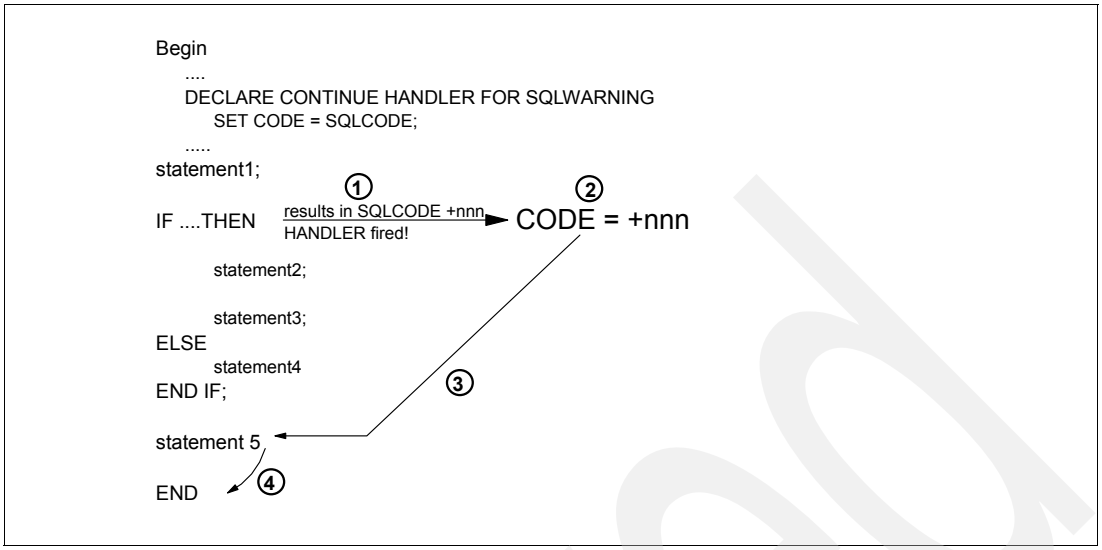

Figure 14-3 CONTINUE handler 3

Example 4

The following handler sets the END_OF_C1 flag to Y and continues after the statement that returned no rows (such as FETCH C1):

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET END_OF_C1 = 'Y';
```

EXIT handler examples

After the handler is invoked, control is returned to the end of the compound statement. As mentioned before, nested compound statements are not allowed condition handlers of external SQL stored procedures. As a consequence the EXIT handler will always not just end the compound statement but it will always end the execution of the complete procedure.

Example 1

As shown in Figure 14-4, the EXIT handler is fired by an SQLEXCEPTION that returned SQLCODE -601 (1). This SQLCODE is assigned to variable CODE (2) and the procedure continues at the end of the compound statement.

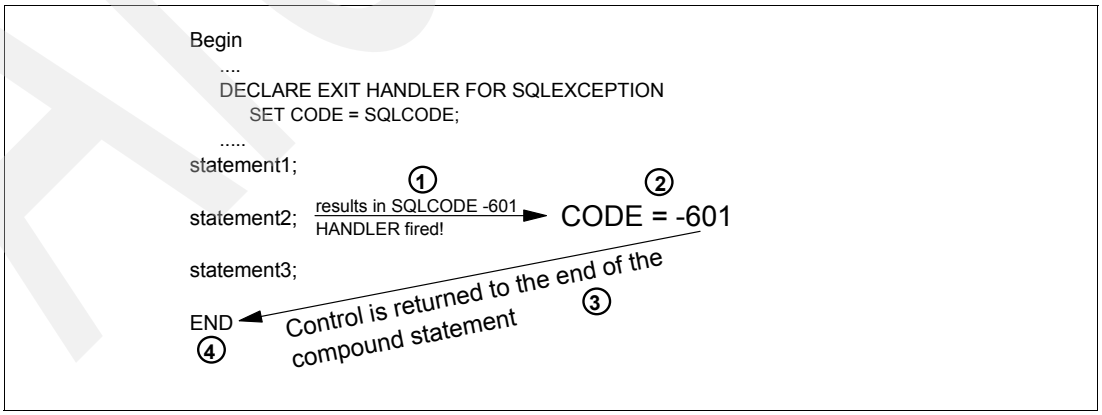

Figure 14-4 EXIT handler

Example 2

The following handler sets the OUTMSG variable to Authorization Error and exits the compound statement that generated the condition:

```
DECLARE AUTH_ERROR CONDITION FOR '42501';
...
DECLARE EXIT HANDLER FOR AUTH_ERROR
   SET OUTMSG = 'AUTHORIZATION ERROR';
```

Using SQLCODE and SQLSTATE

In general, you will want to pass the SQLCODE and SQLSTATE values to the caller in case of an error. If so, you must include output parameters (defined as OUT or INOUT) for those values. In addition you must declare them as SQL variables as follows:

```
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
```

The following code fragment shows the declaration and how it is used:

```
DECLARE SQLCODE INTEGER;
...
DECLARE EXIT HANDLER FOR SQLEXCEPTION
   SET OUTSQLCODE = SQLCODE;
```

Using GET DIAGNOSTICS in a handler

The GET DIAGNOSTICS statement provides information about the execution status of a statement similar to what is provided through SQLCA. See *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854 for a list of all variables that are returned by the GET DIAGNOSTICS statement.

Example 14-22 shows how one of the parameters, MESSAGE_TEXT, can be used. MESSAGE_TEXT identifies the message text of the error or warning returned from the SQL statement that invoked the handler.

Example 14-22 MESSAGE_TEXT statement

```
DECLARE EXIT     HANDLER FOR SQLEXCEPTION
GET DIAGNOSTICS CONDITION 1
PSQLERRMC = MESSAGE_TEXT,
PSQLCODE  = DB2_RETURNED_SQLCODE,
PSQLSTATE = RETURNED_SQLSTATE;
```

14.3.2 Using the RETURN statement for the SQL procedure status

You can use the RETURN statement in an SQL procedure to return an integer status value. If you include a RETURN statement, DB2 sets the SQLCODE in the SQLCA to 0 and the caller must retrieve the return status of the procedure in either of the following ways:

- By using the RETURN_STATUS item of GET DIAGNOSTICS to retrieve the return value of the RETURN statement
- By retrieving SQLERRD(0) of the SQLCA, which contains the return value of the RETURN statement

> **Note:** This is the first occurrence of the SQLERRD fields. The SQLERRD fields are numbered from 0 - 5 in the C language and from 1 - 6 in other high-level languages.

If you do not include a RETURN statement in an SQL procedure, then by default DB2 sets the return status to 0 for an SQLCODE that is 0 or positive and sets it to -1 for a negative SQLCODE.

14.3.3 Using SIGNAL and RESIGNAL to raise a condition

You can use the SIGNAL statement anywhere in an SQL procedure to set a specific SQLSTATE along with an optional message text. In the following code fragment, DB2 generates an SQLSTATE 23503 (SQLCODE -530) when you attempt to insert an order for a missing customer. You can detect and pass a more meaningful message back to the caller as the code fragment in Example 14-23 shows.

Example 14-23 Using SIGNAL

```
DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
   SIGNAL SQLSTATE '75001'
      SET MESSAGE_TEXT = 'Customer is unknown';
INSERT INTO ORDERS (....)
      VALUES (....);
```

The capability to set a specific SQLSTATE in case of an error is useful for packaged applications such as extenders which have their own SQLSTATEs that they want to return to the invoking application. You can achieve this by using the RESIGNAL command within the body of a handler as shown in Example 14-24.

Example 14-24 Using RESIGNAL

```
DECLARE OVERFLOW CONDITION FOR SQLSTATE VALUE '22003';
DECLARE EXIT HANDLER FOR OVERFLOW
   RESIGNAL SQLSTATE '22375'
   SET MESSAGE_TEXT 'Attempt to divide by zero';
```

Note that when you use SIGNAL or RESIGNAL to set the SQLSTATE, the value of SQLCODE returned to the invoking application is a constant (you cannot set it) based on the class code (first 2 bytes) of the SQLSTATE:

- Class code 00 is not allowed.
- Class code 00 or 01 causes SQLCODE +438.
- All other class codes cause SQLCODE -438.

14.3.4 Forcing errors in an SQL procedure when called by a trigger

Suppose a trigger invokes your SQL procedure and you encounter an error or warning situation for which you want to cause a negative SQLCODE so that the trigger will fail. You can issue a COMMIT or ROLLBACK statement within the procedure. These statements are accepted at CREATE PROCEDURE, but at runtime they violate the restriction that COMMIT and ROLLBACK statements are not allowed in procedures invoked from a trigger and generate the error shown in Figure 14-25.

Example 14-25 Error received by the trigger when called stored procedure issues a rollback

```
DSNT408I SQLCODE = -723, ERROR:  AN ERROR OCCURRED IN A TRIGGERED SQL STATEMENT
         IN TRIGGER DEVL7083.EMPTRIG1, SECTION NUMBER 2.
         INFORMATION RETURNED: SQLCODE -751, SQLSTATE 38003, AND MESSAGE TOKENS
         STORED PROCEDURE,DEVL7083.EMPAUDTS
DSNT418I SQLSTATE   = 09000 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNX9CAC SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = 0  0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'00000000'  X'00000000'  X'00000000'  X'FFFFFFFF'
         X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
```

This technique is especially useful in cases where the situation would otherwise return a zero or positive SQLCODE allowing the trigger to continue and commit the changes when you do not want it to do so.

See Chapter 26, “Using triggers and UDFs” on page 629 for details on triggers invoking a stored procedure.

14.4 Migrating to native SQL stored procedures

As stated in the introduction to this chapter, in case you are already using external SQL stored procedures, after migration to DB2 9 for z/OS NFM, you might want to consider migrating them to native SQL stored procedures.

Chapter 15, “Native SQL procedures” on page 253 introduces the advantages and great new possibilities that are offered with native SQL procedures. Future investments regarding the improvements for stored procedure handling and coding are likely to occur mostly to native SQL procedures.

For migrating external SQL stored procedures to native SQL procedures, you need to drop and recreate the stored procedure. If your external SQL procedure was created in DB2 V8, it is possible that you did not specify the FENCED or EXTERNAL keyword at creation time, since it was implied. If this is the case, you simply must drop the existing stored procedure from DB2 and re-run the CREATE PROCEDURE statement omitting both keywords once again.

However, since a different SQL dialect is used in the two types of procedures you need to provide three other significant source language adjustments beyond just enabling native SQL procedure creation:

- Adding new source option values for native procedures which reflect other critical options used during the production steps for external (the separate steps of precompile and bind package).
- Adjusting the source coding practices for SQLPL error handling, to keep the semantics intact given the various incompatibilities listed for the release between external and native.
- Qualifying unqualified SQL parameter and SQL variable references to avoid adverse impact due to the different name resolution rules that apply between native and external.

Be aware that the processes to produce and maintain a callable native SQL procedure (dynamic DDL) are different than the processes used to produce and maintain a callable external SQL procedure (stored procedure DSNTPSMP or JCL). To get the most benefit from the conversion, devote some of the effort to becoming familiar with and comfortable using the lifecycle processes for native SQL procedures.

The technote “Converting an external SQL procedure to a native SQL procedure” can be of help. It is available from the Web site:

```
http://www-01.ibm.com/support/docview.wss?rs=64&context=SSEPEK&uid=swg21297948&loc=en_US&cs
=utf-8&lang=en
```

Tip: Consider making SQL Format SQLPL your default setting in SPUFI. This mode is suitable for all of SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, SPUFI retains SQL comments and terminates each line of an SQL statement with a line feed character (hex 25) before passing the statement to DB2. Lines that end with a split token are not terminated with a line feed character. Use this mode to obtain improved diagnostics and debugging of SQL procedural language.

There are some things we would like to remind you of when you migrate your external SQL stored procedures to native SQL procedures:

- You can only migrate your external SQL stored procedures when your DB2 9 for z/OS runs in new function mode.
- External stored procedures without FENCED or EXTERNAL keywords can be “converted” to native SQL procedures by redeploying the routine after dropping the old one. If you do not apply any other changes to the code, DB2 assigns a version of V1 to it. Refer to Chapter 15, “Native SQL procedures” on page 253 for more about versioning with native SQL procedures.
- In case of a fallback from NFM to either ENFM* or CM*, no changes are allowed to native SQL procedures.
- In case of a fallback from NFM to either ENFM* or CM*, your native SQL procedures (those that you have already created or migrated to) will continue to work.

15

Native SQL procedures

A new type of SQL procedure, the native SQL procedure, is introduced with DB2 9 for z/OS. In this chapter we focus on the installation characteristics and powerful application development capabilities of native SQL stored procedures. We highlight the advantages offered by native SQL stored procedures when compared to external SQL stored procedures. We create a native stored procedure, CALL it, add new versions, and deploy it to a different server. New language constructs and basic debugging functionality are described.

Note that the enhancements that V9 has provided to SQL procedures are all applicable only to the native SQL procedures, not to the external SQL procedures.

This chapter contains the following sections:

- Native vs. external SQL procedures
- Defining a native SQL procedure
- Versioning
- Execution of a native SQL procedure
- Deployment of a native SQL procedure to another server
- DB2/DSN/SQL command changes
- Error handling and debugging
- Migrating external to native SQL procedures

15.1 Native vs. external SQL procedures

External SQL procedures have been around since DB2 for z/OS Version 5. This was the only type of SQL procedures that was available for versions of DB2 prior to Version 9. Starting with DB2 9, there are two different ways for you to work with SQL procedures. We now distinguish between native and external SQL procedures.

With V9 new function mode, when you create a native SQL procedure, its procedural statements are converted to a native representation that is stored in the DB2 catalog and directory, as it is done with other SQL statements. The parameter list and procedure options are stored in the database catalog tables as in the prior releases. When you CALL a native SQL procedure, DB2 loads the native representation from the directory and the DB2 engine executes the procedure.

15.1.1 Differences between external and native SQL procedures

With versions prior to V9, when you create an SQL procedure, the SQL procedure requires SQL code and C code. The logic in your SQL procedure body has to be translated into C code, compiled, and finally exists as a compiled C program outside of your DB2 catalog. Starting with V9, we have external SQL stored procedures, which still require a C compiler and still exist as external load modules outside of DB2, and we also have native SQL procedures that do not require the use of a C program. No changes have been made to the existing (external) SQL procedures functions. No new data types are supported. However, the new DML statements are included.

The following table depicts the differences between native and external SQL procedures in the build process (preparation) as well as the execution phase. It furthermore shows the dramatic simplification introduced by the native SQL procedures.

Table 15-1 External vs. native SQL procedures

	Preparation phase	**Execution phase**
External SQL procedures *from V5 on*	Multiple steps necessary, requires C compiler.	The load module for the generated C program requires a WLM environment to run.
Native SQL procedures *starting with V9 NFM*	Single-step DDL.	Runs entirely within the DB2 engine.

There are other differences between the two types of SQL procedures regarding the creation, the execution environment, the SQL language, and the returned messages. We will talk about these topics later in this chapter.

15.1.2 Advantages of native SQL procedures

Using native SQL procedures bears many advantages over the external SQL procedure approach. Probably the most important aspect is a significant reduction in the total cost of ownership (TCO) when dealing with SQL procedures. The following list contains some further key differentiators that all basically contribute to the reduction in TCO and additionally show why the native SQL procedures should be favored over the existing (external) approach.

- **Simplified build process**
  You will not need a C or C++ compiler to create native SQL procedures. The multiple steps of setup and level of complexity in the build process that are required by an external SQL procedure, have been much reduced and simplified for a native SQL procedure. Enhanced productivity as well as usability are two main consequences of this.

- **Improved performance**

  Native SQL procedures are executed entirely in the DB2 engine, whereas external SQL procedures are executed in the WLM environment. The native SQL procedures are expected to outperform typical external SQL procedures.

- **Offloadable to zIIP**

  Native SQL procedures are eligible for offloading to a zIIP. If a native SQL procedure is called from a DRDA client using TCP/IP, then a portion of the SQL procedure processing is directed to a zIIP. If a native SQL procedure is called from a local application and the SQL procedure performs a parallel query, then a portion of the child tasks will be directed to a zIIP. There should not be any noticeable performance impact other than more CPUs are available to process the parallel requests.

- **Enhanced SQL support**

  Native SQL procedures offer enhanced support for the SQL Procedural Language. This implies new constructs like FOR loops, nested compound statements and more data types (e.g. BIGINT, BINARY, VARBINARY, DECFLOAT). A developer is thus capable of writing very complex SQL procedures.

- **Better family compatibility, portability, and standards compliance**

  Native SQL procedures support more SQL PL language, which is already available on other platforms such as LUW and i5/OS®. Therefore, these types of procedures are much more portable especially in a heterogeneous environment.

- **Application life cycle support for native SQL procedures**

  Native SQL procedures have been designed with the view of the application development lifecycle in mind. You can create a version of an SQL procedure, debug it, replace it or add a new version of the procedure, and finally deploy it into production. DB2 manages the various aspects of the application development lifecycle in a consistent and integrated manner providing enhanced security, including the source code for the native SQL procedures. Also native SQL procedures offer application and tools support like SPUFI, DSNTEP2 on the server side, and the Data Studio on the client side.

15.1.3 Environmental considerations

Both native and external SQL procedures can be called in V9 in New Function Mode. External SQL procedures continue to work in V9 in Compatibility Mode and New Function Mode, and can be created in V9 New Function Mode. Native SQL procedures can be created only starting in V9 New Function Mode.

In V9 there is a new catalog table, which is named SYSIBM.SYSENVIRONMENT, that stores environment information utilized for index on expressions as well as native SQL stored procedures. The environment information consists of a set of options and special registers. Column ENVID in SYSIBM.SYSENVIRONMENT is a unique value for each environment information recorded.

In order to combine the information in SYSIBM.SYSROUTINES with those additional ones stored in SYSIBM.SYSENVIRONMENT, a new column TEXT_ENVID is added to SYSIBM.SYSROUTINES to reference this environment. To give you a better understanding of which information DB2 stores in this new catalog table, you can refer to its structure and column description in Table 15-2.

Table 15-2 SYSIBM.SYSENVIRONMENT catalog table

Column name	Data type	Description
ENVID	INTEGER NOT NULL	Environment identifier
CURRENT_SCHEMA	VARCHAR(128) NOT NULL	The current schema
RELCREATED	CHAR(1) NOT NULL	The release when the environment information is created
PATHSCHEMAS	VARCHAR(2048) NOT NULL	The schema path
APPLICATION_ENCODING_CCSID	INTEGER NOT NULL	The CCSID of the application environment
ORIGINAL_ENCODING_CCSID	INTEGER NOT NULL	The original CCSID of the statement text string
DECIMAL_POINT	CHAR(1) NOT NULL	The decimal point indicator: C Comma P Period
MIN_DIVIDE_SCALE	CHAR(1) NOT NULL	The minimum divide scale: N The usual rules apply for decimal division in SQL Y Retain at least three digits to the right of the decimal point after any decimal division
STRING_DELIMITER	CHAR(1) NOT NULL	The string delimiter that is used in COBOL string constants: A Apostrophe(') Q Quote (“)
SQL_STRING_DELIMITER	CHAR(1) NOT NULL	The SQL string delimiter that is used in string constants: A Apostrophe(') Q Quote(“)
MIXED_DATA	CHAR(1) NOT NULL	Uses mixed DBCS data: N No mixed data Y Mixed data
DECIMAL_ARITHMETIC	CHAR(1) NOT NULL	The rules that are to be used for CURRENT PRECISION and when both operands in a decimal operation have a precision of 15 or less: 1 DEC15 specifies that the rules do not allow a precision greater than 15 digits 2 DEC31 specifies that the rules allow a precision of up to 31 digits
DATE_FORMAT	CHAR(1) NOT NULL	The date format: I ISO - yyyy-mm-dd J JIS - yyyy-mm-dd U USA - mm/dd/yyyy E EUR - dd.mm.yyyy L Locally defined by an installation exit routine

Column name	Data type	Description
TIME_FORMAT	CHAR(1) NOT NULL	The time format: I ISO - hh.mm.ss J JIS - hh.mm.ss U USA - hh:mm AM or hh:mm PM E EUR - hh.mm.ss L Locally defined by an installation exit routine
FLOAT_FORMAT	CHAR(1) NOT NULL	The floating point format. I IEEE floating point format S System /390 floating point format
HOST_LANGUAGE	CHAR(8) NOT NULL	The host language: * ASM * C * CPP * IBMCOB * PLI * FORTRAN
CHARSET	CHAR(1) NOT NULL	The character set: A Alphanumeric
FOLD	CHAR(1) NOT NULL	FOLD is only applicable when HOST_LANGUAGE is C or CPP. Otherwise FOLD is blank. N Lower case letters in SBCS ordinary identifiers are not folded to uppercase Y Lower case letters in SBCS ordinary identifiers are folded to uppercase blank Not applicable
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape.
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The rounding mode that is used when arithmetic and casting operations are performed on DECFLOAT data: C ROUND_CEILING D ROUND_DOWN F ROUND_FLOOR G ROUND_HALF_DOWN E ROUND_HALF_EVEN H ROUND_HALF_UP U ROUND_UP

Note: The same set of environment variables can be associated with multiple native SQL procedures or with multiple versions of a native SQL procedure.

Authorization

In order to be able to use the CREATE PROCEDURE SQL statement, you must have at least one of the following:

- The CREATEIN privilege on the schema that you are using
- SYSADM or SYSCTRL authority

If the authorization ID matches the schema name, it implicitly has the CREATEIN privilege on the schema. Refer to Chapter 7, “Security and authorization” on page 65 or *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854, for detailed information regarding authorization.

Operational hint

As we explained before, native stored procedures are not translated to C code, compiled, and bound later. Instead, native SQL stored procedures are compiled into runtime structures and bound at creation time. As a result the procedure’s non-logic part, as well as the logic part, is stored as a package in your DB2 directory.

On average, native SQL procedure packages are significantly larger than packages of stored procedures written in any other programming language or those from external SQL stored procedures. In these fenced or external procedures, the logic part corresponds to the load module that executes outside the engine.

Note: Your packages are loaded into the EDMPOOL at execution time. Since those packages might be significantly larger than other packages, you might run into EDMPOOL short of space conditions if you do not monitor its usage.

In DB2 z/OS Version 9, some parts of the EDMPOOL have been moved above the 2 GB bar, which are amongst others structures like the skeleton cursor and skeleton package tables. This resolves some size limitations from former versions and also significantly reduces the EDMPOOL size below the bar with an average estimation of 60%.

Changed messages from SQL procedures

DB2 V9 for z/OS issues different messages for the new native SQL procedures than it does for external SQL procedures. When an external SQL procedure CREATE statement is processed using DSNTEP2 or DSNTEP4 or SPUFI to register it in the catalog, SQL return codes are issued for errors. DSNHxxxx messages are only issued during the precompiler steps of the build process for an external SQL procedure. For native SQL procedures, DB2 issues SQL return codes.

15.2 Defining a native SQL procedure

In this chapter, a sample native SQL procedure is created by employing SQL processors like SPUFI and DSNTEP2. For completeness, the CREATE PROCEDURE syntax for native SQL procedures is included later in this chapter. Tooling like the Data Studio also supports SQL procedure development from the client side. For more information about development, deployment, and execution of native SQL procedures in Data Studio, refer to Chapter 27, “The IBM Data Studio” on page 643

15.2.1 Sample native SQL procedure

The SQL procedure MEDIAN_RESULT_SET in Example 15-1 basically calculates the median value of all the salary values in the table “staff”. It will also return a result set by querying the name, job, and salary columns from the table “staff” for which the salary contains the value higher than the calculated median value.

Example 15-1 Median_Result_Set SQL procedure

```
CREATE PROCEDURE MEDIAN_RESULT_SET (OUT medianSalary DECIMAL(7,2))
   VERSION MEDIAN_V1
   LANGUAGE SQL
```

```
  READS SQL DATA
   DYNAMIC RESULT SETS 1
 BEGIN
   DECLARE v_numRecords INTEGER DEFAULT 1;
   DECLARE v_counter INTEGER DEFAULT 0;
   DECLARE c1 CURSOR FOR
     SELECT salary FROM staff ORDER BY salary;
   DECLARE c2 CURSOR WITH RETURN FOR
     SELECT name, job, salary
        FROM staff
       WHERE salary > medianSalary
       ORDER BY salary;
   DECLARE EXIT HANDLER FOR NOT FOUND
     SET medianSalary = 0;
   SELECT COUNT(*) INTO v_numRecords FROM STAFF;
   OPEN c1;
   WHILE v_counter < (v_numRecords / 2 + 1) DO
     FETCH c1 INTO medianSalary;
     SET v_counter = v_counter + 1;
   END WHILE;
   CLOSE c1;
   OPEN c2;
 END
```

With the introduction of native SQL procedures, the semantics of the CREATE PROCEDURE statement for an SQL procedure have changed. Starting in Version 9.1 NFM, all SQL procedures that are created without the FENCED option or the EXTERNAL option in the CREATE PROCEDURE statement are native SQL procedures. In previous releases of DB2, if you did not specify either of these options, the procedures were created as external SQL procedures.

Note: The FENCED keyword is not really new on CREATE PROCEDURE SQL statements. It has been used as the default for SQL and other stored procedures. The meaning of FENCED is that the procedure runs in an external address space, which typically used the assigned WLM stored procedure address space. In addition to that, FENCED also specifies that the SQL procedure program is an MVS load module with an external name.

Starting with DB2 V9, if you specify FENCED or EXTERNAL together with LANGUAGE SQL on a CREATE PROCEDURE statement, you are asking for the *creation of an external SQL procedure*.

Using the VERSION option introduces application life cycle into the native SQL procedure. It creates a first initial version MEDIAN_V1 of the SQL procedure. Without specifying this keyword a default of V1 is assumed. In 15.3.7, “ALTER PROCEDURE syntax” on page 296 where the ALTER statement is explained, a new version is added for the same SQL procedure. To learn more about versioning refer also to 15.2.2, “CREATE PROCEDURE syntax” on page 265.

For native SQL procedures the LANGUAGE SQL keyword is now optional.

The DYNAMIC RESULT SETS 1 option on the CREATE PROCEDURE statement indicates that a result set may be returned to the caller of this procedure. DECLARE c2 CURSOR WITH RETURN FOR statement declares that the cursor c2 is associated with this result set. Finally, the OPEN c2 statement executed at the last statement in the body of this procedure will allow the caller to retrieve data from the result set associated with the cursor c2. Note that

instead of host variables as used in external procedures, SQL procedures use declared SQL variables and parameters which are used without colons. The condition handler is here used with a not found condition.

Using DB2-supplied dynamic SQL processing applications

Prior to V9, SQL procedures have been processed by either the precompiler or the SQL procedure processor DSNTPSMP stored procedure. With DB2 V9, you can easily use SPUFI, DSNTEP2, or DSNTEP4 to process your SQL procedures.

An SQL procedure is essentially a single SQL statement that consists of multiple SQL sub-statements, and may also include comments and white space. For readability, it is usually formatted on multiple lines. SPUFI, DSNTEP2, and DSNTEP4 accept multi-line input that lies within columns 1 and 72 of one or more 80-column input records. Such preprocessing of SQL into a linear, comment-free format is the standard, long-established behavior of SPUFI, DSNTEP2, and DSNTEP4, but this behavior complicates diagnostics and debugging of SQL procedure language, and effectively modifies the source by discarding SQL comments and comment blocks.

The SELECT statement augmented with some comments shown below:

```
SELECT name, job, salary, -- Result set to be returned
FROM staff
WHERE salary > medianSalary
ORDER BY salary;
```

is converted to:

```
SELECT name, job, salary FROM staff WHERE salary > medianSalary ORDER BY salary;
```

To enhance the usability of SPUFI, DSNTEP2, and DSNTEP4, three additional processing options affect how they preprocess SQL input before handing it off for PREPARE:

- SQL

  specifies that SQL statements are to be preprocessed as in earlier versions of DB2. SPUFI, DSNTEP2, and DSNTEP4 accept multi-line SQL statements but copy them seamlessly into an SQL buffer for PREPARE, effectively converting the multi-line input to a single line. In addition, SQL comments are removed.

- SQLCOMNT

  specifies that SQL-style comments are to be passed in the statement buffer for PREPARE instead of stripped out and discarded (the default behavior of SPUFI, DSNTEP2, and DSNTEP4). A LF character is appended after the detected SQL comment if none are found in the remaining input buffer (the portion that would otherwise be discarded). The purpose is to prevent loss of SQL comments, to assist with diagnostics and debugging, and to retain the format of the SQL procedure.

- SQLPL

  like SQLCOMNT, specifies that SQL-style comments are to be preserved. It also requests that a line formatting (LF) character is to be appended to each input line that ends without a token split. The purpose is to preserve the SQL comments and multi-line format of an SQL procedure in order to provide better diagnostics and debugging. It also helps retain the *format* of the SQL procedure when it is stored in the DB2 catalog.

To maintain multi-line granularity for debugging purposes and to preserve SQL comments, you can manually embed a line formatting character at the end of each line that ends with an undelimited token. Alternatively, you can use the new SQLPL option to have a line feed character x'25' added automatically.

With the option in effect, the above example is passed for PREPARE as:

```
SELECT name, job, salary, -- Result set to be returned<LF> FROM staff<LF> WHERE salary >
medianSalary<LF> ORDER BY salary;<LF>
```

Alternatively, you may wish to minimize modification of the input by preserving SQL comments but without adding line formatting characters. However, if SQL comments are to be preserved it is necessary to terminate them with line formatting characters because the parser otherwise cannot distinguish where they end, as the following example shows:

```
SELECT name, job, salary, -- Result set to be returned FROM staff WHERE salary >
medianSalary ORDER BY salary;
```

Use the SQLCOMNT option to specify that SQL comments are to be preserved, and terminated automatically by a line feed character if one is not provided in the source. Note that only SQL comments will be terminated. This is the difference between the SQLCOMNT option and the SQLPL option.

Note: When creating a SQL procedure, we recommend to use SQLPL as SQL FORMAT. This is especially true for scenarios where tools are employed, that extract the DDL from the DB2 catalog. This way we ensure that the multi-line formatting is preserved.

SQL FORMAT on SPUFI defaults panel

The SPUFI defaults panel DSNESP02 has a new option 6 (that is, SQL format). Figure 15-1 shows the new appearance of the SPUFI defaults panel.

```
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 DSNESP02                CURRENT SPUFI DEFAULTS           SSID: DB9B
 ===>

  1  SQL TERMINATOR .. ===> #          (SQL Statement Terminator)
  2  ISOLATION LEVEL   ===> CS         (RR=Repeatable Read, CS=Cursor Stability,
                                        UR=Uncommitted Read)
  3  MAX SELECT LINES  ===> 250        (Max lines to be return from SELECT)
  4  ALLOW SQL WARNINGS===> NO         (Continue fetching after sqlwarning)
  5  CHANGE PLAN NAMES ===> NO         (Change the plan names used by SPUFI)
  6  SQL FORMAT..... ===> SQLPL        (SQL, SQLCOMNT, or SQLPL)
Output data set characteristics:
  7  SPACE UNIT ..... ===> TRK        (TRK or CYL)
  8  PRIMARY SPACE ... ===> 5         (Primary space allocation 1-999)
  9  SECONDARY SPACE . ===> 6         (Secondary space allocation 0-999)
 10  RECORD LENGTH ... ===> 4092      (LRECL=Logical record length)
 11  BLOCK SIZE ..... ===> 4096       (Size of one block)
 12  RECORD FORMAT ... ===> VB        (RECFM=F, FB, FBA, V, VB, or VBA)
 13  DEVICE TYPE ..... ===> SYSDA     (Must be DASD unit name)
 Output format characteristics:
 14  MAX NUMERIC FIELD ===> 33        (Maximum width for numeric fields)
 15  MAX CHAR FIELD .. ===> 20        (Maximum width for character fields)
 16  COLUMN HEADING .. ===> NAMES     (NAMES, LABELS, ANY or BOTH)
 PRESS:  ENTER to process    END to exit             HELP for more information
```

Figure 15-1 SPUFI defaults panel - DSNESP02

As indicated in the previous section it is recommended to make use of the SQLPL format when developing SQL procedures. Also notice that the SQL TERMINATOR has been set to pound sign (#). This is required, as the SQL procedure contains embedded SQL statements that terminate with a default semicolon (;) and need to be distinguished from the embracing BEGIN and END.

In SPUFI just use the SQL statements shown in Example 15-2 to create the procedure. This is the DDL to create the required *staff* table with the salary column the median is created for.

Example 15-2 STAFF table DDL

```
CREATE TABLE STAFF
  ( NAME   VARCHAR(20),
    JOB    VARCHAR(20),
    SALARY INTEGER)#
```

Then some synthetic data is created by multiple INSERT statements on the *staff* table as shown in Example 15-3.

Example 15-3 Insert into STAFF table

```
INSERT INTO STAFF(NAME, JOB, SALARY)
   VALUES('Emp1', 'Manager', 45000)#
INSERT INTO STAFF(NAME, JOB, SALARY)
   VALUES('Emp2', 'Developer', 32000)#
INSERT INTO STAFF(NAME, JOB, SALARY)
   VALUES('Emp3', 'Tester', 30000)#
```

Finally the SQL procedure can be created by using the SQL statement in Example 15-4.

Example 15-4 Median_Result_Set SQL procedure

```
CREATE PROCEDURE MEDIAN_RESULT_SET (OUT medianSalary DECIMAL(7,2))
   VERSION MEDIAN_V1
   LANGUAGE SQL
   READS SQL DATA
    DYNAMIC RESULT SETS 1
  BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
      SELECT salary FROM staff ORDER BY salary;
    DECLARE c2 CURSOR WITH RETURN FOR
      SELECT name, job, salary
        FROM staff
       WHERE salary > medianSalary
       ORDER BY salary;
    DECLARE EXIT HANDLER FOR NOT FOUND
      SET medianSalary = 0;
    SELECT COUNT(*) INTO v_numRecords FROM STAFF;
    OPEN c1;
    WHILE v_counter < (v_numRecords / 2 + 1) DO
      FETCH c1 INTO medianSalary;
      SET v_counter = v_counter + 1;
    END WHILE;
    CLOSE c1;
    OPEN c2;
  END#
```

Calling this stored procedure returns the median salary in the output parameter as well as the staff that contain a higher salary than the median in the result set.

New parameter option: SQLFORMAT

When you execute DSNTEP2 or DSNTEP4, you can specify SQL, SQLCOMNT, or SQLPL behavior as the argument of a new application program parameter called SQLFORMAT.

Example 15-5 demonstrates the creation of the MEDIAN_RESULT_SET SQL procedure in DSNTEP2. This sample illustrates the use of SQLFORMAT parameter to select SQLPL behavior when calling DSNTEP2.

Example 15-5 SQLFORMAT parameter

```
//DSNTEP2  EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *
 DSN SYSTEM(DB9A)
 RUN  PROGRAM(DSNTEP2) PLAN(DSNTEP91) +
      LIB('DSN910.RUNLIB.LOAD') +
      PARMS('/SQLFORMAT(SQLPL),SQLTERM(#)')
 END
//*
//SYSIN    DD *
CREATE PROCEDURE MEDIAN_RESULT_SET (OUT MEDIANSALARY DECIMAL(7,2))
 VERSION MEDIAN_V1
 LANGUAGE SQL
 READS SQL DATA
 DYNAMIC RESULT SETS 1
 BEGIN
    DECLARE V_NUMRECORDS INTEGER DEFAULT 1;
    DECLARE V_COUNTER INTEGER DEFAULT 0;
    DECLARE C1 CURSOR FOR
      SELECT SALARY FROM STAFF ORDER BY SALARY;
    DECLARE C2 CURSOR WITH RETURN FOR
      SELECT NAME, JOB, SALARY
         FROM STAFF
         WHERE SALARY > MEDIANSALARY
         ORDER BY SALARY;
    DECLARE EXIT HANDLER FOR NOT FOUND
      SET MEDIANSALARY = 0;
    SELECT COUNT(*) INTO V_NUMRECORDS FROM STAFF;
    OPEN C1;
    WHILE V_COUNTER < (V_NUMRECORDS / 2 + 1) DO
      FETCH C1 INTO MEDIANSALARY;
      SET V_COUNTER = V_COUNTER + 1;
    END WHILE;
    CLOSE C1;
    OPEN C2;
  END#
/*
```

Note also the use of the SQLTERM parameter to change the SQL terminator from a semicolon to a pound sign (#). Individual statements within the SQL procedure body are terminated with a semicolon, so a different character is required to mark where the SQL procedure itself terminates.

Functional comments on SQL statements

Another possibility is to select the SQLPL behavior by using a so-called functional comment directly in the SQL stream being processed by SPUFI, DSNTEP2, or DSNTEP4. The name of the control statement is SQLFORMAT.

Example 15-6 creates the required table and insert statements with the SQL SQLFORMAT, and the default SQL termination character (;). Then it switches to a SQLPL behavior as well as a pound sign (#) for termination, to create the actual SQL procedure.

Example 15-6 Usage of functional comments

```
--#SET SQLFORMAT SQL
--#SET TERMINATOR ;

CREATE TABLE STAFF
  ( NAME   VARCHAR(20),
    JOB    VARCHAR(20),
    SALARY INTEGER);

INSERT INTO STAFF(NAME, JOB, SALARY)
   VALUES('Emp1', 'Manager', 45000);
INSERT INTO STAFF(NAME, JOB, SALARY)
   VALUES('Emp2', 'Developer', 32000);
INSERT INTO STAFF(NAME, JOB, SALARY)
   VALUES('Emp3', 'Tester', 30000);
-- ************************************************
--#SET SQLFORMAT SQLPL
--#SET TERMINATOR #
CREATE PROCEDURE MEDIAN_RESULT_SET (OUT medianSalary DECIMAL(7,2))
   VERSION MEDIAN_V1
   LANGUAGE SQL
   READS SQL DATA
    DYNAMIC RESULT SETS 1
  BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
      SELECT salary FROM staff ORDER BY salary;
    DECLARE c2 CURSOR WITH RETURN FOR
      SELECT name, job, salary
         FROM staff
        WHERE salary > medianSalary
        ORDER BY salary;
    DECLARE EXIT HANDLER FOR NOT FOUND
      SET medianSalary = 0;
    SELECT COUNT(*) INTO v_numRecords FROM STAFF;
    OPEN c1;
    WHILE v_counter < (v_numRecords / 2 + 1) DO
      FETCH c1 INTO medianSalary;
      SET v_counter = v_counter + 1;
    END WHILE;
    CLOSE c1;
    OPEN c2;
  END#
-- ************************************************
--#SET SQLFORMAT SQL
--#SET TERMINATOR ;
```

Note the use of the TERMINATOR control statement to change the SQL terminator character from the default semicolon to the pound sign, and then back to the semicolon.

15.2.2 CREATE PROCEDURE syntax

Figure 15-2, Figure 15-3 on page 266, and Figure 15-4 on page 267 show the complete CREATE PROCEDURE syntax for native SQL procedures.

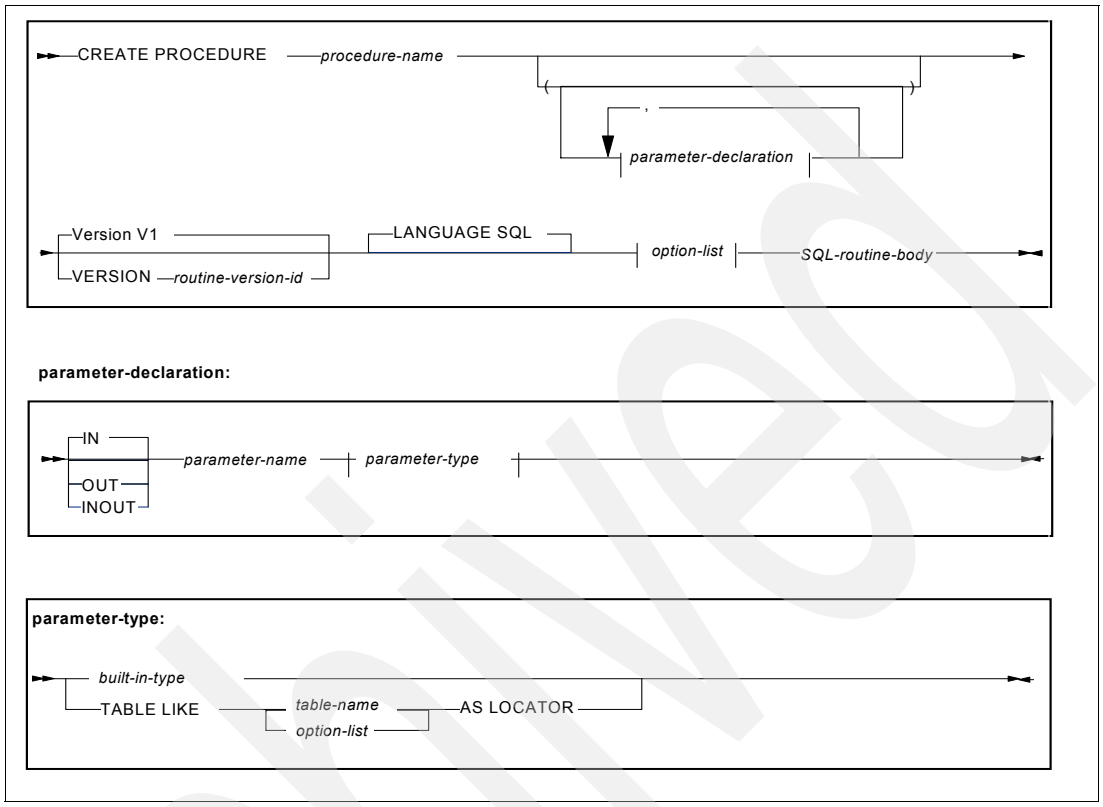

Figure 15-2 Start of CREATE PROCEDURE syntax

Figure 15-3 shows the continuation of the CREATE PROCEDURE statement with the built-in type.

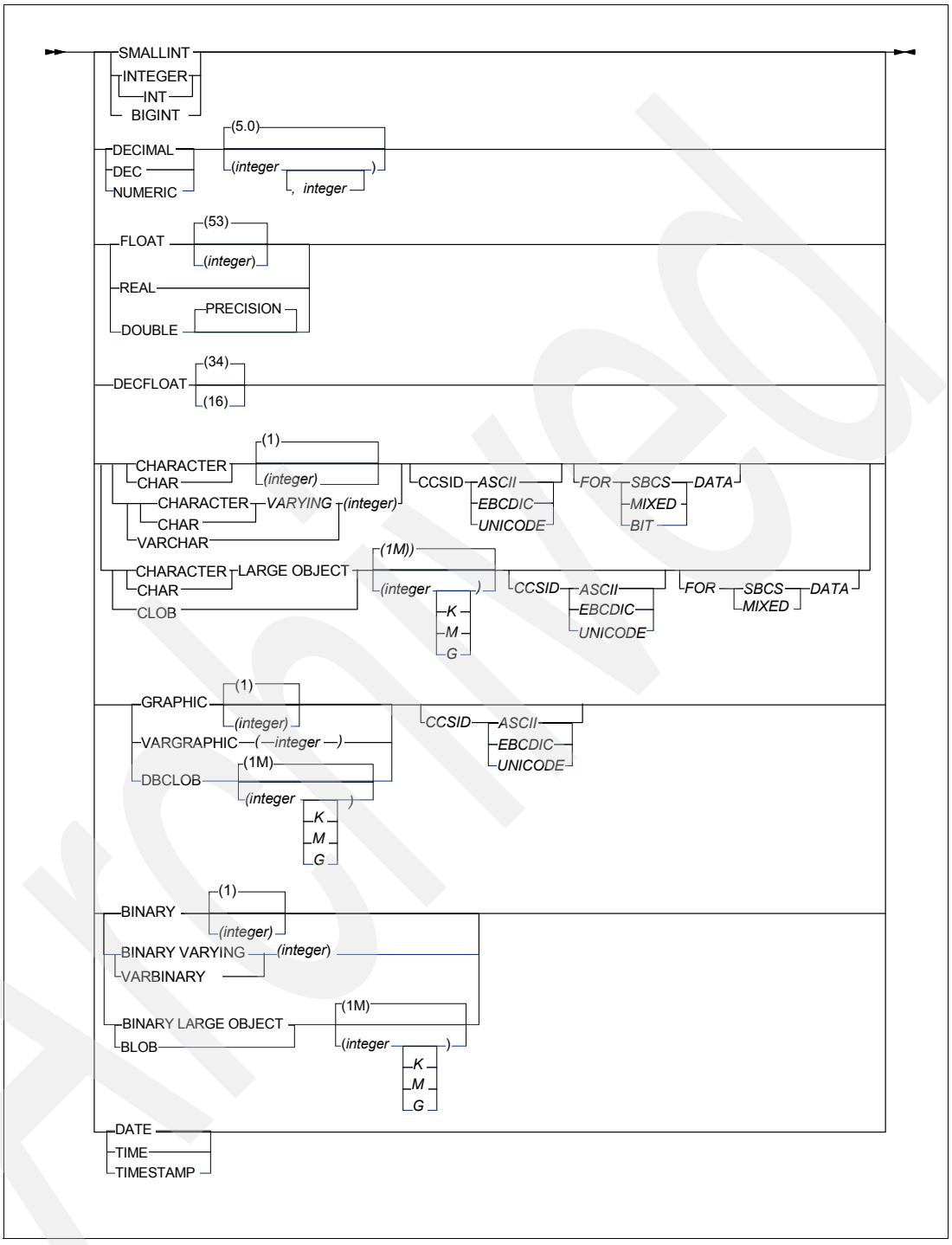

Figure 15-3 CREATE PROCEDURE built-in-type

Figure 15-4 shows the last part of the CREATE PROCEDURE syntax with the option list.

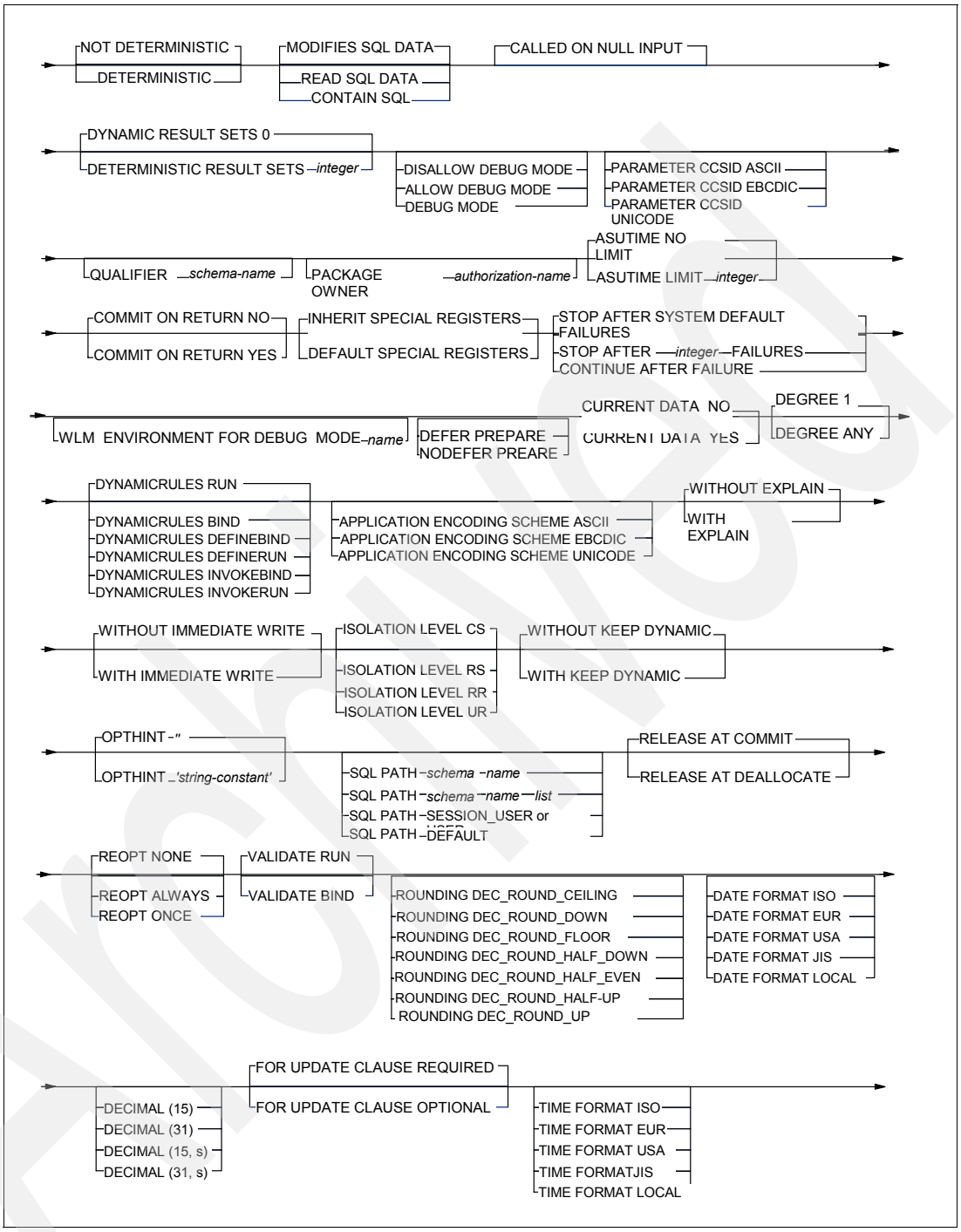

Figure 15-4 CREATE PROCEDURE option list

In comparison to what we had and still have for external SQL procedures, the most keywords were added to the part shown in Figure 15-4. In this section, we only cover those keywords that are new or have some changed rules. Additionally, most of the parameters are basically the same as those used in BIND and REBIND. Refer to either *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854, or *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844, for a more detailed description of those commonly used options.

LANGUAGE SQL

For native SQL procedures, keyword LANGUAGE SQL is now optional.

VERSION

This specifies the version identifier for the first version of the procedure that is to be generated. A routine-version-id can be up to 124 UTF-8 bytes. You can use an ALTER PROCEDURE statement with the ADD VERSION clause or the BIND DEPLOY command to create additional versions of the procedure. V1 is the default version identifier. Refer to 15.3.7, “ALTER PROCEDURE syntax” on page 296, to learn more about adding versions.

ALLOW/DISALLOW/DISABLE DEBUG MODE

With this option, you can specify whether the procedure can be run in debugging mode. The default is determined using the value of the CURRENT DEBUG MODE special register.

- ALLOW DEBUG MODE specifies that the procedure can be run in debugging mode.
- DISALLOW DEBUG MODE specifies that the procedure cannot be run in debugging mode. You can use an ALTER PROCEDURE statement to change this option to ALLOW DEBUG MODE for the initial version of the procedure.
- DISABLE DEBUG MODE specifies that the procedure can never be run in debugging mode. The procedure cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE once the procedure has been created or altered using DISABLE DEBUG MODE. To change this option, create a version of the procedure using the desired option and make that version the active version. Do not specify DISABLE DEBUG MODE if WLM ENVIRONMENT FOR DEBUG MODE is specified.

Important: If you use either ALLOW DEBUG MODE or DISALLOW DEBUG MODE, you must have specified a valid value for WLMENV in DSNTIJUZ. You should also make sure that your WLM environment is set up properly so that debugging really can be invoked. You have to provide a valid value for the WLMENV in the DSNZPARM even if you specify DISALLOW DEBUG MODE on your CREATE statement, because with ALTER PROCEDURE you have the chance to switch between ALLOW and DISALLOW later on. If you already know for sure at creation time that you are not going to use debugging at all, you should use the DISABLE DEBUG MODE option.

QUALIFIER schema-name

Use this keyword to specify the implicit qualifier that is used for unqualified names of tables, views, indexes, and aliases that are referenced in the procedure body. The default value is the same as the default schema.

PACKAGE OWNER authorization-name

This specifies the owner of the package that is associated with the procedure. The SQL authorization ID of the process is the default value. The owner must have the privileges that are required to execute the SQL statements that are contained in the routine body. The value of PACKAGE OWNER is subject to translation when sent to a remote system.

WLM ENVIRONMENT FOR DEBUG MODE name

This specifies the workload manager (WLM) application environment that is used by DB2 when you are debugging the procedure. The name of the WLM environment is an SQL identifier. If you do not specify WLM ENVIRONMENT FOR DEBUG MODE, DB2 uses the default WLM-established stored procedure address space specified at installation time on Routine parameters panel: DSNTIPX.

Note: As for every other stored procedure, you must have the appropriate authority for the WLM application environment to define a procedure that is to run in a specified WLM application environment.

Do not specify WLM ENVIRONMENT FOR DEBUG MODE when DISABLE DEBUG MODE is specified.

If WLM ENVIRONMENT is specified for native SQL procedures without the FOR DEBUG keywords, an error is issued. This means that if one of the forms of the WLM ENVIRONMENT clause supported for external SQL procedures is specified (WLM ENVIRONMENT name or WLM ENVIRONMENT (name,*)), then an error is issued. If WLM ENVIRONMENT is specified for a native SQL procedure, WLM ENVIRONMENT FOR DEBUG must be specified.

DEFER PREPARE or NODEFER PREPARE

This parameter specifies whether to defer preparation of dynamic SQL statements that refer to remote objects or to prepare them immediately. The default depends on the value in effect for the REOPT option. If REOPT NONE is in effect, the default is NODEFER PREPARE. Otherwise, the default is DEFER PREPARE.

CURRENT DATA

This specifies whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is in effect. CURRENTDATA also determines whether block fetch can be used for distributed, ambiguous cursors.

DEGREE

This specifies whether to attempt to run a query using parallel processing to maximize performance.

DYNAMICRULES

This specifies the values that apply, at runtime, for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements
- Whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements

In addition to the value of the DYNAMICRULES clause, the runtime environment of a native SQL procedure controls how dynamic SQL statements behave at runtime. The combination of the DYNAMICRULES value and the runtime environment determines the value for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The following values can be specified:

RUN This specifies that dynamic SQL statements are to be processed using run behavior. RUN is the default.

BIND This specifies that dynamic SQL statements are to be processed using bind behavior.

DEFINEBIND This specifies that dynamic SQL statements are to be processed using either define behavior or bind behavior.

DEFINERUN This specifies that dynamic SQL statements are to be processed using either define behavior or run behavior.

INVOKEBIND This specifies that dynamic SQL statements are to be processed using either invoke behavior or bind behavior.

INVOKERUN This specifies that dynamic SQL statements are to be processed using either invoke behavior or run behavior.

APPLICATION ENCODING SCHEME

This specifies the default encoding scheme for SQL variables in static SQL statements in the procedure body. The value is used for defining an SQL variable in a compound statement if the CCSID clause is not specified as part of the data type, and the PARAMETER CCSID routine option is not specified. This option also affects the content of the data that is returned by the SQL statement DESCRIBE. DB2 will return column names, label names, or both (if requested) in the specified application encoding scheme.

WITH EXPLAIN or WITHOUT EXPLAIN

This specifies whether information will be provided about how SQL statements in the procedure will execute.

WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE

This specifies whether immediate writes are to be done for updates that are made to group buffer pool dependent page sets or partitions. This option is only applicable for data sharing environments. The IMMEDWRITE subsystem parameter has no affect on this option.

ISOLATION LEVEL

This specifies how far to isolate the procedure from the effects of other running applications.

KEEP DYNAMIC

This specifies whether DB2 keeps dynamic SQL statements after commit points.

OPTHINT

This specifies whether query optimization hints are used for static SQL statements that are contained within the body of the procedure.

SQL PATH

This specifies the SQL path that DB2 uses to resolve unqualified user-defined distinct type, function, and procedure names in the procedure body. The default value is SYSIBM, SYSFUN, SYSPROC, and procedure-schema, where procedure-schema is the schema qualifier for the procedure that is the target of the statement.

RELEASE AT

This specifies when to release resources that the procedure uses: either at each commit point or when the procedure terminates.

REOPT

This specifies whether DB2 will determine the access path at runtime by using the values of SQL variables or SQL parameters, parameter makers, and special registers.

VALIDATE RUN or VALIDATE BIND

This specifies whether to recheck, at runtime, errors of the type OBJECT NOT FOUND and NOT AUTHORIZED that are found during bind or rebind. The option has no effect if all objects and needed privileges exist.

VALIDATE RUN This specifies that if needed objects or privileges do not exist when the CREATE PROCEDURE statement is processed, warning messages are returned, but the CREATE PROCEDURE statement succeeds.

The DB2 subsystem rechecks for the objects and privileges at runtime for those SQL statements that failed the checks during processing of the CREATE PROCEDURE statement. The authorization checks the use of the authorization ID of the owner of the procedure. VALIDATE RUN is the default.

VALIDATE BIND This specifies that if needed objects or privileges do not exist at the time the CREATE PROCEDURE statement is processed, an error is issued and the CREATE PROCEDURE statement fails.

ROUNDING

This specifies the desired rounding mode for manipulation of DECFLOAT data. The default value is taken from the DEFAULT DECIMAL FLOATING POINT ROUNDING MODE in DECP.

- DEC_ROUND_CEILING

  This specifies that numbers are rounded towards positive infinity.

- DEC_ROUND_DOWN

  This specifies that numbers are rounded towards 0 (truncation).

- DEC_ROUND_FLOOR

  This specifies that numbers are rounded towards negative infinity.

- DEC_ROUND_HALF_DOWN

  This specifies that numbers are rounded to nearest. If equidistant, round down.

- DEC_ROUND_HALF_EVEN

  This specifies that numbers are rounded to nearest. If equidistant, round so that the final digit is even.

DATE FORMAT

This specifies the date format for result values that are string representations of date or time values.

DECIMAL(15), DECIMAL(31), DECIMAL(15,s), or DECIMAL(31,s)

This specifies the maximum precision that is to be used for decimal arithmetic operations.

FOR UPDATE CLAUSE

This specifies whether the FOR UPDATE clause is required for a DECLARE CURSOR statement if the cursor is to be used to perform positioned updates.

TIME FORMAT

This specifies the time format for result values that are string representations of date or time values.

Compatibilities

For compatibility with previous versions of DB2, you can still specify the clauses listed below. DB2 will ignore those statements and issue SQLCODE +434 as a warning.

- LANGUAGE SQL
- STAY RESIDENT
- PROGRAM TYPE
- RUN OPTIONS
- NO DBINFO
- COLLID or NO COLLID
- SECURITY

- PARAMETER STYLE GENERAL WITH NULLS

Note: If you specify WLM ENVIRONMENT *wlmname* without the FOR DEBUG keyword, DB2 issues an error SQLCODE.

In addition to the above-mentioned keywords that DB2 ignores, you can continue to use the following:

- For the DYNAMIC RESULT SETS keyword, you can also:
  - Omit the DYNAMIC keyword.
  - Use SET instead of SETS.
- For DETERMINISTIC you can use NOT VARIANT instead.
- For NOT DETERMINISTIC you can use VARIANT instead.
- For CALLED ON NULL INPUT, NULL CALL can be used.

15.2.3 New features

Native SQL procedures come with enhanced support for the SQL Procedural Language, which allows the user to write complex SQL procedures. In this section new supported language constructs are presented and enriched with code samples.

FOR SQL control statement

Native SQL procedures now support the FOR loop construct. In such a FOR loop, you embed a SELECT statement. DB2 then executes the SELECT statement and creates a result set for it. A cursor iterating over the result set is associated with the FOR loop and allows you to access all rows in the result set sequentially. The body of the FOR loop control statement is executed for every row contained in the result set. The values in the columns of the current row are accessible in the body of the loop. The for-loop-name can and should be used to qualify the column names in order to distinguish them from any other variables that may be available in the current execution scope. Generally, when the FOR control statement is used a user has to ensure that the queried table exists at creation time.

To illustrate the usage of a FOR loop construct consider the SQL statement in Example 15-7, which calculates the sum of the salaries of all employees in a table.

Example 15-7 total staff salary

```
SELECT SUM(salary) FROM staff;
```

Utilizing the FOR SQL statement, an equivalent result can be obtained with the native SQL procedure in Example 15-8.

Example 15-8 Native SQL procedure: CALC_SALARY

```
CREATE PROCEDURE CALC_SALARY(OUT SUM INTEGER)
 VERSION V1
 LANGUAGE SQL
 READS SQL DATA
  BEGIN
    SET SUM = 0;
    FOR V1 AS
      C1 CURSOR FOR
      SELECT SALARY FROM STAFF
      DO SET SUM = SUM + V1.SALARY;
    END FOR;
```

```
END#
```

Note: You always have to be aware that rewriting statements might result in a performance decrease because it could invalidate some internal processing capabilities.

Example 15-9 is a bit more complex. It shows a native SQL procedure that receives an owner authorization ID as input parameter. A query on the DB2 catalog table SYSIBM.SYSPACKAGE then retrieves all package names that are associated to this ID. A FOR loop then iterates over all these qualifying packages and tries to rebind them. For the actual rebind the DB2 supplied stored procedure ADMIN_COMMAND_DSN is called from within the FOR loop. The message parameter that is populated by ADMIN_COMMAND_DSN is finally passed back to the caller as output parameter of the REBIND_PACKAGES procedure.

Example 15-9 Native SQL procedure: REBIND_PACKAGES

```
CREATE PROCEDURE SYSPROC.REBIND_PACKAGES (
  IN PACKAGE_OWNER VARCHAR(128),
  OUT MSG_OUT VARCHAR(32704) )
  VERSION V1
  LANGUAGE SQL
  MODIFIES SQL DATA
  BEGIN
    DECLARE MSG VARCHAR(1331) DEFAULT '';
    SET MSG_OUT = '';
    FOR V1 AS
      C1 CURSOR FOR
      SELECT COLLID, NAME, VERSION
       FROM SYSIBM.SYSPACKAGE WHERE OWNER = PACKAGE_OWNER
      DO
        SET MSG = '';
        CALL SYSPROC.ADMIN_COMMAND_DSN(
         'REBIND PACKAGE(' ||
              V1.COLLID    || '.'  ||
              V1.NAME      || '.(' ||
              V1.VERSION   || '))', MSG);
        SET MSG_OUT = MSG_OUT || MSG;
   END FOR;
END#
```

Example 15-10 shows another native SQL procedure that implements the FOR loop construct. The basic idea here is that the procedure checks whether all values in a column of type INTEGER feature the same value. If this is true, then the value is returned in the output parameter; otherwise, NULL is returned.

Example 15-10 Native SQL procedure: NODIFF

```
CREATE PROCEDURE NODIFF (
  OUT RESULT INTEGER )
  VERSION V1
  LANGUAGE SQL
  MODIFIES SQL DATA
  BEGIN
    DECLARE PREVVALUE INTEGER;
    SET PREVVALUE = NULL;
```

```
    F1: FOR V1 AS
      C1 CURSOR FOR
      SELECT DEPNO FROM STAFF
      DO
        IF PREVVALUE IS NULL THEN
          SET PREVVALUE = V1.DEPNO;
          SET RESULT = V1.DEPNO;
        END IF;
        IF PREVVALUE <> V1.DEPNO THEN
          SET RESULT = NULL;
          LEAVE F1;
        END IF;
    END FOR;
END#
```

For completeness, the FOR statement syntax diagram is attached in Figure 15-5. For detailed parameter explanations, refer to *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

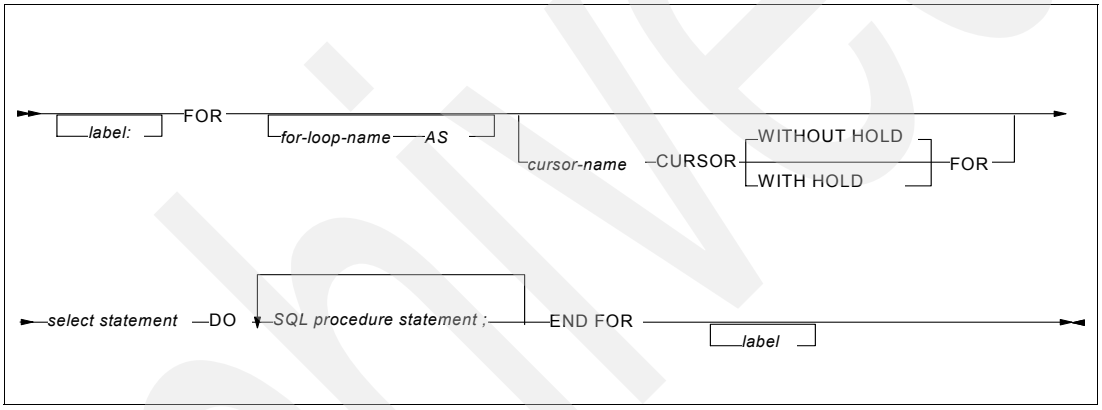

Figure 15-5 FOR statement syntax

Extended GOTO

Native SQL procedures provide support for an extended version of the GOTO statement. This statement now allows for branching out (upwards) of the current compound statement to different levels within the same scope. There are two major restrictions on the target label that you need to be aware of:

- If the GOTO statement is in a condition handler, the target label must be defined in that condition handler as well.
- If the GOTO statement is not in a condition handler, the target label must not be defined in a condition handler.

Example 15-11 depicts a case where the GOTO statement branches out of the current compound statement to a label defined at a higher level.

Example 15-11 GOTO sample

```
CREATE PROCEDURE GOTO()
P1: BEGIN
  DECLARE I, A INTEGER;
  SET I = 1;

  LAB1: SET A = 1;
  BEGIN
```

```
    LAB2: SET A = 2;
    BEGIN
      SET I = I + 1;
      IF I < 3 THEN GOTO LAB1;
      END IF;
    END;
  END;
END P1#
```

Note: When a GOTO statement branches out of a compound statement, all open cursors that are declared in the compound statement that contains the GOTO are closed. The only exception are cursors that return a result set. The same holds for nested compound statements.

Example 15-12 declares cursors at multiple levels. The GOTO statement branches out of the compound statement labelled L1. Therefore, all cursors that are defined within this compound statement (C1, C2, and C3) are closed.

Example 15-12 Another GOTO sample

```
CREATE PROCEDURE GOTO()
L0: BEGIN
  DECLARE CURSOR C0 ...
  ...
  TARGETLABEL: ...
  ...
  L1: BEGIN
    DECLARE CURSOR C1 ...
    ...
    L2: BEGIN
      DECLARE CURSOR C2...
      ...
      GOTO TARGETLABEL;
      ...
      L3: BEGIN
        DECLARE CURSOR C3 ...
        ...
      END L3;
    END L2;
  END L1;
END L0#
```

Tip: Using GOTO is usually considered a bad programming practice and should be avoided. In general, it decreases the readability of code, and makes it harder to maintain.

Nested compound statements in SQL procedures

With the introduction of native SQL stored procedures on DB2 for z/OS, nested compound statements are supported. A compound statement groups other statements together into an executable block (delimited by BEGIN and END). SQL variables can be declared within a compound statement.

Up to DB2 V8, the body of an SQL procedure could contain only a single compound statement, which could contain other SQL statements, except for another compound statement, or a single SQL procedure statement other than the compound statement. Thus,

you could not nest compound statements in an SQL procedure. Additionally, this meant that a condition handler could not contain a compound statement.

Starting with DB2 V9, you can now use:

- A compound statement within a condition handler
- Nested compound statements to define different scopes for SQL variables, cursors, condition names, and condition handlers

A compound statement is easily recognized as starting and ending with the keywords BEGIN and END. You can provide a label on the BEGIN statement to identify the code block.

You can use this label to qualify names of SQL variables that are defined in a compound statement, while cursor names must be unique within a procedure. These labels are very useful with nested compound statements. It is possible that you have an SQL procedure that contains both nested compound statements and compound statements that are at the same level. Figure 15-6 illustrates a possible outline of an SQL procedure that consists of the compound statement OUTERMOST, which contains two other compound statements, INNER1 and INNER2. Additionally, the INNER1 compound statement contains another compound statement, INNERMOST.

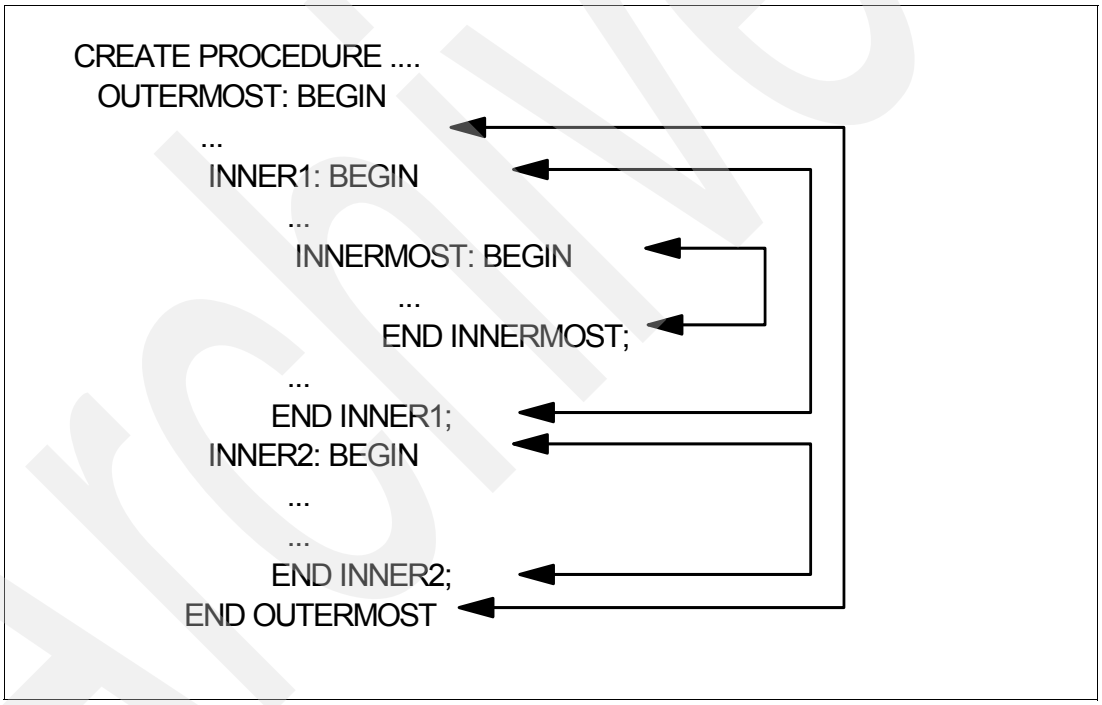

Figure 15-6 Nested compound statement

Using statement labels with nested compound statements

You can use nested compound statements within an SQL procedure to define the scope of SQL variable declarations, condition names, and condition handlers. Each compound statement has its own defined scope, and can be defined to have a label. It is possible to define multiple labels, each having the same name, within a procedure as long as they are defined in different scopes. A label name must be unique within the compound statement for which it is defined, including any labels defined within compound statements that are nested within that compound statement. That is, a label specified on a compound statement must be unique from other labels within the same scope. However, compound statements that are not

nested can have the same name. Also, a label must not be the same as the name of the procedure.

You can reference a label in another statement such as the GOTO, LEAVE, and ITERATE SQL Procedure language control statements. A label can only be referenced in the compound statement in which it is defined, including any compound statements that are nested in that compound statement.

Example 15-13 shows a very simple native SQL procedure, SCOPELAB, that implements labels in nested compound statements. The procedure features an integer variable named FLAG as INOUT parameter. The outermost compound statement is labelled OUTER1 and contains two further compound statement labels INNER1 and INNER 2 which are on the same nesting level. The SQL comments indicate the flow of the logic in the procedure which is determined by the input value of the FLAG parameter.

Example 15-13 Scoping label names (1)

```
CREATE PROCEDURE SCOPELAB(INOUT FLAG INTEGER)
  VERSION V1
  LANGUAGE SQL
  READS SQL DATA
   -- OUTERMOST COMPOUND STATEMENT
   OUTER1: BEGIN                  -- (1)
      -- INNER COMPOUND STATEMENT WITH LABEL INNER1
      INNER1: BEGIN               -- (2)
         IF FLAG = 1 THEN
            ABC: LEAVE INNER1;    -- (3)
         ELSEIF FLAG = 2 THEN
            XYZ: LEAVE OUTER1;   -- (4)
         END IF;
      END INNER1;
      -- END OF INNER COMPOUND STATEMENT INNER1

      -- INNER COMPOUND STATEMENT WITH LABEL INNER2
      INNER2: BEGIN               -- (6)
         XYZ: SET FLAG = FLAG + 4;
      END INNER2;
      -- END OF INNER COMPOUND STATEMENT INNER2
   END OUTER1#
```

If a FLAG features a value of 1, the code traverses to the statement labelled ABC, which then makes the program flow exit the INNER1 label scope. Next the INNER2 scope is entered, and the command with label XYZ executed. In this case, the procedure returns a value of 5 to the caller. In case the SQL procedure is invoked with a value of 2 in the FLAG parameter, label XYZ in the INNER1 label scope, jumps out of the OUTER1 scope which at the same time ends the procedure. A value of 2 is returned here.

Notice that the XYZ label is defined twice. This is valid, because both are in different scopes on the same nesting level.

Figure 15-7 illustrates the use of statement labels and their scope. As you can see, a reference to OUTER 1 from inside the INNER1 compound statement is okay, because label OUTER1 is on a higher level than INNER1. In contrast to that, the statement of label NOWAY is not allowed and leads to an error, because the target label INNER2 is not within the scope of the LEAVE statement.

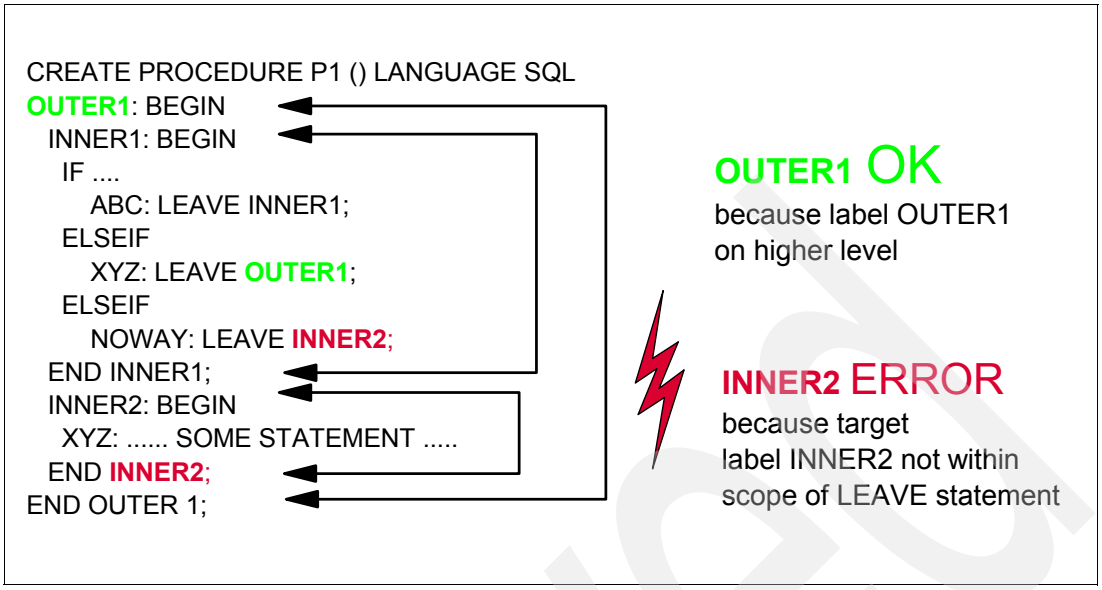

Figure 15-7 Scoping label names (2)

Let us have a look at more examples to get used to what is allowed and what is not allowed with regards to usage of labels in nested compound statements. See Figure 15-8.

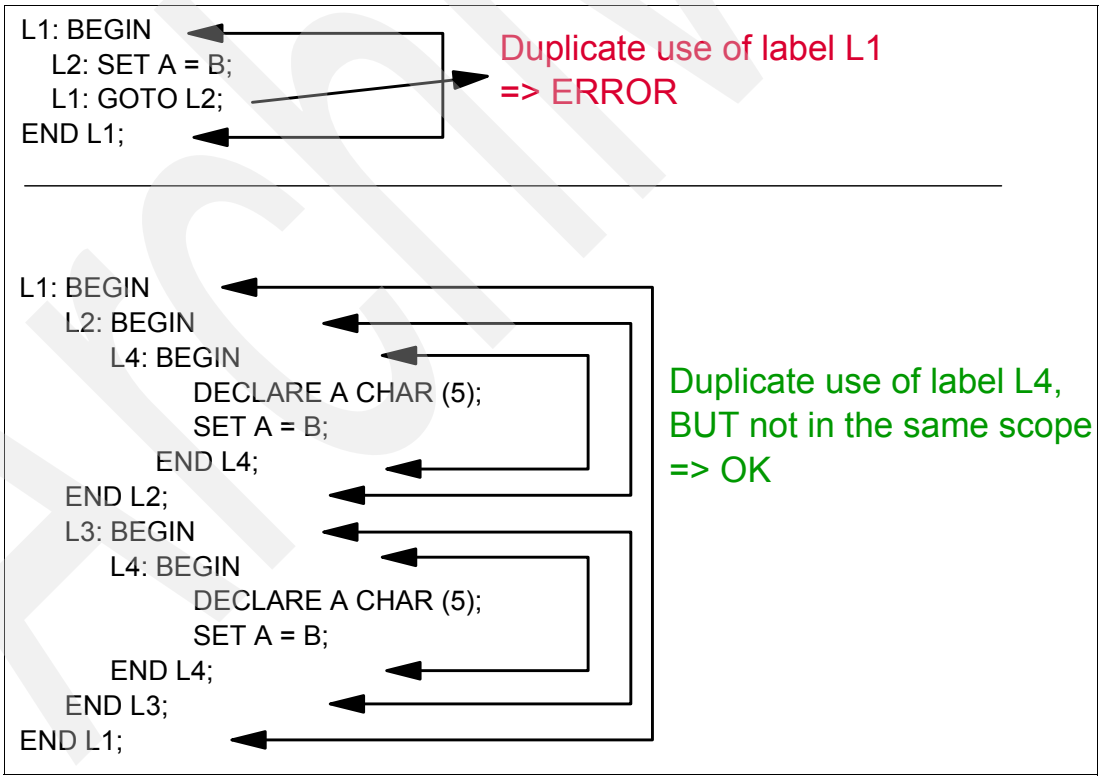

Figure 15-8 Correct and incorrect label usage

In the upper part of Figure 15-8, the use of label L1 is invalid, because it is in the same scope as the outer label L1. The correct usage of nested labels is shown in the lower part of Figure 15-8. Here we used label L4 twice as well, but since they are not in the same scope, you do not receive an error in this case.

In general, labels for compound statements are useful to avoid conflicts with column names in SQL statements. A developer should always use the qualified variable name, as shown in Example 15-14. Here, using the unqualified name SALARY would cause an ambiguity problem, which could lead to undesired side effects when calling the procedure.

Example 15-14 Scoping label names (3)

```
CREATE PROCEDURE SCOPELA2(OUT RES VARCHAR(20))
  VERSION V1
  LANGUAGE SQL
  READS SQL DATA
   OUTER1: BEGIN
     DECLARE SALARY INTEGER DEFAULT 32000;
     DECLARE C1 CURSOR FOR
       SELECT NAME
       FROM STAFF AS T
       WHERE T.SALARY = OUTER1.SALARY;
     OPEN C1;
     FETCH C1 INTO RES;
     CLOSE C1;
...
END#
```

Scoping SQL variable declarations

One more situation in which you can use nested compound statement is the definition of SQL variable names within an SQL stored procedure. An SQL variable is declared within a compound statement. With the use of nested compound statements, it is possible for you to define multiple SQL variables with the same name within an SQL stored procedure.

When you use non-unique variable names in an SQL stored procedure, each SQL variable must be declared within a different scope. An SQL variable name must be unique in the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement. Following this, an SQL variable can only be referenced in the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to an unqualified SQL variable name, the variable can be declared within the compound statement that contains the reference, or within a compound statement in which that compound statement is nested. In this case, DB2 uses the declaration of a variable by that name in the innermost compound statement.

If multiple variables with the same name exist in the stored procedure and there is an unqualified reference to the variable name, the name may not be interpreted as intended, because it resolves to the most local definition. For this reason we recommend that you always qualify the SQL variable with the name as the label from the compound statement in which it was declared.

Example 15-15 shows a stored procedure that implements some techniques for scoping variable declarations. It basically returns a value in the FLAG parameter, which is dependent on its input value. In case the procedure is invoked with FLAG = 1, a value of 300 is returned. For FLAG = 2, NULL is returned, and for FLAG = 3, a value of 100 is returned to the caller. The outermost block of the procedure is labelled OUTER1. It contains two further nested compound blocks INNER1 and INNER2 which are on the same nesting level.

Example 15-15 Scoping variable declarations

```
CREATE PROCEDURE SCOPEVAR(INOUT FLAG INTEGER)
```

```
VERSION V1
LANGUAGE SQL
READS SQL DATA
   -- OUTERMOST COMPOUND STATEMENT
   OUTER1: BEGIN
      DECLARE A INTEGER DEFAULT  50;

      -- INNER COMPOUND STATEMENT WITH LABEL INNER1
      INNER1: BEGIN
         DECLARE A INTEGER DEFAULT NULL;
         DECLARE W INTEGER DEFAULT NULL;
         SET A = A + OUTER1.A;       -- NULL + 50 = NULL
         SET OUTER1.A = 100;
         SET INNER1.A = 200;
         IF FLAG = 1 THEN
            SET FLAG = A + OUTER1.A; -- FLAG = 300
         END IF;

         -- SET INNER2.A = 300;       -- ERROR
      END INNER1;
      -- END OF INNER COMPOUND STATEMENT INNER1 -------

      INNER2: BEGIN
         DECLARE A INTEGER DEFAULT NULL;
         DECLARE Z INTEGER DEFAULT NULL;
         SET A = A + OUTER1.A;       -- NULL + 100 = NULL
         IF FLAG = 2 THEN
            SET FLAG = A;            -- FLAG = NULL
         END IF;
      END INNER2;
      -- END OF INNER COMPOUND STATEMENT INNER2 -------

      -- SET INNER1.A = 200;          -- ERROR

      SET OUTER1.A = 100;

      IF FLAG = 3 THEN
         SET FLAG = A;               -- FLAG = 100
         -- A REFERENCES HERE THE OUTER1.A
      END IF;
   END OUTER1#
```

The variable A is declared in the OUTER1 block. In the nested INNER1 compound statement, another variable A is declared. Both variables A are visible and accessible for statements within the INNER1 compound statement, and can be distinguished by using the respective labels as prefix. If no label prefix is specified, the declaration of the innermost nesting is used. For example, the unqualified variable A in the following statement in the INNER1 block,

```
SET A = A + OUTER1.A
```

is equivalent to specifying INNER1.A. The latter should be preferred because this improves the readability of the code. As indicated with a comment, the following statement would cause an error when defined in the INNER1 block:

```
SET INNER2.A = 300
```

This is because INNER1 and INNER2 are at the same nesting level. Therefore a local variable defined in the INNER2 scope is not visible from within the INNER1 scope and thus cannot be referred to. However, the variable A in the OUTER1 scope is visible to both compound statements INNER1 and INNER2 and hence can be accessed.

As indicated in the comment, the following statement would also result in an error.

```
SET INNER1.A = 200;
```

This is because a variable from within the scope of INNER1 is requested, which is no longer valid.

The following unqualified statement to a variable A refers to its declaration in the OUTER1 scope:

```
SET FLAG = A;
```

Here both inner compound statement scopes INNER1 and INNER2 are no longer active.

Note: DECLARE statements always have to appear directly after the BEGIN statement. Therefore, compound statements are very useful, especially in large SQL procedures, to bring variable declarations close to their usage. This means that you can avoid declaring all your variables up front in the procedure.

Scoping cursor definitions

Nested compound statements can also be used in an SQL procedure to define the scope of a cursor declaration. You can only reference a cursor name in the compound statement in which it is declared, including any compound statements that are nested within that compound statement. However, you cannot reference a cursor that is defined at a lower level from a higher level in the SQL procedure.

A cursor name remains a single part name as in previous releases. That means a cursor name must be unique within the stored procedure, even when compound statements are used.

However, any cursor that is declared in an SQL procedure as a result set cursor (that is, the WITH RETURN clause was specified as part of the cursor declaration) can be referenced by the calling application. This is true even for the case where the cursor is not declared in the outermost compound block of the procedure.

Example 15-16 Scoping cursor definitions (1)

```
CREATE PROCEDURE SCOPECUR(OUT P1 VARCHAR(128) CCSID EBCDIC)
  VERSION V1
  LANGUAGE SQL
  READS SQL DATA
  DYNAMIC RESULT SETS 1
    OUTER1: BEGIN
      DECLARE I VARCHAR(128) FOR SBCS DATA;
      DECLARE J VARCHAR(128) FOR SBCS DATA;
      DECLARE X CURSOR WITH RETURN FOR       -- DECLARATION FOR X
         SELECT OWNER FROM SYSIBM.SYSPACKAGE
          WHERE NAME = 'DSNUTILS';

         INNER1: BEGIN
            DECLARE Y CURSOR WITH RETURN FOR -- INNER DECLARATION FOR Y
```

```
            SELECT QUALIFIER FROM SYSIBM.SYSPACKAGE
            WHERE NAME = 'DSNUTILU';

          OPEN X;                          -- REFERENCES X IN OUTER1
          FETCH X INTO I;
          CLOSE X;
          -- OPEN X AGAIN FOR CALLER
          OPEN X;
        END INNER1;
      SET P1 = I;

      -- FOLLOWING ACCESS TO CURSOR Y RESULTS IN AN ERROR
      --OPEN Y;                 -- REFERENCES Y DECLARED IN INNER1 SCOPE
      --FETCH Y INTO J;
      --SET P2 = J;
      --CLOSE Y;
      --OPEN Y;                 -- OPEN X AGAIN FOR CALLER

  END OUTER1#
```

Example 15-16 defines two cursors, X and Y. X is accessible and visible in the OUTER1 and INNER1 block, whereas Y is only valid within the INNER1 block. Cursor X is opened and rows are fetched from it inside of the INNER1 block. Accessing Y from within the OUTER1 block would result in an error as indicated by the comments. Notice also that the calling application can access the result set associated with X, although the OPEN statement is not performed in the outermost compound statement.

The stored procedure shown in Example 15-17 deletes certain database objects that are associated with an ownerid specified in one of the input parameters. The object types to be deleted are either triggers, sequences, or distinct types. The procedure keeps track of the successfully dropped objects by inserting the respective names in a table MYRESULT.

Several nested compound statements are employed that allow for structuring the procedure code. Notice that the cursor is not defined in the outermost block, but closer to its utilization in the GET_OBJ block. As shown in the sample, it is possible to first construct a SELECT statement as a string, prepare it, and later on declare and open the cursor on it. Since DECLARE statements always have to appear at the beginning of a block, the nested compound statement GET_OBJ has been injected.

Example 15-17 Scoping cursor definitions (2)

```
  CREATE PROCEDURE SCOPECUR2 (IN  OBJOWNER  VARCHAR(128),
                              IN  TYPE     CHAR(1),
                              OUT RES      VARCHAR(128))

    VERSION V1
    LANGUAGE SQL
    MODIFIES SQL DATA
     MAIN: BEGIN
       DECLARE SELSTMT  VARCHAR(256);
       DECLARE TABNAME  VARCHAR(128);
       DECLARE OBJTYPE  VARCHAR(32);
       DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
       DECLARE SQLCODE  INTEGER DEFAULT 0;

      -- SET THE TABLENAME ACCORDING TO TYPE PARAMETER
      IF (TYPE = 'S') THEN
```

```
    SET TABNAME = 'SYSIBM.SYSSEQUENCES';
    SET OBJTYPE = 'SEQUENCE';
  ELSEIF (TYPE = 'T') THEN
    SET TABNAME = 'SYSIBM.SYSTRIGGERS';
    SET OBJTYPE = 'TRIGGER';
  ELSEIF (TYPE = 'D') THEN
    SET TABNAME = 'SYSIBM.SYSDATATYPES';
    SET OBJTYPE = 'TYPE';
  END IF;

  -- INIT SQL SELECT STATEMENT
  SET SELSTMT = 'SELECT T.SCHEMA, T.NAME ' ||
                'FROM '  || TABNAME  || ' AS T ' ||
                'WHERE T.OWNER = ''' || OBJOWNER || '''';

  PREPARE SQLSTMT FROM SELSTMT;

  GET_OBJ: BEGIN
    DECLARE OBJNAME   VARCHAR(128);
    DECLARE OBJSCHEMA VARCHAR(128);
    DECLARE C1 CURSOR FOR SQLSTMT;

    OPEN C1;
    FETCH_LOOP:
    LOOP
      FETCH C1 INTO OBJSCHEMA, OBJNAME;
      IF (SQLCODE <> 0) THEN
        -- NO MORE ELEMENT FOUND EXIT LOOP
        LEAVE FETCH_LOOP;
      END IF;

      DRP_OBJ:
      BEGIN
        -- THIS SECTION DROPS THE ASSOCIATED OBJECT
        DECLARE DRPSTMT VARCHAR(256);
          SET DRPSTMT = 'DROP ' || OBJTYPE || ' ' ||
                        OBJSCHEMA || '.' || OBJNAME;
          EXECUTE IMMEDIATE DRPSTMT;
          INSERT INTO MYRESULT
          VALUES ('DROPPED '||OBJSCHEMA||'.'||OBJNAME);
      END DRP_OBJ;

    END LOOP FETCH_LOOP;
    CLOSE C1;

  END GET_OBJ;
  SET RES = 'Objects successfully removed';
END MAIN!
```

Scoping condition names

Nested compound statements can be used in an SQL procedure to define the scope of SQL condition names. A condition is declared within a compound statement. With the use of nested compound statements, it is possible to define multiple conditions with the same name in an SQL procedure. When non-unique condition names exist in an SQL procedure, each condition must be declared within a different scope. A condition name must be unique in the compound statement in which it is declared, excluding any declarations in compound statements that are nested in that compound statement. You can reference a condition name in the declaration of a condition handler, or in a RESIGNAL or SIGNAL statement, but note that a condition name can only be referenced in the compound statement in which it is

declared, including any compound statements that are nested within that compound statement.

As for cursor names, a condition name remains a single part name as in previous releases. When multiple conditions are defined with the same name there is no way to explicitly refer to the condition that is not the most local in scope. Any reference to a non-unique condition name is an implicit reference to the innermost declaration of a condition of that name.

Example 15-18 shows only a theoretical implementation of non-unique condition names in an SQL procedure.

Example 15-18 Scoping condition names

```
CREATE PROCEDURE SCOPECND ()
 OUTER1: BEGIN
    DECLARE ABC CONDITION FOR '02000';
    DECLARE XYZ CONDITION FOR '0100E';
       INNER1: BEGIN
          DECLARE ABC CONDITION FOR '08001';
          SIGNAL ABC;
          --   (1) SIGNAL ABC IS INTERPRETED AS REFERRING
          --   TO THE CONDITION NAMED ABC THAT IS
          --   DECLARED IN THE INNERMOST BLOCK.
          --   (2) IF THE STATEMENT WERE CHANGED TO BE
          --   SIGNAL XYZ, IT WOULD BE INTERPRETED AS
          --   REFERRING TO THE CONDITION NAMED XYZ
          --   THAT WAS DECLARED IN THE OUTERMOST BLOCK.
       END INNER1;

       SIGNAL ABC;
       --    (3) SIGNAL ABC HERE IS INTERPRETED AS REFERRING
       --    TO THE CONDITION NAMED ABC THAT IS
       --    DECLARED IN THE OUTERMOST BLOCK.
 END OUTER1#
```

Using a compound statement in a condition handler declaration

With support for nested compound statements, you can now use a compound statement within the declaration of a condition handler. Up to DB2 V8, the action of a condition handler could only be a single SQL procedure statement. With DB2 V9, the SQL statement can be a compound statement which in turn encloses one or more other SQL statements. This enables you to use multiple statements within the body of a condition handler by enclosing all of them within the compound statement.

Example 15-19 shows an SQL procedure that issues a DROP statement on a table name, which has been provided by an input parameter of a procedure. The procedure implements a condition handler that is triggered if the provided table name does not exist, and thus the DROP would result in an error. The handler implements a compound statement which prepares a string to return from the procedure, as well as writes further debug information into a predefined table MYRESULT.

Example 15-19 Using compound statements in a condition handler

```
CREATE PROCEDURE SCPCNHDL(IN  TABNAME VARCHAR(128),
                          OUT PARM VARCHAR(80))
  BEGIN
    DECLARE OUT_BUFFER VARCHAR(80);
```

```
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';

  A: BEGIN                                                -- (1)
     DECLARE DRPSTMT VARCHAR(256);
     DECLARE EXIT HANDLER FOR NO_TABLE                    -- (3)
        BEGIN                                             -- (4)
           SET OUT_BUFFER = 'ERROR DURING DROP TABLE';
           INSERT INTO MYRESULT
           VALUES ('TABLE '||TABNAME||' DOES NOT EXIST');
        END;

     -- DROP POTENTIALLY NONEXISTENT TABLE:
     SET DRPSTMT = 'DROP TABLE ' || TABNAME;
     EXECUTE IMMEDIATE DRPSTMT;                           -- (2)
     B: BEGIN
       SET OUT_BUFFER = 'TABLE DROPPED SUCCESSFULLY ';
     END B;
  END A;

  -- COPY OUT_BUFFER TO OUTPUT PARAMETER:
  C: BEGIN
     SET PARM = OUT_BUFFER;                               -- (5)
  END C;
END
```

Scoping condition handler declarations

Nested compound statements can be used to localize condition handling by scoping the declarations of condition handlers. Each compound statement has its own scope for variable definitions as well as for its condition names and condition handlers. The declaration of a condition handler associates the handler with an exception or completion condition in a compound statement.

The declaration specifies the condition that activates the condition handler, the type of the condition handler (CONTINUE or EXIT), and the handler action. The type of the condition handler determines where control is returned to after successful completion of the handler action.

The scope of the declaration of a condition handler is the compound statement in which it is declared, including any compound statements that are nested within that compound statement. A condition handler declared in a compound statement can handle a condition that is encountered in a compound statement that is enclosed within this compound statement if the condition is not handled at a lower level. However, a condition handler declared in an inner scope takes precedence over a condition handler defined to handle the same condition in an outer scope, even if the condition handler declared in an outer compound statement is more specific than a condition handler that is declared in an inner scope.

A condition handler is activated when it is the most appropriate condition handler for a condition that has been encountered. The most appropriate handler is the condition handler that most closely matches the SQLSTATE of the exception or completion condition.

Example 15-20 reconsiders the procedure SCOPECUR2 from Example 15-17 and augments it with two condition handler declarations. One condition handler is declared in the outermost block MAIN and catches general SQLExceptions. The procedure SCOPECUR2 would have

returned an error with SQLCODE -87 in case an empty string for the input parameter TYPE has been provided in the CALL. The enhancement with the EXIT HANDLER allows for a graceful return of the procedure when a SQL error occurs even for exceptions INSIDE of the GET_OBJ block. Furthermore, the output parm RES then contains debug information. In addition to this, the check for a SQLCODE <> 0 in the FETCH_LOOP has been replaced with another EXIT HANDLER. As a consequence of this handler, the cursor C1 is closed in the handler body, and the program logic of the procedure then continues with the execution after the GET_OBJ block, which basically returns from the procedure code.

Example 15-20 Scoping of condition handler declarations (1)

```
CREATE PROCEDURE SCOPEHND (IN  OBJOWNER  VARCHAR(128),
                           IN  TYPE   CHAR(1),
                           OUT RES    VARCHAR(128))

  VERSION V1
  LANGUAGE SQL
  MODIFIES SQL DATA
    MAIN: BEGIN
      DECLARE SELSTMT  VARCHAR(256);
      DECLARE TABNAME  VARCHAR(128);
      DECLARE OBJTYPE  VARCHAR(32);
      DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
      DECLARE SQLCODE  INTEGER DEFAULT 0;
      -- Declare a general SQL error condition handler
      DECLARE EXIT HANDLER FOR SQLEXCEPTION
        SET RES = 'SQL ERROR, SQLCODE= ' || CHAR(SQLCODE) ||
                  ' SQLSTATE= ' || SQLSTATE;

    -- SET THE TABLENAME ACCORDING TO TYPE PARAMETER
    IF (TYPE = 'S') THEN
       SET TABNAME = 'SYSIBM.SYSSEQUENCES';
       SET OBJTYPE = 'SEQUENCE';
    ELSEIF (TYPE = 'T') THEN
       SET TABNAME = 'SYSIBM.SYSTRIGGERS';
       SET OBJTYPE = 'TRIGGER';
    ELSEIF (TYPE = 'D') THEN
       SET TABNAME = 'SYSIBM.SYSDATATYPES';
       SET OBJTYPE = 'TYPE';
    END IF;

    -- INIT SQL SELECT STATEMENT
    SET SELSTMT = 'SELECT T.SCHEMA, T.NAME ' ||
                  'FROM '  || TABNAME  || ' AS T ' ||
                  'WHERE T.OWNER = ''' || OBJOWNER || '''';

    PREPARE SQLSTMT FROM SELSTMT;

    GET_OBJ: BEGIN
      DECLARE OBJNAME   VARCHAR(128);
      DECLARE OBJSCHEMA VARCHAR(128);
      DECLARE C1 CURSOR FOR SQLSTMT;
      -- Error handling for the Fetch Loop
      DECLARE EXIT HANDLER FOR NOT FOUND
        CLOSE C1;
```

```
    OPEN C1;
    FETCH_LOOP:
    LOOP
      FETCH C1 INTO OBJSCHEMA, OBJNAME;

      DRP_OBJ:
      BEGIN
        -- THIS SECTION DROPS THE ASSOCIATED OBJECT
        DECLARE DRPSTMT VARCHAR(256);
          SET DRPSTMT = 'DROP ' || OBJTYPE || ' ' ||
                        OBJSCHEMA || '.' || OBJNAME;
          EXECUTE IMMEDIATE DRPSTMT;
          INSERT INTO MYRESULT
          VALUES ('DROPPED '||OBJSCHEMA||'.'||OBJNAME);
      END DRP_OBJ;

    END LOOP FETCH_LOOP;
    CLOSE C1;

  END GET_OBJ;
  SET RES = 'Objects successfully removed';

END MAIN!
```

One additional example is shown in Figure 15-9.

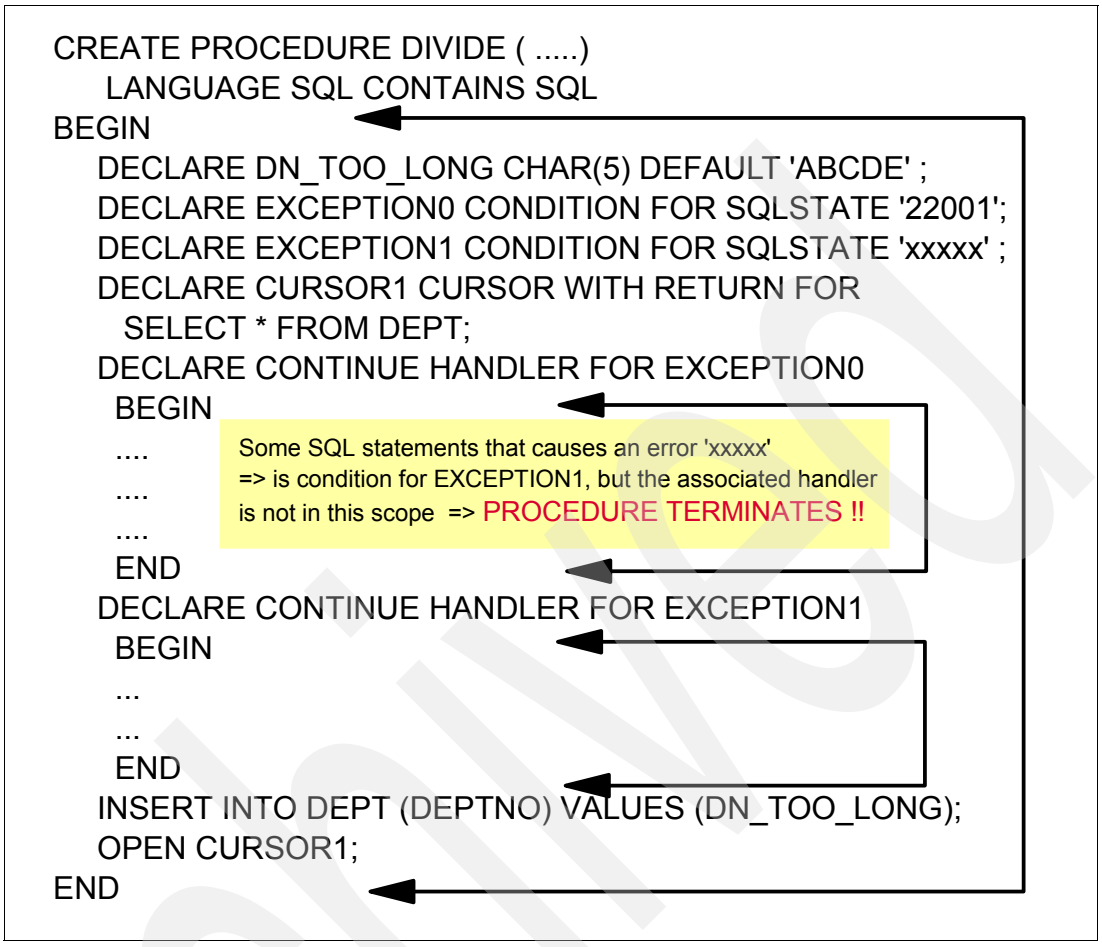

Figure 15-9 Scoping of condition handler declarations (2)

As shown in Figure 15-9, if exception condition exception1 is raised in the body of the condition handler for exception0, there is no appropriate handler. In this case the procedure is terminated because of the unhandled exception condition.

Note: Note that the condition handler defined for exception1 is not within the scope of the condition handler for exception0. Condition handlers that are declared in the same compound statement cannot handle conditions encountered in each other or in themselves. The scope of a condition handler includes the compound statement in which it is declared, but excludes the bodies of other condition handlers declared in the same compound statement.

Name resolution differences between native and external SQL procedures

In V9 the rules used for name resolution in a native SQL procedure differ from the rules that were used for SQL procedures in prior releases. Because an SQL parameter or SQL variable can have the same name as a column name, you should explicitly qualify the names of any SQL parameters, SQL variables, or columns that have non-unique names.

For more information about how the names of these items are resolved, see the topic “References to SQL parameters and SQL variables” in *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854. The rules used for name resolution in external SQL procedures remain unchanged.

There is one difference in the name resolution in native and external stored procedures that we need to explicitly mention. Assume that we have a procedure body that looks like that shown in Figure 15-10.

```
CREATE PROCEDURE ...
   BEGIN;
     DECLARE dept CHAR(3);
     DECLARE x CHAR(3);
     ...
     DECLARE c1 CURSOR FOR
       SELECT dept INTO x
       FROM emp;
     ..
  END;
```

Figure 15-10 Name resolution in external and native SQL stored procedures

As you can see, the dept is ambiguous. It is used twice:

- Declared as an SQL variable
- A column in the table emp

In the given situation, there is a difference between external and native SQL procedures in the way dept is resolved:

- External SQL stored procedures match this dept to the SQL variable in the declare statement.
- Native SQL stored procedures and SQL stored procedures on LUW and iSeries interpret this dept as the column name of table emp in the SELECT statement of cursor c1.

Recommendation: Qualify names if there is a potential for ambiguity. Refer to Figure 15-11 to see how the changed CREATE PROCEDURE might look when you follow this rule.

```
CREATE PROCEDURE ...
  STEP1: BEGIN;
     DECLARE dept CHAR(3);
     DECLARE x CHAR(3);
     DECLARE y CHAR(3);
     ...
     DECLARE c1 CURSOR FOR
       SELECT STEP1.dept, emp.dept INTO x,y
       FROM emp;
     ..
  END STEP1;
```

Figure 15-11 Better practice with name resolution in external and native SQL stored procedures

Summary of name scoping in compound statements

We have described the name scoping in compound statements on the last few pages in detail. Use summarization Table 15-3 for future reference.

Table 15-3 Summary of name scoping

Type of name	Must be unique within...	Qualification allowed?	Can be referenced within...
SQL variable	The compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement.	Yes, it can be qualified with the label of the compound statement in which the variable was declared.	The compound statement in which it is declared, including any compound statements that are nested within that compound statement. When multiple SQL variables are defined with the same name you can use a label to explicitly refer to a specific variable that is not the most local in scope.
condition	The compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement.	No	The compound statement in which it is declared, including any compound statements that are nested within that compound statement. Can be used in the declaration of a condition handler, or in a SIGNAL or RESIGNAL statement. Note: When multiple conditions are defined with the same name there is no way to explicitly refer to the condition that is not the most local in scope.
cursor	The compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement.	No	The compound statement in which it is declared, including any compound statements that are nested within that compound statement. Note: When multiple cursors are defined with the same name there is no way to explicitly refer to the cursor that is not the most local in scope. However, if the cursor is defined as a result set cursor (that is, the WITH RETURN clause was specified as part of the cursor declaration), the invoking application can access the result set.
label	The compound statement that declared the variable, including any declarations in compound statements that are nested within that compound statement.	No	The compound statement in which it is declared, including any compound statements that are nested within that compound statement. Use a label to qualify the name of an SQL variable or as the target of a GOTO, LEAVE, or ITERATE statement.

Allow an empty compound statement

As an aid to migration of applications from platforms and RDBMSs other than DB2 for z/OS, and for more compatibility within the DB2 family, DB2 V9 now allows you to have empty compound statements in your procedure body. The SQL Standard also allows the SQL procedure statement to be optional in the context of a compound statement in a routine.

Example 15-21 illustrates the use of an empty compound statement as a way of ignoring a condition. This might be useful when you are inserting a row into a table that has a unique column, and if the value to be inserted for the row already exists in the table, then the row does not need to be inserted. Although DB2 will detect the case where the value to be inserted is not unique, there is no need to tell the application about this. Instead, the INSERT statement will not be processed.

Example 15-21 Empty compound statement

```
DECLARE CONTINUE HANDLER FOR '23505'
BEGIN -- ignore error for duplicate value
END;
```

Support of new data types

More data types are now supported in native SQL procedure support. The data types BIGINT, BINARY, VARBINARY, and DECFLOAT can be used in SQL statements as well as SQL control statements (WHILE, IF, etc.). Example 15-22 implements these four new supported data types.

Example 15-22 New data types

```
CREATE PROCEDURE TYPES (OUT BIG1 BIGINT,
                        OUT BIN1 BINARY(50),
                        OUT VBIN1 VARBINARY(50),
                        OUT DFLT1 DECFLOAT)
  LANGUAGE SQL
  L1: BEGIN

    DECLARE BIG2 BIGINT;
    DECLARE BIN2 BINARY(50);
    DECLARE VBIN2 VARBINARY(50);
    DECLARE DFLT2 DECFLOAT;

    SET BIG2 = 10000000000098237;
    SET BIN2 = BX'A9A9A9';
    SET VBIN2 = BX'C9C9C9';
    SET DFLT2 = 1.234567891011;

    SET BIG1 = BIG2;
    SET BIN1 = BIN2;
    SET VBIN1 = VBIN2;
    SET DFLT1 = DFLT2;
  END L1#
```

Restriction: XML, UDTs, ROWIDs, LOB locators, and LOB file references are not yet supported.

15.3 Versioning

In V9, the option VERSION is added in the CREATE PROCEDURE and ALTER PROCEDURE statements for native SQL procedures. This means that multiple versions can be created or added for one stored procedure. Any version can be “promoted” to be the active version, but only one can be the current, active version for one procedure. By default, the current active version will be the one to run when the stored procedure is called.

Maintaining existing and adding additional versions is implemented with the help of the ALTER PROCEDURE statement. You can embed this statement in an application program or issue it interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified. If your ALTER statement contains either a REPLACE VERSION or a ADD VERSION clause, the statement can only be dynamically prepared.

In this chapter the sample native SQL procedure MEDIAN_RESULT_SET, introduced in Example 15-1, is used to discuss some general ideas of the versioning concept.

Tip: If you create the following SQL procedures in SPUFI, make sure the proper settings for SQLFORMAT and SQLTERM are performed, as described in “SQL FORMAT on SPUFI defaults panel” on page 261.

15.3.1 Identifying the version to change

To identify the version of the procedure that is to be changed, replaced, or regenerated by the ALTER statement, the following two keywords can be used.

- ACTIVE VERSION

  The currently active version of the procedure is to be changed, replaced, or regenerated as shown in Example 15-23.

  Example 15-23 Alter of active version

```
ALTER PROCEDURE MEDIAN_RESULT_SET
   ALTER ACTIVE VERSION...
```

- VERSION routine-version-id

  routine-version-id is the version identifier that is assigned when the version is defined. This must identify a version of the specified procedure that exists at the current server. The version ID is an SQL identifier of up to 124 UTF-8 bytes. See Example 15-24.

  Example 15-24 Version ID

```
ALTER PROCEDURE MEDIAN_RESULT_SET
   ALTER VERSION MEDIAN_V1...
```

15.3.2 Adding a new version

You use

```
ADD VERSION routine-version-id
```

to specify that a new version of the procedure is to be created. routine-version-id is the version identifier for the new version of the procedure. routine-version-id must not identify a version of the specified procedure that already exists at the current server. When a new version of a procedure is created, the comment that is recorded in the catalog for the new version will be the same as the comment that is in the catalog for the currently active version. When you add a new version of a procedure the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the procedure. The parameter names can differ from the other versions of the procedure. For options that are not explicitly specified, the system default values will be used.

In Example 15-25 a new version of the MEDIAN_RESULT_SET procedure (created in Example 15-1 on page 258) is introduced.

Example 15-25 Add new version: MEDIAN_V2

```
ALTER PROCEDURE MEDIAN_RESULT_SET
  ADD VERSION MEDIAN_V2
     (OUT MEDIANSALARY DECIMAL(7,2))
     LANGUAGE SQL
```

```
    READS SQL DATA
    DYNAMIC RESULT SETS 1
    BEGIN
      DECLARE V_NUMRECORDS INTEGER DEFAULT 1;
      DECLARE V_COUNTER INTEGER DEFAULT 0;
      DECLARE C1 CURSOR FOR
        SELECT SALARY FROM STAFF ORDER BY SALARY;
      DECLARE C2 CURSOR WITH RETURN FOR
        SELECT NAME, JOB, SALARY
          FROM STAFF
          WHERE SALARY <= MEDIANSALARY
          ORDER BY SALARY;
      DECLARE EXIT HANDLER FOR NOT FOUND
        SET MEDIANSALARY = 0;
      SELECT COUNT(*) INTO V_NUMRECORDS FROM STAFF;
      OPEN C1;
      WHILE V_COUNTER < (V_NUMRECORDS / 2 + 1) DO
        FETCH C1 INTO MEDIANSALARY;
        SET V_COUNTER = V_COUNTER + 1;
      END WHILE;
      CLOSE C1;
     OPEN C2;
  END#
```

Example 15-25 basically adds the new version MEDIAN_V2 to the existing SQL procedure. In addition to this the selection criterion for the returned result set is changed as well. The two DB2 catalog tables SYSIBM.SYSROUTINES and SYSIBM.SYSPARMS each now contain two rows for the procedure MEDIAN_RESULT_SET, one for MEDIAN_V1 and the other for MEDIAN_V2.

15.3.3 Activating an existing version

Only one of the two MEDIAN_RESULT_SET versions is the currently active version. To find out which version is the active one, SYSIBM.SYSROUTINES contains a column named ACTIVE. Currently MEDIAN_V1 should be the active version, which implies that the column value for this row should specify a Y. The ACTIVE column value for MEDIAN_V2 should feature an N. Calling the procedure would thus return a result set that contains all employees with a salary larger than the median.

In order to make MEDIAN_V2 the active version, which is then used in the next CALL statement, a second ALTER statement with the ACTIVATE option has to be executed:

```
ALTER PROCEDURE MEDIAN_RESULT_SET ACTIVATE VERSION MEDIAN_V2#
```

Starting from this moment, DB2 executes procedure MEDIAN_RESULT_SET MEDIAN_V2 when it is called. Furthermore, the values for the ACTIVE column in catalog table SYSIBM.SYSROUTINES now reflect this new default.

Note: A way to interactively specify the version to be executed is to employ the special register CURRENT ROUTINE VERSION. If set, this register overrides the setting in the ACTIVE column in the SYSROUTINES catalog table. In the section about stored procedure execution, more details are provided.

15.3.4 Rebinding an existing version

When DB2 maintenance is applied that changes how an SQL procedure is generated, the procedure might need to be regenerated to process the maintenance changes.

REGENERATE automatically rebinds, at the local server, the package for the SQL control statements for the procedure and rebinds the package for the SQL statements that are included in the procedure body. If a remote bind is also needed, you must explicitly use the BIND PACKAGE COPY command for all of the remote servers.

The following command causes the regeneration of the active version of the MEDIAN_RESULT_SET procedure, which is still MEDIAN_V2:

```
ALTER PROCEDURE MEDIAN_RESULT_SET
  REGENERATE ACTIVE VERSION#
```

Note: REGENERATE is different from a REBIND PACKAGE in the sense that the REBIND PACKAGE command implies the generation of possibly better access paths for the embedded SQL statements. REGENERATE only affects the SQL control statements in the procedure definition. These stay the same for a REBIND PACKAGE.

15.3.5 Replacing the active version

This specifies that a version of the procedure is to be replaced. When you bind the replaced version of the procedure, this might result in a new access path even if the routine body is not changed. When you replace a procedure, the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the procedure. For options that are not explicitly specified, the system default values for those options are used, even if those options were explicitly specified for the version of the procedure that is being replaced.

Example 15-26 illustrates how version MEDIAN_V2 of the MEDIAN_RESULT_SET procedure is replaced by a new one. The code updates basically consist of some restructuring and the introduction of nested compound statements.

Example 15-26 Replace version MEDIAN_V2

```
ALTER PROCEDURE MEDIAN_RESULT_SET
  REPLACE VERSION MEDIAN_V2
     (OUT MEDIANSALARY DECIMAL(7,2))
    LANGUAGE SQL
     READS SQL DATA
     DYNAMIC RESULT SETS 1
     MAIN: BEGIN
       DECLARE C2 CURSOR WITH RETURN FOR
         SELECT NAME, JOB, SALARY
            FROM STAFF
           WHERE SALARY <= MEDIANSALARY
           ORDER BY SALARY;
       GET_MEDIAN:
       BEGIN
          DECLARE V_NUMRECORDS INTEGER DEFAULT 1;
          DECLARE V_COUNTER INTEGER DEFAULT 0;
          DECLARE C1 CURSOR FOR
            SELECT SALARY FROM STAFF ORDER BY SALARY;
          DECLARE EXIT HANDLER FOR NOT FOUND
```

```
      SET MEDIANSALARY = 0;
    SELECT COUNT(*) INTO V_NUMRECORDS FROM STAFF;
    OPEN C1;
    WHILE V_COUNTER < (V_NUMRECORDS / 2 + 1) DO
      FETCH C1 INTO MEDIANSALARY;
      SET V_COUNTER = V_COUNTER + 1;
    END WHILE;
    CLOSE C1;
  END GET_MEDIAN;
  OPEN C2;
END MAIN#
```

15.3.6 Dropping an existing version

This drops the version of the procedure that is identified with *routine-version-id*, which is the version identifier that is assigned when the version is defined. routine-version-id must identify a version of the procedure that already exists at the current server and must not identify the currently active version of the procedure. Only the identified version of the procedure is dropped.

Example 15-27 employs the ALTER statement to drop version MEDIAN_V1. MEDIAN_V2 cannot be dropped because this is the currently active procedure version.

Example 15-27 Drop Version MEDIAN_V1

```
ALTER PROCEDURE MEDIAN_RESULT_SET
  DROP VERSION MEDIAN_V1#
```

When only a single version of the procedure exists at the current server, you can use the DROP PROCEDURE statement to drop the procedure. A version of the procedure for which the version identifier is the same as the contents of the CURRENT ROUTINE VERSION special register can be dropped if that version is not the currently active version of the procedure.

15.3.7 ALTER PROCEDURE syntax

Figure 15-12, Figure 15-13 on page 297, and Figure 15-14 on page 298 show the complete ALTER PROCEDURE syntax for native SQL stored procedures. You have a few more options here compared to the ALTER PROCEDURE for external SQL procedures.

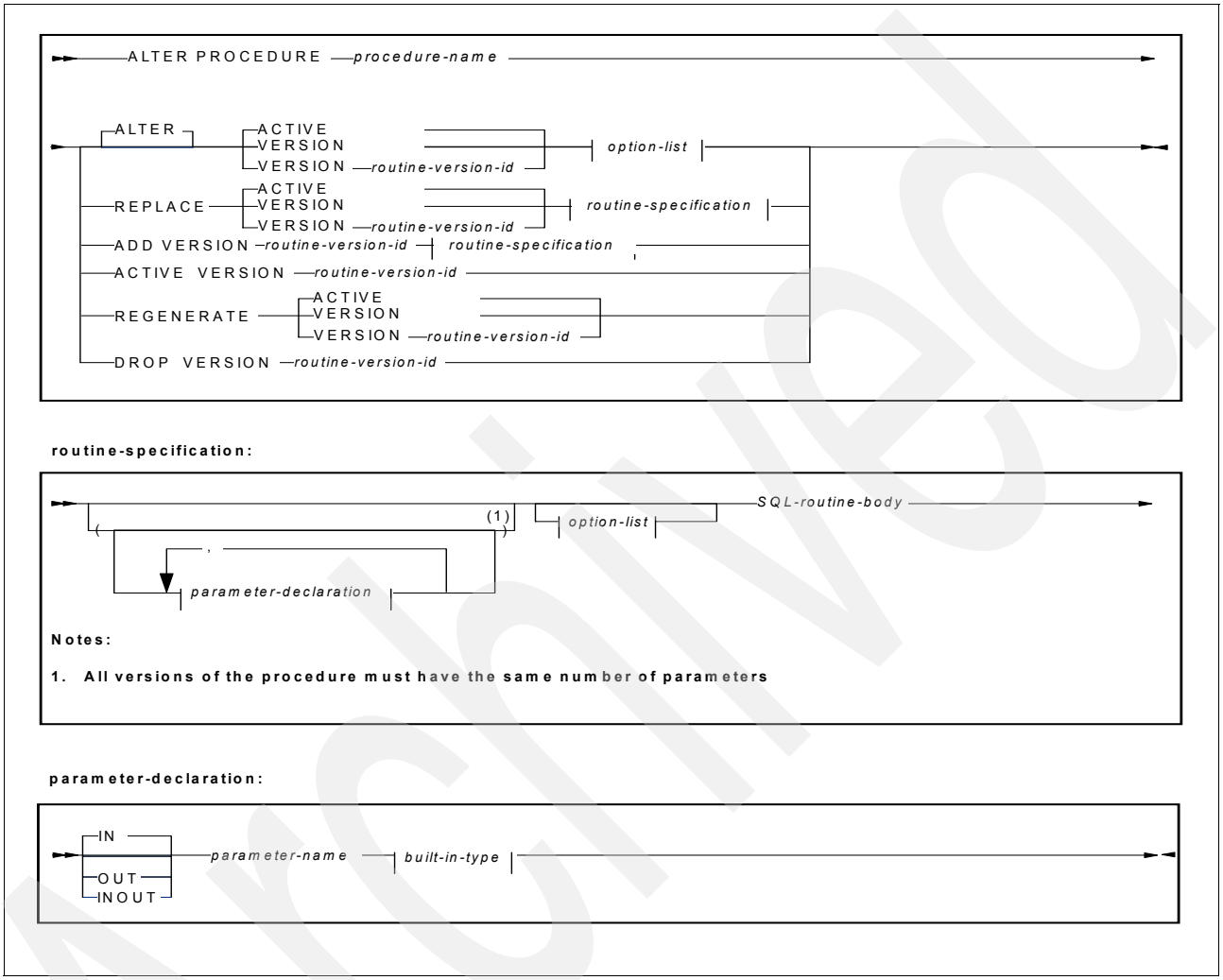

Figure 15-12 Start of ALTER PROCEDURE syntax

Figure 15-13 shows the continuation of the ALTER PROCEDURE statement with the built-in type.

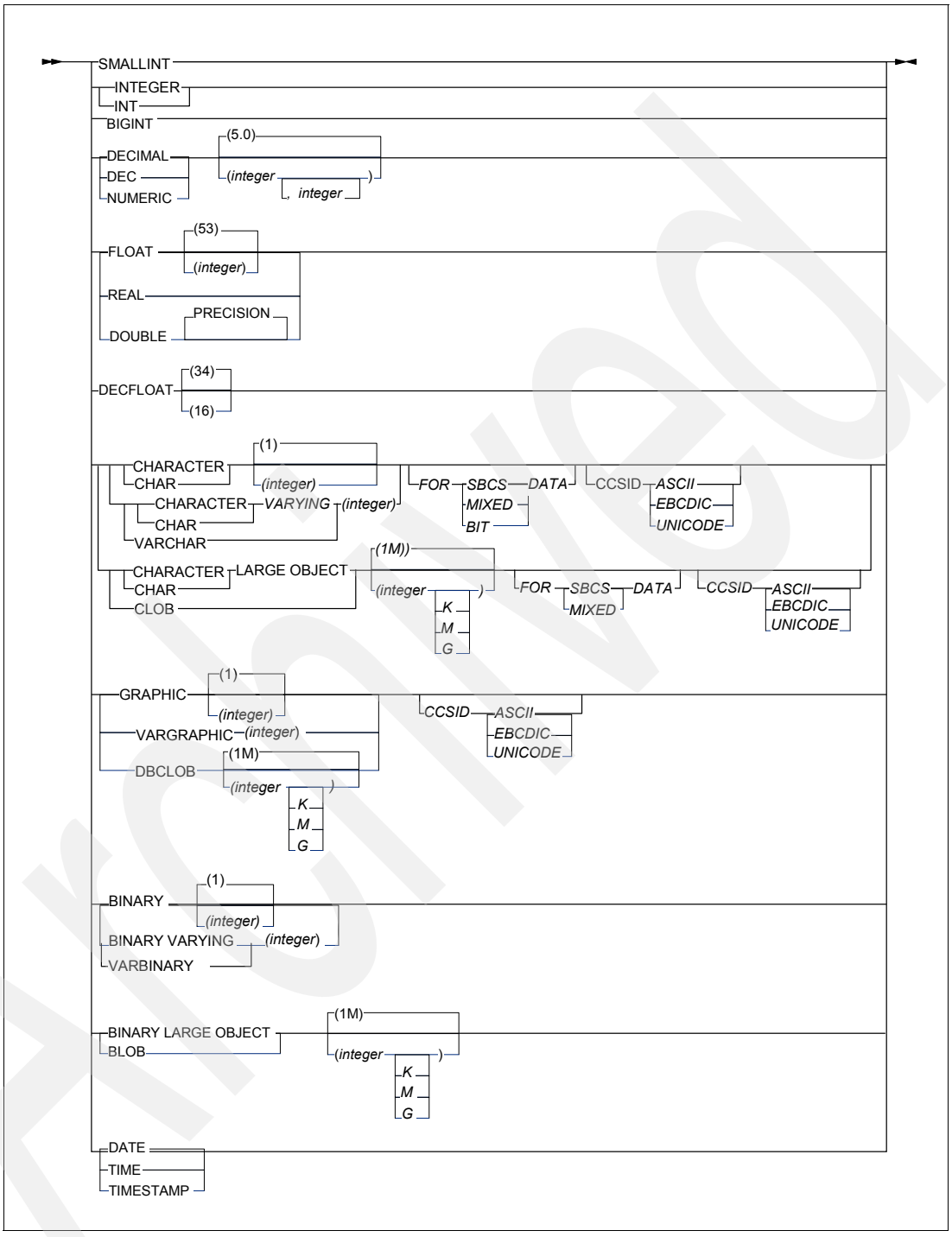

Figure 15-13 ALTER PROCEDURE built-in type

Figure 15-14 shows the last part of the CREATE PROCEDURE syntax with the option list.

Most of the options are the same as described at “CREATE PROCEDURE syntax” on page 265.

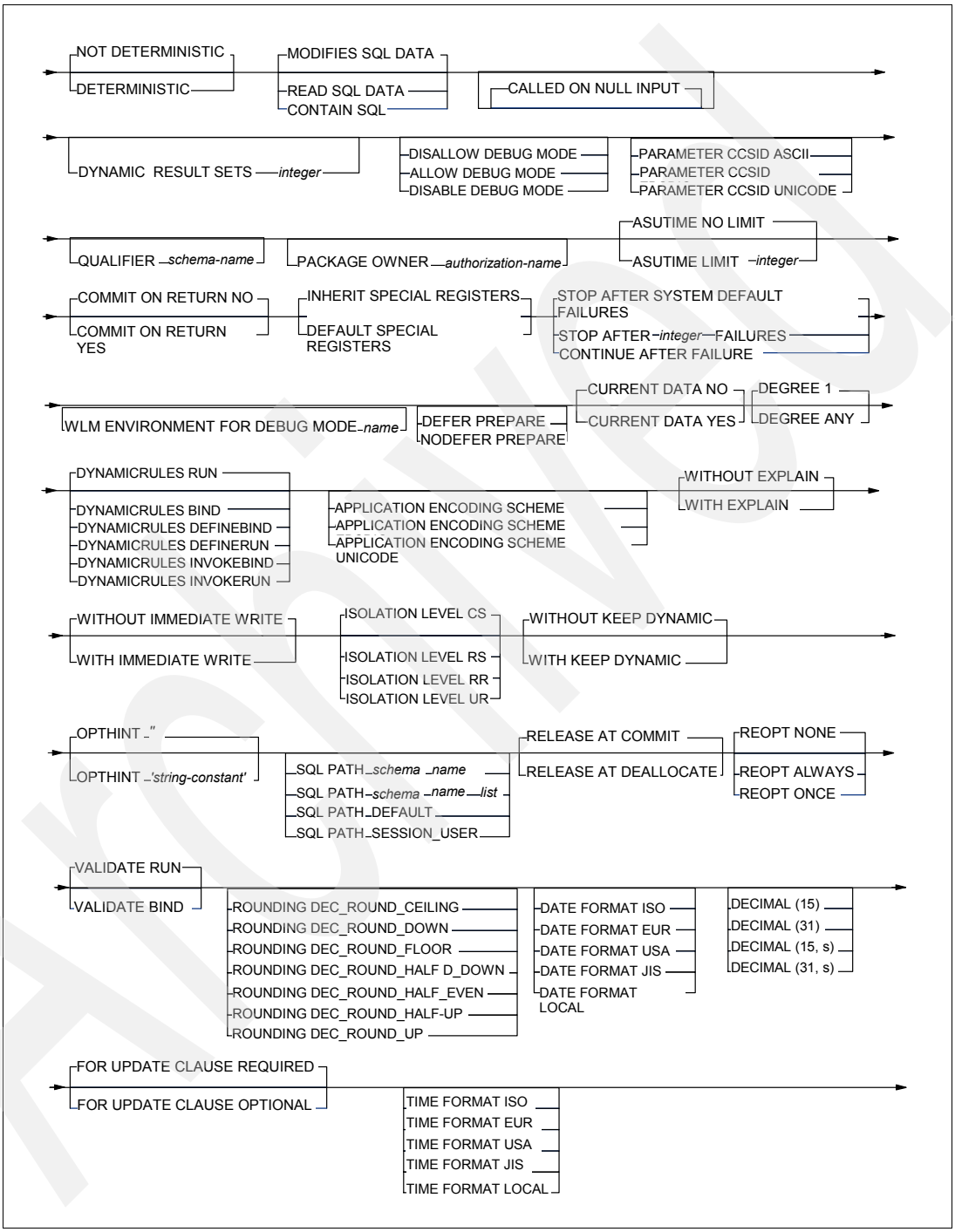

Figure 15-14 ALTER PROCEDURE option-list

15.4 Execution of a native SQL procedure

With V9 new function mode, when you create a native SQL stored procedure, its procedural statements are now converted to a native representation that is stored in the DB2 catalog and directory, as is done with other SQL statements. The parameter list and procedure options are stored in the database catalog tables as in the prior releases. When you call a native SQL procedure, DB2 loads the native representation from the catalog and the DB2 engine executes the procedure.

Figure 15-15 depicts the DB2 components involved when a native SQL procedure is called either from a remote application, a DB2 attached program, or an allied address space respectively. As illustrated, the SQL statements are no longer executed in an external WLM address space but natively in the database system services address space. For execution, the procedure packages are loaded into the EDM pool.

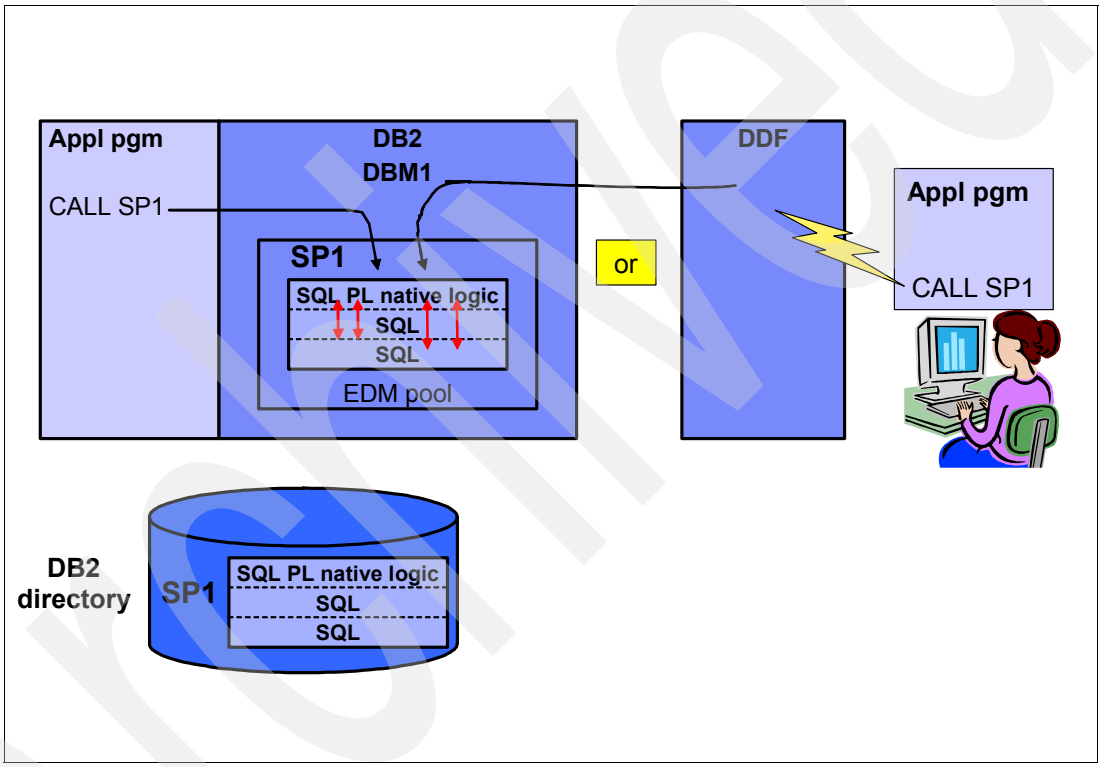

Figure 15-15 Native SQL procedure execution in DB2 9 for z/OS

15.4.1 Which procedure is executed

When calling a native SQL procedure, the resolution is performed with the help of the schema name, the procedure name, and the number of parameters. Parameter overloading is not supported.

Once the correct procedure has been resolved, the current active version is determined. Only one active version of a native SQL procedure exists at any point in time. By default the current active version is the version that contains a flag Y in the ACTIVE column of the catalog table SYSIBM.SYSROUTINES. Any version that exists for a native SQL procedure can be “promoted” to be the active version with the ACTIVATE VERSION option of the ALTER statement. To default the execution of the procedure MEDIAN_RESULT_SET to version MEDIAN_V2, the following command can be used:

```
ALTER PROCEDURE MEDIAN_RESULT_SET ACTIVATE VERSION MEDIAN_V2#
```

When the stored procedure is now invoked via the SQL CALL statement, the second version (MEDIAN_V2) is called and executed. Example 15-28 shows a small Java excerpt that executes the SQL procedure:

Example 15-28 Sample Java invocation (1)

```
CallableStatement cstmt =
        con.prepareCall("CALL PAOLOR3.MEDIAN_RESULT_SET(?)");

        cstmt.registerOutParameter(1, Types.INTEGER);

        boolean hasResultSet = false;
        hasResultSet = cstmt.execute();
```

For ad hoc testing of a specific version, a new special register called CURRENT ROUTINE VERSION is provided. If an application sets this register to an existing valid version, it temporarily overrides the setting in column ACTIVE of the SYSIBM.SYSROUTINES catalog table. If the provided value is not valid, the default catalog settings are used. The subsequent CALL to the stored procedure has to be dynamically prepared, otherwise the register is not taken into account. This is mainly because a static CALL to the native SQL procedure already has the version identifier bound to in the respective package.

Example 15-29 makes use of the CURRENT ROUTINE VERSION register to invoke the version MEDIAN_V1 for the subsequent CALL to the procedure.

Example 15-29 Sample Java invocation (2)

```
       Statement stmt = con.createStatement();
        stmt.executeUpdate("SET CURRENT ROUTINE VERSION = 'MEDIAN_V1'");

        CallableStatement cstmt =
        con.prepareCall("CALL PAOLOR3.MEDIAN_RESULT_SET(?)");

        cstmt.registerOutParameter(1, Types.INTEGER);

        boolean hasResultSet = false;
        hasResultSet = cstmt.execute();
```

Tip: When you use the CURRENT ROUTINE VERSION special register to test a version of one or more native SQL procedures, you should use a routine version identifier that is a value other than the default value (V1) on the CREATE PROCEDURE statement. This will avoid having the special register affect more procedures than you intend when testing a new version of a procedure. For example, assume that you want to run version VER2 of procedure P1, and procedure P1 invokes another procedure, P2. If a version exists for both procedures P1 and P2 with the routine version identifier VER2, that version will be used for both procedures.

To fall back to the default version specified in the catalog, just set this register to an empty string constant. The register's value is not cached in the catalog, that is, in case the thread which set this value terminates, the register's content is no longer accessible and the default catalog value determines the current version.

Tip: If you do not want to write an application program for calling your stored procedure, you can do this via the Developer Workbench, which comes with DB2 9 for LUW and the DB2 for z/OS Client Management Package and replaces the Development Center. This tool is used to handle similar development and testing functionality for DB2 UDB V8 for LUW.

15.5 Deployment of a native SQL procedure to another server

Deployment of an SQL procedure is the step of distributing or installing the procedure created on one system to other system(s). Prior to V9, customers deployed SQL stored procedures by:

- Copying over the load modules of the stored procedures (this ensures that the logic of the stored procedure body is not changed after deployment)
- Sending DBRM for the stored procedure over and issuing a BIND PACKAGE
- Issuing CREATE PROCEDURE to define the procedure

The DB2 enhanced support for deployment of native SQL procedures is useful if you want to install a native SQL procedure to a production system after it has been tested well on the test system. This can now be done via the extended BIND PACKAGE command featuring the new keyword DEPLOY.

Deployment is different from remote BIND package, because the logic of the procedure body (stored as a special section in the package) will not be re-bound. That means you do not need to worry about unexpected behavior change after the deployment.

Scenario for deployment

The following scenario shows you how to create, test, and deploy multiple versions of an SQL stored procedure from one server to another. When done, you can change a version's logic again, test it, and redeploy it again for general use. The deployment requires that the remote server is properly defined in the communication database of the DB2 subsystem that conducts the deployment.

New DEPLOY keyword on the BIND command

DEPLOY is a new BIND option. The syntax for the DEPLOY option is:

```
BIND PACKAGE ..... DEPLOY(collection-id.package-id) COPYVER(version-id) ...
```

You can use DEPLOY when the target DB2 subsystem is also a DB2 for z/OS subsystem that is operating with a compatible PTF level. If the PTF level is not compatible, SQLCODE -20249 is issued.

If you specify ACTION(ADD) for a version that does not exist at the target location, DB2 creates or adds a new version of the native SQL procedure and its associated package while keeping the source native SQL procedure's logic. DB2 adds a new version if a native SQL procedure with the same target name already exists.

If you specify ACTION(REPLACE), DB2 replaces the version specified in COPYVER. If you specify REPLVER, the version ID must be the same as the COPYVER version ID or DB2 returns TSO error message DSNE977E.

For the example shown in the next few figures starting with Figure 15-16, we used SANJOSE as the name of the current server and BERLIN as the name of the remote server.

Figure 15-16 shows the CREATE statement for the first version (MEDIAN_V1) of the native SQL procedure MEDIAN_RESULT_SET introduced earlier in this chapter. This time a schema name of TEST is used.

```
CREATE PROCEDURE TEST.MEDIAN_RESULT_SET
   (OUT MEDIANSALARY DECIMAL(7,2))
 VERSION MEDIAN_V1
 LANGUAGE SQL
 READS SQL DATA
 DYNAMIC RESULT SETS 1
  BEGIN
    DECLARE V_NUMRECORDS INTEGER DEFAULT 1;
    DECLARE V_COUNTER INTEGER DEFAULT 0;
    DECLARE C1 CURSOR FOR
      SELECT SALARY FROM STAFF ORDER BY SALARY;
    DECLARE C2 CURSOR WITH RETURN FOR
      SELECT NAME, JOB, SALARY
         FROM STAFF
        WHERE SALARY > MEDIANSALARY
        ORDER BY SALARY;
    DECLARE EXIT HANDLER FOR NOT FOUND
      SET MEDIANSALARY = 0;
    SELECT COUNT(*) INTO V_NUMRECORDS FROM STAFF;
    OPEN C1;
    WHILE V_COUNTER < (V_NUMRECORDS / 2 + 1) DO
      FETCH C1 INTO MEDIANSALARY;
      SET V_COUNTER = V_COUNTER + 1;
    END WHILE;
    CLOSE C1;
    OPEN C2;
  END#
```

Figure 15-16 CREATE PROCEDURE SAMPLE for deployment on server SANJOSE

Notice that the schema name used becomes the collection-id of the package. At the same time the procedure name also becomes the package-id itself. This is important in the following deployment step.

The next step in the deployment process is to create a new SQL procedure for the production environment on a remote DB2 subsystem, named BERLIN.PRODUCTION.MEDIAN_RESULT_SET from the existing SQL procedure that we have just created (that is, SANJOSE.TEST.MEDIAN_RESULT_SET).

As you can see from Figure 15-17 on page 303, both procedures have the same version MEDIAN_V1. The new procedure schema and at the same time collection-id is PRODUCTION, whereas the procedure name as well as the package-id stay the same. Furthermore the procedure BERLIN.PRODUCTION.MEDIAN_V1 has a different qualifier, as this makes sense for the production environment. The example shown in Figure 15-17 on page 303 is considered being a remote bind on location BERLIN.

```
BIND PACKAGE(BERLIN.PRODUCTION)       -
     DEPLOY(TEST.MEDIAN_RESULT_SET) -
     COPYVER(MEDIAN_V1)             -
     ACTION(REPLACE)                -
     QUALIFIER(PAOLOR1)
```

Figure 15-17 Bind Package statement with DEPLOY option

Now that V1 of SANJOSE.TEST.MEDIAN_RESULT_SET has successfully been deployed to BERLIN.PRODUCTION.MEDIAN_RESULT_SET, we can, for example, add a new version of MEDIAN_RESULT_SET at the SANJOSE server, as shown in Figure 15-18; run function tests to make sure that it works properly; and deploy the new version to the production environment in BERLIN once we are satisfied with it.

```
ALTER PROCEDURE TEST.MEDIAN_RESULT_SET
  ADD VERSION MEDIAN_V2
     (OUT MEDIANSALARY DECIMAL(7,2))
     LANGUAGE SQL
     READS SQL DATA
     DYNAMIC RESULT SETS 1
     MAIN: BEGIN
       DECLARE C2 CURSOR WITH RETURN FOR
         SELECT NAME, JOB, SALARY
           FROM STAFF
          WHERE SALARY <= MEDIANSALARY
          ORDER BY SALARY;
       GET_MEDIAN:
       BEGIN
          DECLARE V_NUMRECORDS INTEGER DEFAULT 1;
          DECLARE V_COUNTER INTEGER DEFAULT 0;
          DECLARE C1 CURSOR FOR
            SELECT SALARY FROM STAFF ORDER BY SALARY;
          DECLARE EXIT HANDLER FOR NOT FOUND
            SET MEDIANSALARY = 0;
          SELECT COUNT(*) INTO V_NUMRECORDS FROM STAFF;
          OPEN C1;
          WHILE V_COUNTER < (V_NUMRECORDS / 2 + 1) DO
            FETCH C1 INTO MEDIANSALARY;
            SET V_COUNTER = V_COUNTER + 1;
          END WHILE;
          CLOSE C1;
       END GET_MEDIAN;
       OPEN C2;
     END MAIN#
```

Figure 15-18 ALTER PROCEDURE add MEDIAN_V2

Refer to the bind statement shown in Figure 15-19, which we used to deploy version MEDIAN_V2 of procedure SANJOSE.TEST.MEDIAN_RESULT_SET to BERLIN.PRODUCTION.UPDATE_BALANCE.

```
BIND PACKAGE(BERLIN.PRODUCTION)       -
     DEPLOY(TEST.MEDIAN_RESULT_SET) -
     COPYVER(MEDIAN_V2)             -
     ACTION(REPLACE)                -
     QUALIFIER(PAOLOR1)
```

Figure 15-19 Bind package statement with deploy option for MEDIAN_V2

15.6 DB2/DSN/SQL command changes

There are some changes when working with native versus external SQL procedures. The new command outputs are explained here.

15.6.1 START/STOP PROCEDURE

When you issue a START or STOP PROCEDURE command for your native SQL stored procedure the command affects all versions of a procedure. There is no way to start or stop a specific version.

STOP PROCEDURE(TEST.MEDIAN_RESULT_SET) thus stops both versions MEDIAN_V1 and MEDIAN_V2.

15.6.2 -DISPLAY PROCEDURE

When you start working with native SQL procedures, you should be aware of the slightly different behavior of the -DISPLAY PROCEDURE DB2 command. Native SQL procedures are not displayed in the DISPLAY PROCEDURE output except for two situations:

- If specific native SQL procedures are under the effect of a STOP PROCEDURE command, then the procedure names and status will be displayed, but the statistics will be 0. Example 15-30 shows the `DISPLAY PROCEDURE(PAOLOR3.*)` output for two stopped procedures defined in the schema PAOLOR3.
  The two procedures have been explicitly stopped with the following commands:

  ```
  STOP PROCEDURE(PAOLOR3.MEDIAN_RESULT_SET)
  STOP PROCEDURE(PAOLOR3.NODIFF)
  ```

  Example 15-30 DISPLAY output - specific procedures have been stopped

  ```
  DSNX940I  -DB9A DSNX9DIS DISPLAY PROCEDURE REPORT FOLLOWS -
  ------- SCHEMA=PAOLOR3
  PROCEDURE      STATUS ACTIVE QUED MAXQ TIMEOUT FAIL WLM_ENV
  MEDIAN_RESULT_SET
                 STOPQUE     0    0    0       0    0
  NODIFF
                 STOPQUE     0    0    0       0    0
  DSNX9DIS DISPLAY PROCEDURE REPORT COMPLETE
  DSN9022I  -DB9A DSNX9COM '-DISPLAY PROC' NORMAL COMPLETION
  ```

- In case all procedures in the PAOLOR3 schema are stopped, a different `DISPLAY PROCEDURE(PAOLOR3.*)` is obtained. Example 15-31 shows the output after the following command has been issued:

  STOP PROCEDURE(PAOLOR3.*)

Example 15-31 DISPLAY output - all procedures in a schema have been stopped

```
STC00053  DSNX940I  ) DSNX9DIS DISPLAY PROCEDURE REPORT FOLLOWS -

------- SCHEMA=PAOLOR3
DSNX9DIS PROCEDURES A - Z* STOP QUEUE
DSNX9DIS DISPLAY PROCEDURE REPORT COMPLETE
STC00053  DSN9022I  ) DSNX9COM '-DISPLAY PROC' NORMAL COMPLETION
```

- Once the SQL procedure is started again, it does not appear in the DISPLAY PROCEDURE output anymore. See Example 15-32.

Example 15-32 DISPLAY output - procedures started

```
DSNX940I  -DB9A DSNX9DIS DISPLAY PROCEDURE REPORT FOLLOWS -
------- SCHEMA=PAOLOR3
DSNX9DIS PROCEDURE HAS NOT BEEN ACCESSED OR IS NOT DEFINED
DSNX9DIS DISPLAY PROCEDURE REPORT COMPLETE
DSN9022I  -DB9A DSNX9COM '-DISPLAY PROC' NORMAL COMPLETION
***
```

- A native SQL procedure that is currently being debugged will show under DISPLAY PROCEDURE command as in ACTIVE state. This is a consequence of the fact that the procedure that is being debugged is executing in a WLM environment. The WLM environment is specified in the CREATE PROCEDURE or ALTER PROCDURE statement. For the procedure MEDIAN_RESULT_SET the following statement has been applied:

```
WLM ENVIRONMENT FOR DEBUG MODE DB9AWLM ALLOW DEBUG MODE
```

Example 15-33 contains the DISPLAY PROCEDURE output for a currently debugged MEDIAN_RESULT_SET procedure. Notice that the WLM_ENV column features a value of the specified WLM environment DB9AWLM.

Example 15-33 Debugged procedure (DISPLAY output)

```
DSNX940I  -DB9A DSNX9DIS DISPLAY PROCEDURE REPORT FOLLOWS -
------- SCHEMA=PAOLOR3
PROCEDURE      STATUS ACTIVE QUED MAXQ TIMEOUT FAIL WLM_ENV
MEDIAN_RESULT_SET
               STARTED     1    0    1       0    0 DB9AWLM
DSNX9DIS DISPLAY PROCEDURE REPORT COMPLETE
DSN9022I  -DB9A DSNX9COM '-DISPLAY PROC' NORMAL COMPLETION
***
```

15.6.3 REBIND PACKAGE

When you issue a REBIND PACKAGE statement against a native SQL procedure, the only bind option that you can change with it is the EXPLAIN bind option.

REBIND PACKAGE only rebinds the SQL statements that are included in the procedure. The native representation of the logic part (that is, the control statements in the procedure definition) is not rebound.

Note: The behavior mentioned above is different from ALTER PROCEDURE REGENERATE, which is described in 15.3.7, “ALTER PROCEDURE syntax” on page 296. ALTER PROCEDURE REGENERATE does not only rebind all the SQL statements, but also regenerates the native representation of the logic part.

15.6.4 Impact on other SQL statements

In addition to the CREATE and ALTER PROCEDURE modifications, more SQL enhancements have been introduced to support the new native SQL procedures.

COMMENT ON PROCEDURE

The COMMENT ON PROCEDURE statement has been extended to handle multiple versions of a procedure. Similar to the ALTER PROCEDURE statement, a distinct procedure version can be defined with either the ACTIVE VERSION or the VERSION routine-version-id keyword. Example 15-34 shows how to add a value to the REMARKS columns in SYSIBM.SYSROUTINES of version MEDIAN_V2.

Example 15-34 Comment on procedure MEDIAN_RESULT_SET

```
COMMENT ON PROCEDURE PAOLOR3.MEDIAN_RESULT_SET
  VERSION MEDIAN_V2
  IS 'THIS IS A TEST COMMENT';
```

GRANT and REVOKE

Privileges granted or revoked are the same for all versions of a native SQL procedure.

DROP statement

The SQL DROP statement drops all versions of a SQL procedure as well as all associated packages (packages that are remotely bound are not dropped). To drop only one existing version of a procedure, and in addition only the package associated with this version, the statement `ALTER PROCEDURE...DROP VERSION routine-version-id` can be used (refer to 15.3.6, “Dropping an existing version” on page 295).

15.6.5 New stored procedure-related special registers

Two new special registers are available with native SQL procedures: CURRENT DEBUG MODE and CURRENT ROUTINE VERSION.

CURRENT DEBUG MODE

The SET CURRENT DEBUG MODE statement shown in Figure 15-20 assigns a value to the CURRENT DEBUG MODE special register. The special register sets the default value for the DEBUG MODE option for CREATE PROCEDURE statements that define native SQL or Java procedures, or ALTER PROCEDURE statements that create or replace a version of a native SQL procedure.

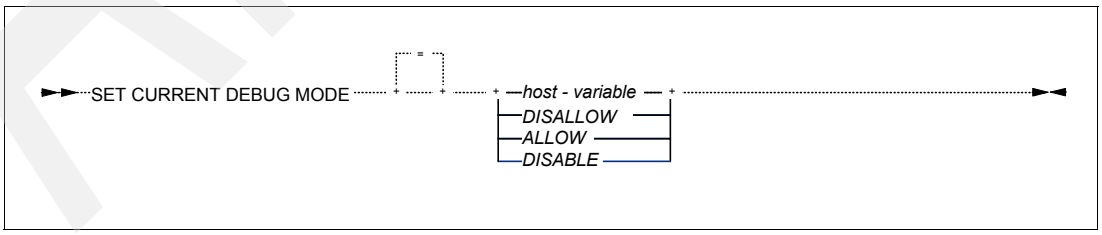

Figure 15-20 SET CURRENT DEBUG MODE syntax

The mode options are:

- DISALLOW

  This specifies that DISALLOW DEBUG MODE is the default option for CREATE PROCEDURE statements when defining an SQL or Java procedure, or ALTER PROCEDURE statements that create or replace a version of a native SQL procedure.

- ALLOW

  This specifies that ALLOW DEBUG MODE is the default option for CREATE PROCEDURE statements when defining an SQL or Java procedure, or ALTER PROCEDURE statements that create or replace a version of a native SQL procedure.

- DISABLE

  This specifies that DISABLE DEBUG MODE is the default option for CREATE PROCEDURE statements when defining an SQL or Java procedure, or ALTER PROCEDURE statements that create or replace a version of a native SQL procedure.

For more information on the DEBUG MODE options, refer to 15.2.2, “CREATE PROCEDURE syntax” on page 265.

A sample illustrating how to work with this register is provided in 15.7, “Error handling and debugging” on page 308.

CURRENT ROUTINE VERSION

The SET CURRENT ROUTINE VERSION statement assigns a value to the CURRENT ROUTINE VERSION special register. The special register sets the override value for the version identifier of native SQL procedures when they are invoked. You can issue this statement interactively or embed it in an application program. It is an executable statement that can be dynamically prepared.

Figure 15-21 shows the syntax of the associated SET statement.

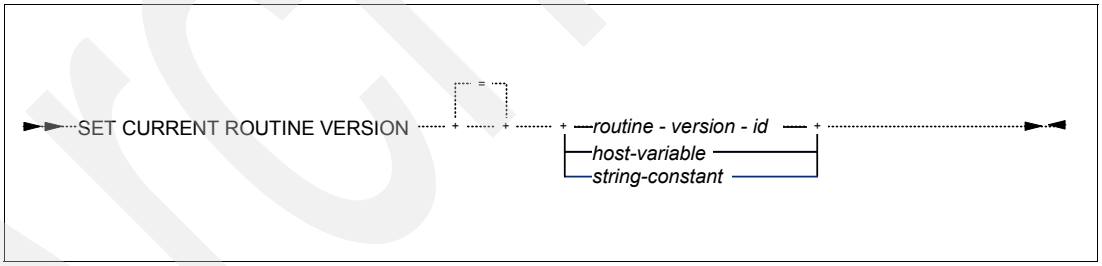

Figure 15-21 SET CURRENT ROUTINE VERSION syntax

If you want to reset the special register, specify an empty string constant, a string of blanks, or a host variable that is empty or contains only blanks. A routine version override is not in effect when the special register is reset.

If you set the CURRENT ROUTINE VERSION special register to a version identifier, it will affect all SQL procedures that are subsequently invoked using CALL statements that specify the name of the procedure using a host variable, until the value of CURRENT ROUTINE VERSION is changed. If a version of the procedure that is identified by the version identifier in the special register exists for an SQL procedure that is being invoked, that version of the procedure is used. Otherwise, the currently active version of the procedure (as noted in the catalog) is used.

A sample showing how to use this register is provided in 15.4, “Execution of a native SQL procedure” on page 299.

15.7 Error handling and debugging

The DDL for CREATE and ALTER PROCEDURE contains the option DEBUG MODE that determines whether the procedure can be debugged or not. This option only applies to native SQL and Java procedures.

Possible values are ALLOW DEBUG MODE, which basically enables the procedure for debugging with the Unified Debugger technology, as well as the values DISALLOW or DISABLE DEBUG MODE that prevent debugging capabilities on this procedure. The major difference between DISALLOW and DISABLE is that an ALTER PROCEDURE statement can be used to switch between DISALLOW and ALLOW option. However, once a procedure is defined with DISABLE DEBUG MODE, it can never be debugged in its lifetime. To change this status the procedure either has to be dropped and recreated, or a new version has to be added, since these options have a version granularity. Figure 15-22 depicts the possible state changes for the DEBUG MODE option. Note that once a procedure version is in DISABLE state, no transition to another state is possible in the version's lifecycle.

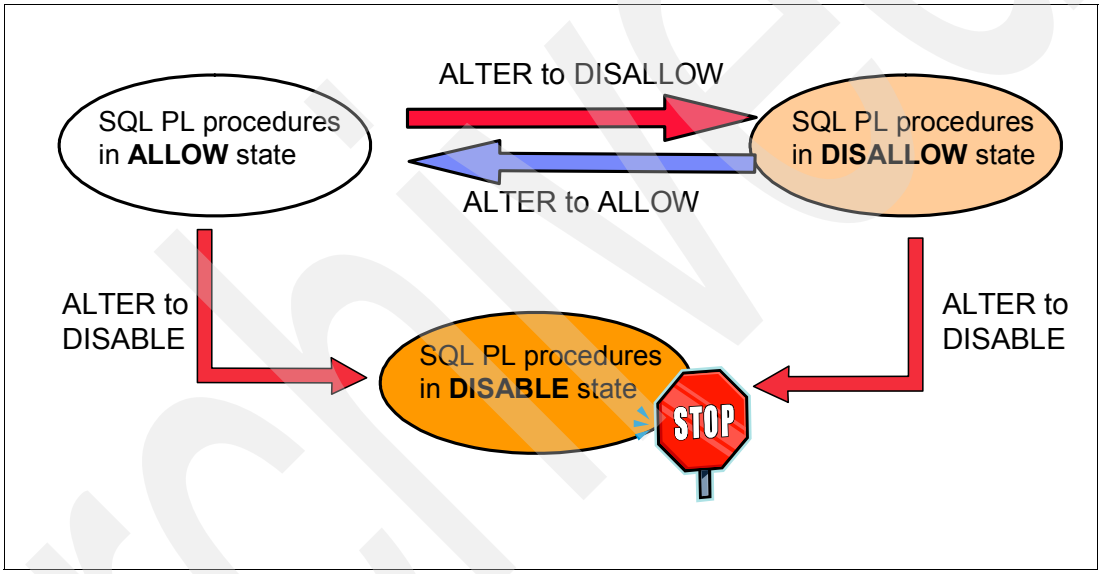

Figure 15-22 DEBUG MODE state diagram

The specified value is recorded in the DB2 catalog table SYSIBM.SYSROUTINES in DEBUG_MODE column. The default value for either ALLOW, DISALLOW or DISABLE when a procedure is created or altered, can be influenced with the help of the special register CURRENT DEBUG MODE. Example 15-35 shows how the MEDIAN_RESULT_SET procedure is created with DEBUG MODE ALLOW, by employing the special register.

Example 15-35 DEBUG MODE ALLOW

```
SET CURRENT DEBUG MODE = ALLOW#

CREATE PROCEDURE MEDIAN_RESULT_SET
   (OUT MEDIANSALARY DECIMAL(7,2))
 VERSION MEDIAN_V1
 LANGUAGE SQL
 READS SQL DATA
 DYNAMIC RESULT SETS 1
 WLM ENVIRONMENT FOR DEBUG MODE DB9AWLM
  BEGIN
    DECLARE V_NUMRECORDS INTEGER DEFAULT 1;
```

```
  DECLARE V_COUNTER INTEGER DEFAULT 0;
  DECLARE C1 CURSOR FOR
    SELECT SALARY FROM STAFF ORDER BY SALARY;
  DECLARE C2 CURSOR WITH RETURN FOR
    SELECT NAME, JOB, SALARY
       FROM STAFF
      WHERE SALARY > MEDIANSALARY
      ORDER BY SALARY;
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET MEDIANSALARY = 0;
  SELECT COUNT(*) INTO V_NUMRECORDS FROM STAFF;
  OPEN C1;
  WHILE V_COUNTER < (V_NUMRECORDS / 2 + 1) DO
    FETCH C1 INTO MEDIANSALARY;
    SET V_COUNTER = V_COUNTER + 1;
  END WHILE;
  CLOSE C1;
  OPEN C2;
END#
```

Note: The DEBUG MODE DISABLE setting is mainly used for security. A native SQL procedure that has been created as (or subsequently altered to) DEBUG MODE DISABLE can never be allowed to be debugged. Note that the ability to debug means the procedure source would be viewable to step through. It would also mean that SQL variable values could be changed during the debugging session which could cause data integrity issues. The purpose of DEBUG MODE DISABLE feature is to protect such unauthorized viewing of the source code and for the impact of the side effects from the debug sessions, for the SQL procedures that require such protection.

15.7.1 Compound statements within condition handlers

A very helpful debugging feature that is now supported in native SQL procedures is compound statements within condition handlers. Multiple statements can now be more easily written in a condition handler body by using a compound statement. Example 15-36 shows a schematic implementation of a compound statement, where multiple error values are materialized within the BEGIN - END block.

Example 15-36 Error handling with compound statement in condition handlers

```
BEGIN
     DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
     DECLARE PrvSQLState CHAR(5) DEFAULT '00000';
     DECLARE ExceptState INT;
     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
           SET PrvSQLState = . . .
           SET ExceptState = . . .
           . . .
        END;
  END
```

We recommend to update existing procedures to make use of the compound block technique. Before this language element was available, a programmer was able to simulate a compound block within the handler as illustrated in Example 15-37.

Example 15-37 Simulation of a compound block

```
CREATE PROCEDURE ...
  BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
       IF (1=1) THEN
           stmt1
           stmt2
       END IF;
    rest of procedure body
```

However, you should refrain from using this trick, because in native SQL procedures, SQLCODE and SQLSTATE values will be reset after executing the IF clause.

In 14.3, “Handling error conditions” on page 245 we provide a general introduction to condition handlers. Further examples can be found in “Using a compound statement in a condition handler declaration” on page 284.

15.7.2 GET DIAGNOSTICS

The GET DIAGNOSTICS SQL statement has been enhanced for use with native SQL stored procedures.

DB2_LINE_NUMBER, which is the new keyword on the GET DIAGNOSTICS statement, returns the line number where an error is encountered in parsing a dynamic statement. Also, it returns the line number where an error is encountered in parsing, binding, or executing a CREATE or ALTER statement for a native SQL procedure.

DB2_LINE_NUMBER also returns the line number when a CALL statement invokes a native SQL procedure and the procedure returns with an error. This information is not returned for an external SQL procedure. This value will only be meaningful if the statement source contains new line control characters, that is, the source is not converted to a single statement line when it gets processed. For example, in DSNTEP2 new line control characters are included when SQLPL is used as SQL format.

Stacked diagnostics area support

Starting with APAR PK43524 a stacked diagnostics area for the condition handler is supported. The STACKED DIAGNOSTICS provides the capability to remember the diagnostics information for an exception raised during the entire duration of the handling of that exception, even when the CURRENT diagnostics area can keep changing with the execution of the body of the handler.

The new keywords introduced are CURRENT and STACKED in the GET DIAGNOSTICS statement (where default is CURRENT). Therefore, the GET STACKED DIAGNOSTICS statement used within the body of a handler will look at the stacked diagnostics area instead of the current one. The SQL procedure in Example 15-38 divides the numerator by the denominator, which are both provided as input parameters. In case the denominator is set to 0 an error condition is raised, which is caught by the CONTINUE HANDLER. In the handler, the first GET DIAGNOSTICS statement obtains information from the current diagnostics area which contains the diagnostic information for the last SQL statement that was executed except for another GET DIAGNOSTICS statement. Therefore, MSG_TEXT is set to the diagnostic information concerning the statement SET DIVIDE_RESULT = -1. The second GET DIAGNOSTICS statement specifies the STACKED keyword. The use of the STACKED keyword allows access to the stacked diagnostics area, which contains the diagnostic information for the condition that caused the handler to be activated. Therefore,

DIVIDE_ERROR is set to the diagnostic information concerning the statement SET DIVIDE_RESULT = NUMERATOR / DENOMINATOR.

Example 15-38 GET STACKED DIAGNOSTICS

```
CREATE PROCEDURE DIVIDE_PROC(
       IN  NUMERATOR     INTEGER,
       IN  DENOMINATOR   INTEGER,
       OUT DIVIDE_RESULT INTEGER,
       OUT DIVIDE_ERROR  VARCHAR(70),
       OUT MSG_TEXT      VARCHAR(70) )
 VERSION V1
 LANGUAGE SQL
 READS SQL DATA

  BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN

      -- ERROR OCCURRED RETURN A -1 IN DIVIDE_RESULT
      SET DIVIDE_RESULT = -1;

      -- GET DIAGNOSTIC INFORMATION FOR SET STATEMENT
      GET CURRENT DIAGNOSTICS CONDITION 1 MSG_TEXT = MESSAGE_TEXT;

      -- GET DIAGNOSTIC INFORMATION ABOUT CONDITION THAT
      -- ACTIVATED THE HANDLER
      GET STACKED DIAGNOSTICS CONDITION 1 DIVIDE_ERROR = MESSAGE_TEXT;
    END;

    SET DIVIDE_RESULT = NUMERATOR / DENOMINATOR;

  END
```

15.7.3 Unified Debugger

Native SQL procedures that are defined with the option ALLOW DEBUG MODE can now be debugged with the Unified Debugger technology. For detailed information, refer to Chapter 28, “Tools for debugging DB2 stored procedures” on page 735.

15.8 Migrating external to native SQL procedures

Migrating an external to a native SQL procedure is fairly simple. Native SQL procedures support all statements that are also supported in the external SQL procedure body, even more. The only changes that have to be performed reside in the procedure options of the DDL.

If the external procedure has been defined with the options FENCED or EXTERNAL, these have to be deleted from the procedure options of a native SQL procedure. Furthermore when the keyword WLM ENVIRONMENT is specified in an external procedure, the FOR DEBUG clause has to be added for a native procedure. Otherwise, the WLM ENVIRONMENT clause needs to be deleted. For more information on migrating from external to native SQL procedures, refer to Chapter 14, “External SQL procedures” on page 233.

16

Debugging

In this chapter we discuss the categories of errors and approaches to resolving them, debugging options from classical batch to using the IBM Debug Tool, and take a look at an example of the PARAMETER STYLE SQL parameter and its usefulness. This chapter also removes the mystery of how to setup and get DB2 COBOL stored procedures operational using the IBM Debug Tool with Main Frame Interface (MFI), for z/OS only.

This chapter contains the following:

- SQL error categories
- Debugging options
- Classical debugging of stored procedures
- Compiler and LE options for debugging
- IBM Debug Tool
- GET DIAGNOSTICS

16.1 SQL error categories

There are five categories of errors when using stored procedures. We will discuss the SQL error codes that accompany each category. We will not be covering every conceivable error, just those that are common and some that are less common, but need awareness. In each category there is a table that refers to the SQLCODE, possible reasons for the error, and appropriate responses to correct the error. When reason codes and resource types are included with the error message, you can refer to *DB2 Version 9.1 for z/OS Messages,* GC18-9849 and *DB2 Version 9.1 for z/OS Codes,* GC18-9843 for more codes and details.

- **BIND** SQL errors: Errors that are recognized during the BIND process
- **Connectivity SQL errors**: Errors that appear at execution time and deal with the loss or lack of connectivity between the invoker and the stored procedure. Also included in this section are errors pertaining to an implementation where the invoker is on an LUW platform and the stored procedure being invoked is on the mainframe.
- **CALL statement SQLCODEs**: Errors that produce a non-zero SQLCODE when the invoker calls the stored procedure.
- **Invoking program, non-CALL SQL errors**: Errors that are related to having successfully executed a stored procedure call, but have a problem when referring to the parameters or other objects shared with the stored procedure.
- **Unhandled SQL errors to CALL statements**: Errors that may have been encountered in the stored procedure, but are not communicated to the caller, therefore creating unexpected conditions in the calling program.

16.1.1 BIND SQL errors

Because the information from the CREATE PROCEDURE statement is recorded in the DB2 catalog, the BIND process retrieves what it needs to validate the SQL CALL statement. Table 16-1 represents the types of errors that can be uncovered.

Table 16-1 BIND SQL errors

SQLCODE	REASON	RESPONSE
-440	Procedure name called and parameter list specified are not compatible. ► Routine-name was either incorrectly specified or does not exist in the database. ► A qualified reference was made, and the qualifier was incorrectly spelled. ► A user's current path does not contain the schema to which the desired function belongs, and an unqualified reference was used. ► The wrong number of arguments was included. ► The routine invoker is not authorized to execute the routine.	Fix the problem and retry. This can involve a change to the SQL statement, the addition of new routines or a change to the user's current path, a change to the execution privileges for the stored procedure.

16.1.2 Connectivity SQL errors

The errors listed in Table 16-2 reflect various types of connection problems from Logical Unit of Work issues to security violations for remote processing.

Table 16-2 Connectivity SQL errors

SQLCODE	REASON	RESPONSE
-114	THE LOCATION NAME location DOES NOT MATCH THE CURRENT SERVER ► A three -part SQL procedure name was provided for one of the following SQL statements: – ASSOCIATE LOCATORS – CALL – DESCRIBE PROCEDURE ► The first part of the SQL procedure name, which specifies the location where the stored procedure resides, did not match the value of the SQL CURRENT SERVER special register.	Take one of these actions to resolve the mismatch: ► Change the location qualifier to match the CURRENT SERVER special register. ► Issue an SQL CONNECT to the location where the stored procedure resides before issuing the SQL statement. Ensure that the SQL CALL statement is issued before the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE. ► Bind the package containing the three -part SQL procedure name with the BIND option DBPROTOCOL(DRDA). With this option, DB2 implicitly uses the DRDA protocol for remote access to the stored procedure. ► Correct the statements so that the exact syntax used to specify the procedure name on the CALL statement is the same as that on the ASSOCIATE LOCATOR and/or DESCRIBE PROCEDURE. – If an unqualified name is used to CALL the procedure, the one-part name must also be used on the other statements. – If the CALL statement is made with a three-part name, and the current server is the same as the location in the three-part name, the ASSOCIATE LOCATOR or DESCRIBE procedure can omit the location.
-426	DYNAMIC COMMIT NOT VALID AT AN APPLICATION SERVER WHERE UPDATES ARE NOT ALLOWED ► An application executing using DRDA protocols has attempted to issue a dynamic COMMIT statement, or a stored procedure has attempted to issue a COMMIT_ON_RETURN, while connected to a location at which updates are not allowed. ► A dynamic COMMIT or COMMIT_ON_RETURN can be issued only while connected to a location at which updates are allowed.	The IMS or CICS protocols should be used to commit work in these environments.

SQLCODE	REASON	RESPONSE
-842	A CONNECTION TO location-name ALREADY EXISTS One of the following situations occurred: ► A CONNECT statement identifies a location with which the application process has a private connection, using system-directed access. ► SQLRULES(STD) is in effect and a CONNECT statement identifies an existing SQL connection. ► A private connection, using system-directed access, cannot be established because of an existing SQL connection to that location. ► A CONNECT (type 2) request that includes the USER/USING clause identifies an existing SQL connection.	The correction depends on the error, as follows: ► If the location name is not the intended name, correct it. ► If SQLRULES(STD) is in effect and the CONNECT statement identifies an existing SQL connection, replace the CONNECT with SET CONNECTION or change the option to SQLRULES(DB2). ► If the CONNECT statement identifies an existing private connection, destroy that connection (by using the RELEASE statement in a previous unit of work) before executing the CONNECT statement. If the SQL statements following the CONNECT can be executed using system-directed access, an alternative solution is to change the application to use that method. ► If system-directed access cannot be used, destroy the conflicting SQL connection (by using the RELEASE statement in a previous unit of work) before executing the SQL statement that requires system-directed access. An alternative solution is to change the application so that only application-directed access is used. ► Destroy the connection (by using the RELEASE statement in a previous unit of work) before executing the CONNECT statement, which includes the USER/USING clause. Correct the error in the application, rebind the plan or package, and resubmit the job.
-925	COMMIT NOT VALID IN IMS, CICS OR RRSAF ENVIRONMENT ► An application executing in either an IMS or CICS environment or an application executing in an RRSAF environment when DB2 is not the only resource manager has attempted to execute a COMMIT statement. The SQL COMMIT statement cannot be executed in these environments.	The IMS, CICS, or RRS protocols should be used to commit work in these environments. ► If a stored procedure is being called from IMS or CICS, ensure that the stored procedure is not defined to perform a commit on return.
-926	ROLLBACK NOT VALID IN IMS, CICS OR RRSAF ENVIRONMENT ► An application executing in either an IMS or CICS environment or an application executing in an RRSAF environment when DB2 is not the only resource manager has attempted to execute a ROLLBACK statement. The SQL ROLLBACK statement cannot be executed in these environments.	The IMS, CICS, or RRS protocols should be used to roll back work in these environments.
-30082	CONNECTION FAILED FOR SECURITY REASON reason-code (reason-string) ► The attempt to connect to a remote database server was rejected due to invalid or incorrect security information. ► The cause of the security error is described by the reason-code and reason-string values. ► DB2 Version 9.1 for z/OS Codes, GC18-9843.	DB2 uses the communications database (CDB) to control network security functions. Make the appropriate changes to the CDB to correct the security failure.
-30090	An update operation or a dynamic commit or rollback was attempted at a server that was supporting an application that was in a read-only execution environment (IMS or CICS).	Do not attempt to update data or issue dynamic commits or rollbacks from IMS or CICS applications that are accessing remote data.

16.1.3 CALL statement error SQLCODEs

Table 16-3 contains information relating to the errors that can occur on the CALL statement.

Table 16-3 CALL statement error SQLCODEs

SQLCODE	REASON	RESPONSE
-430	routine-type routine-name (SPECIFIC NAME specific-name) HAS ABNORMALLY TERMINATED ► An abnormal termination has occurred while the routine routine-name (stored procedure or function) was in control.	The stored procedure or function needs to be fixed. Contact the author of the routine or your database administrator. Until it is fixed, the routine should not be used. See 16.2, “Debugging options” on page 328.
-440	NO routine-type BY THE NAME routine-name HAVING COMPATIBLE ARGUMENTS WAS FOUND ► This occurs in a reference to stored procedure routine-name, when DB2 cannot find a stored procedure it can use to implement the reference. ► There are several reasons why this could occur. – routine-name was either incorrectly specified or does not exist in the database. – A qualified reference was made, and the qualifier was incorrectly spelled. – A user’s current path does not contain the schema to which the desired function belongs, and an unqualified reference was used. – The wrong number of arguments were included. – The routine invoker is not authorized to execute the routine.	Fix the problem and retry. This could involve a change to the SQL CALL statement, the addition of new routines, or a change to the user’s current path.
-444	USER PROGRAM name COULD NOT BE FOUND DB2 received an SQL CALL statement for a stored procedure and found the row in the SYSIBM.SYSROUTINES catalog table associated with the requested procedure name. However, the MVS load module identified in the EXTERNAL_NAME column of the SYSIBM.SYSROUTINES row could not be found.	If the EXTERNAL_NAME column value in the SYSIBM.SYSROUTINES table is incorrect, use the ALTER PROCEDURE statement to correct the value. ► If the EXTERNAL_NAME column value is correct, use the MVS linkage editor to create the required MVS load module in one of the MVS load libraries used by your installation for stored procedures. ► This error can also occur if you are invoking a WLM-managed stored procedure that is not APF authorized, and the DB2 load libraries are not in the STEPLIB concatenation because they are being loaded from LINKLIST. – If you want the stored procedure program to run APF-authorized, linkedit it with AC=1 into an MVS APF authorized library. – If you do not want the stored procedure program to run APF authorized, add the DB2 load library to the STEPLIB concatenation of the JCL used to start the WLM-managed address space.

SQLCODE	REASON	RESPONSE
-450	USER-DEFINED FUNCTION OR STORED PROCEDURE name, PARAMETER NUMBER parmnum, OVERLAYED STORAGE BEYOND ITS DECLARED LENGTH. ► Upon return from a stored procedure name, DB2 has detected an overlay storage beyond a parameter's declared length. The parameter number is specified for a stored procedure or function. ► An example of this error would be if a decimal parameter in the invoking program was defined larger than the definition in the CREATE PROCEDURE parameter definition. ► Another example, two comparable parameters were referenced in the parameter list in the wrong order.	Check the calling parameter list sequence against the defined parameter list sequence and the procedure's parameter list sequence. If the sequence is correct, then check the data definitions of each parameter. Contact the author of the function/procedure or your database administrator. Until it is fixed, the function/procedure should not be used.
-470	SQL CALL STATEMENT SPECIFIED A NULL VALUE FOR INPUT PARAMETER number, BUT THE STORED PROCEDURE DOES NOT SUPPORT NULL VALUES. ► DB2 received an SQL CALL statement for a stored procedure and found a null value in the incoming parameter list. The stored procedure was defined in the SYSIBM.SYSROUTINES catalog table with PARAMETER_STYLE of GENERAL, which specifies that the routine does not accept null values. ► A call to a stored procedure with a LANGUAGE value of JAVA or COMPJAVA receives this SQLCODE if an input parameter in the compiled Java stored procedure has a Java base type that cannot be set to a null value. ► number: The parameter number from the ORDINAL field in SYSIBM.SYSPARMS.	If the stored procedure should not accept null values, change the calling application to provide a nonnull value. If the stored procedure should accept null values, use the ALTER PROCEDURE statement to change the PARAMETER STYLE of the stored procedure to be SQL or GENERAL WITH NULLS.

SQLCODE	REASON	RESPONSE

-471	INVOCATION OF FUNCTION OR PROCEDURE name FAILED DUE TO REASON rc
	A routine was invoked. The routine invocation was not accepted because of DB2 reason code rc.
	RC00E79001
	► DB2 received an SQL CALL statement for a stored procedure. The statement was not accepted because the routine was stopped.
	► Possible reasons are:
	– the STOP PROCEDURE ACTION(REJECT) command was issued for this procedure, or
	– the STOP FUNCTION ACTION(REJECT) command was issued for this user-defined function, or
	– there was a **previous abnormal termination of the routine**.
	
	RC00E79002
	► The CALL statement was not accepted because the procedure could not be scheduled before the installation-defined time limit expired.
	► Reasons:
	– The DB2 STOP PROCEDURE(name) command was in effect.
	– The dispatching priority assigned by WLM to the caller of the user-written routine was too low, which resulted in WLM not assigning the request to a TCB before the time limit expired.
	– The user-written routine could not be assigned to a TCB in the DB2-established stored procedures address space in the required time interval, because all available stored procedure TCBs were in use.
	All other reason codes:
	Refer to *DB2 Version 9.1 for z/OS Codes,* GC18-9843 for the meanings and suggested response activities.
		RC00E79001
		► If the user-written routine was stopped by an abnormal termination, correct the cause of the abnormal termination and,
		► Use the -START PROCEDURE command.
		
		RC00E79002
		► If the routine was stopped, issue a DB2 START PROCEDURE command.
		► If the WLM application environment is quiesced, issue the MVS command
		`WLM DISPLAY,APPLENV=wlmenv`
		to verify the status of the application environment. Then the MVS command
		`WLM VARY APPLENV=wlmenv,RESUME`
		can be used to activate the environment if it is quiesced.
		► For TCB management, contact your DB2 administrator to raise the dispatching priority of the procedure.

-577	object-type object-name ATTEMPTED TO MODIFY DATA WHEN THE DEFINITION OF THE FUNCTION OR PROCEDURE DID NOT SPECIFY THIS ACTION The current environment does not allow SQL statements that modify data. One of the following situations has occurred: ► A user-defined function or stored procedure object-name was invoked and attempted to modify data, but the function or procedure was defined without the MODIFIES SQL option. ► In an environment of nested functions and procedures, the SQL option in effect is the most restrictive one that has been specified in the nested hierarchy. The SQL data access option in effect does not allow for modifying the data.	Either: ► Use an ALTER statement to change the definition of the function or procedure to allow statements that modify data, or ► Remove the failing SQL statement from the procedure.

SQLCODE	REASON	RESPONSE
-729	A STORED PROCEDURE SPECIFYING COMMIT ON RETURN CANNOT BE THE TARGET OF A NESTED CALL STATEMENT A stored procedure defined with the COMMIT ON RETURN attribute was called from a stored procedure, user-defined function, or trigger. Stored procedures defined with COMMIT ON RETURN cannot be nested in this way.	The SQL statement is not executed. If the CALL statement references a remote server, the unit of work is placed in a must rollback state. Remove the CALL to the stored procedure that was defined with the COMMIT ON RETURN attribute.
-751	object-type object-name (SPECIFIC NAME specific name) ATTEMPTED TO EXECUTE AN SQL STATEMENT statement THAT IS NOT ALLOWED A stored procedure issued an SQL statement that forced the DB2 thread to roll back the unit of work. The SQL statement that caused the thread to be placed in the MUST_ROLLBACK state is one of the following: ► COMMIT ► ROLLBACK All further SQL statements are rejected until the SQL application that issued the SQL CALL statement rolls back the unit of work. Remotely called stored procedures cannot execute embedded SQL Commit and/or Rollback statements unless: ► The connection with the requester system uses one phase commit protocols ► The requester system indicates that commits are allowed (through sending a DRDA RDBCMTOK=TRUE indication) when the stored procedure is called. ► **Note:** For DB2 Connect™ requester systems, this requires that the client application must use Connect Type 1, or Remote Unit of Work connections. Connect Type 2 or Distributed Unit of Work connections will cause DB2 Connect to indicate that commits are not allowed, thus embedded SQL Commit and/or Rollback statements in a stored procedure will fail.	Any Commit or Rollback statements in the stored procedure must be removed, or the client application should be modified to establish an environment that allows the stored procedure to execute SQL Commit and/or Rollback statements. When control returns to the SQL application that issued the SQL CALL statement, the SQL application must roll back the unit of work. This can be done by issuing an SQL ROLLBACK statement or the equivalent IMS or CICS operation.

16.1.4 Invoking program, non-CALL SQL errors

You have executed the CALL statement, the stored procedure returns to your invoking program with an SQLCODE of zero and subsequent actions generate a non-zero SQLCODE.

What can go wrong? Table 16-4 contains information about errors that are not detected until statements that are dependent on the call execute. Most of these errors are “linkage” in nature, and occur when referring to parameters or other objects shared with the stored procedure.

Table 16-4 Non-CALL SQL errors

SQLCODE	REASON	RESPONSE
-423	INVALID VALUE FOR LOCATOR IN POSITION position-# ► The value specified in a result set locator host variable, a LOB locator host variable, or a table locator that is specified at position-# in the locator variable list of the SQL statement does not identify a valid result set locator, LOB locator variable, or table locator, respectively.	For a result set locator there are two common causes for the error: ► The host variable used as a result set locator was never assigned a valid result set locator value. Result set locator values are returned by the DESCRIBE, PROCEDURE, and ASSOCIATE LOCATORS statements. Make sure the value in your host variable is obtained from one of these statements. ► Result set locator values are only valid as long as the underlying SQL cursor is open. If a commit or rollback operation closes an SQL cursor, the result set locator associated with the cursor is no longer valid. For an LOB locator, some common causes for the error are: ► The host variable used as a LOB locator was never assigned a valid LOB value. ► A commit or rollback operation or an SQL FREE LOCATOR statement freed the locator. For a table locator, the error commonly occurs when the host variable that was used as a table locator was never assigned a valid table locator value.
-482	THE PROCEDURE procedure-name RETURNED NO LOCATORS ► The procedure identified in an ASSOCIATE LOCATORS statement returned no result set locators.	Determine if result set locators are returned from the identified procedure by using the DESCRIBE PROCEDURE statement.
-496	THE SQL STATEMENT CANNOT BE EXECUTED BECAUSE IT REFERENCES A RESULT SET THAT WAS NOT CREATED BY THE CURRENT SERVER ► The SQL statement cannot be executed because the current server is different from the server that called a stored procedure. The SQL statement can be any of the following: – `ALLOCATE CURSOR` – `DESCRIBE CURSOR` – `FETCH, with an allocated cursor` – `CLOSE, with an allocated cursor`	Connect to the server that called the stored procedure, which created the result set before running the SQL statement that failed.

SQLCODE	REASON	RESPONSE
-499	CURSOR cursor-name HAS ALREADY BEEN ASSIGNED TO THIS OR ANOTHER RESULT SET FROM PROCEDURE procedure-name. ► An attempt was made to assign a cursor to a result set using the SQL statement ALLOCATE CURSOR and one of the following applies: ► The result set locator variable specified in the ALLOCATE CURSOR statement has been previously assigned to cursor cursor-name. ► Cursor cursor-name specified in the ALLOCATE CURSOR statement has been previously assigned to a result set from stored procedure procedure-name.	Determine if the target result set named in the ALLOCATE CURSOR statement has been previously assigned to a cursor. ► If the result set has been previously assigned to cursor cursor-name, then either choose another target result set or call the stored procedure again and reissue the ASSOCIATE LOCATOR and ALLOCATE CURSOR statements. ► If the result set has not been previously assigned to a cursor, the cursor cursor-name specified in the ALLOCATE CURSOR statement has been previously assigned to some result set from stored procedure procedure-name. You cannot assign cursor cursor-name to another result set, so you must specify a different cursor name in the ALLOCATE CURSOR statement. Correct the statements so that the exact syntax used to specify the procedure name on the CALL statement be the same as that on the ASSOCIATE LOCATOR and/or DESCRIBE PROCEDURE. ► If an unqualified name is used to CALL the procedure, the one-part name must also be used on the other statements. ► If the CALL statement is made with a three-part name, and the current server is the same as the location in the three-part name, the ASSOCIATE LOCATOR or DESCRIBE procedure can omit the location.

SQLCODE	REASON	RESPONSE
-504	CURSOR NAME *cursor-name* IS NOT DECLARED Cursor *cursor-name* was referenced in an SQL statement, and one of the following is true: ► Cursor *cursor-name* was not declared (using the DECLARE CURSOR statement) or allocated (using the ALLOCATE CURSOR statement) in the application program or SQL routine before it was referenced. ► Cursor *cursor-name* was referenced in a positioned UPDATE or DELETE statement which is not a supported operation for an allocated cursor. ► Cursor *cursor-name* was allocated, but a CLOSE cursor statement naming cursor-name was issued and deallocated the cursor before this cursor reference. ► Cursor *cursor-name* was allocated, but a ROLLBACK operation occurred and deallocated the cursor before this cursor reference. ► Cursor *cursor-name* was allocated, but its associated cursor declared in a stored procedure was not declared WITH HOLD, and a COMMIT operation occurred and deallocated the cursor before this cursor reference. The COMMIT operation can be either explicit (the COMMIT statement) or implicit (that is, a stored procedure defined as COMMIT_ON_RETURN = 'Y' was called before this cursor reference). ► Cursor *cursor-name* was allocated, but its associated stored procedure was called again since the cursor was allocated, new result sets were returned, and cursor *cursor-name* was deallocated. ► The declaration of a cursor *cursor-name* was not in scope for the reference to a cursor named *cursor-name*.	Check the application program or SQL routine for completeness and for a possible spelling error in the cursor declaration or allocation. The declaration for or allocation of a cursor must appear in an application program or SQL routine before SQL statements that reference the cursor. If the *cursor-name* was <UNKNOWN>, then the cursor was not successfully declared or allocated. This error can occur if SQL(DB2) was used, and a warning message was issued during precompilation. Check the precompile output for warning messages on the DECLARE CURSOR or ALLOCATE CURSOR statement, and correct the statement. For an allocated cursor, if an implicit or explicit COMMIT, ROLLBACK, or CLOSE occurred since the cursor was successfully allocated, modify the application program logic to do one of the following actions: ► After the COMMIT, ROLLBACK, or CLOSE operation, call the associated stored procedure again, and reissue the ASSOCIATE LOCATORS and ALLOCATE CURSOR statements. ► For COMMIT, declare the associated cursor in the stored procedure WITH HOLD so the COMMIT operation will not deallocate the cursor. For an allocated cursor, if the associated stored procedure was called again, and new result sets were returned since the cursor was allocated, reissue the ASSOCIATE LOCATORS and ALLOCATE CURSOR statements.

16.1.5 Unhandled SQL errors to CALL statements

In this section we discuss handling SQL errors that may have been encountered in the stored procedure, but not communicated to the caller, therefore creating unexpected conditions in the calling program.

The most prevalent of unhandled errors is the scenario when a stored procedure *cannot* take corrective action because it abnormally terminated. The invoker receives an SQLCODE of -430 from the CALL statement. Since the procedure abended, it was, obviously, unable to handle the error condition, and the invoker cannot rely on any information contained in the output parameters. Once the stored procedure has been corrected, it might be necessary to issue a -START PROCEDURE statement if the procedure was placed in the STOPABN state because the maximum number of abends have been reached for the address space.

Another common unhandled error is when a stored procedure receives SQLCODE of +100 on initial fetch, no rows found, and fails to place a default value into the output parameters. Depending upon the definition of the parameters, the invoking program can receive SQLCODE -310 (invalid decimal data) or -180 (invalid date, time, or timestamp value) on the CALL. If the programs included SQLCODE and/or SQLSTATE parameters, the invoking

program can know that the stored procedure did not locate a row, and process this data as such.

It is important for every stored procedure to correctly handle encountered SQL errors. What is correct is dictated by the needs of the application. Communication, though, is the key to handling most application errors. If the stored procedure determines that a process cannot continue as expected, this fact must be reported to the caller.

At a minimum, the stored procedure needs to share its SQLCODE and or SQLSTATE values and an appropriate error message. Remember, just because the SQLCODE from the CALL statement is zero, it does not mean that the stored procedure accomplished its mission.

PARAMETER STYLE SQL

One technique to simplify the CALL statement parameters to be shared for describing errors encountered is the use of PARAMETER STYLE SQL. This parameter style is specified in the CREATE PROCEDURE statement and causes four additional parameters to be sent to the stored procedure program, one of which is an output SQLSTATE. Table 16-5 describes the additional parameters.

Table 16-5 PARAMETER STYLE SQL additional parameters

SQL parameter	Meaning
SQLSTATE	The SQLSTATE that is to be returned to DB2. This is a CHAR(5) parameter that can have the same values as those that are returned from a user-defined function.
QUALIFIED PROCNAME	The qualified name of the stored procedure. This is a VARCHAR(517) value.
SPECIFIC PROCNAME	The specific name of the stored procedure. The specific name is a VARCHAR(128) value that is the same as the unqualified name.
DIAGNOSTIC MESSAGE	The SQL diagnostic string that is to be returned to DB2. This is a VARCHAR(70) value. Use this area to pass descriptive information about an error or warning to the caller.

Important: The parameters mentioned in Table 16-5, *must* be referenced in the linkage definition area of the stored procedure source program.

The additional parameters associated with PARAMETER STYLE SQL are *not* defined or referenced in the invoking program or in the CREATE PROCEDURE statement. A COBOL example of the stored procedure definitions would be:

1. Define the program/stored procedure parameters.
2. Define a separate and independent indicator variable for each parameter.
3. Establish linkage with the USING clause of the PROCEDURE DIVISION header and the sequence of the parameters. See Example 16-1.

Example 16-1 Sample referencing the additional parameters

```
COBOL:
LINKAGE SECTION.
01 PARM1 ....
01 PARM2 ,,,,
```

```
01 PARM3 ...
01 IV-PARM1 ...
01 IV-PARM2 ...
01 IV-PARM3 ,,,
01 SQL-SQLSTATE                                PIC    X(05).
01 SQL-QUAL-PROCNAME.
     49 SQL-QUAL-PROCNAME-LEN       PIC    S9(04)   COMP.
     49 SQL-QUAL-PROCNAME-TEXT     PIC    X(517).
01 SQL-SPEC-PROCNAME.
     49 SQL-SPEC-PROCNAME-LEN       PIC    S9(04)   COMP.
     49 SQL-SPEC-PROCNAME-TEXT     PIC    X(128).
01 SQL-DIAGNOSTICS.
     49 SQL-DIAGNOSTICS-LEN             PIC    S9(04)   COMP.
     49 SQL-DIAGNOSTICS-TEXT           PIC    X(70).

PROCEDURE DIVISION USING PARM1, PARM2, PARM3,
                                   IV-PARM1, IV-PARM2, IV-PARM3,
                 SQL-STATE, SQL-QUAL-PROCNAME,
                 SQL-SPEC-PROCNAME, SQL-DIAGNOSTICS.
```

For examples in other languages, refer to *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

If the stored procedure program sets the SQL-SQLSTATE parameter, then DB2 will set the SQLCODE for the invoking call statement to a negative value. To see what values will be placed in the caller's SQLSTATE and SQLCODE fields, refer to 10.2.6, "Handling PARAMETER STYLE SQL" on page 119.

16.1.6 Miscellaneous negative SQLCODEs

There are two additional negative SQLCODEs that you should be aware of. One is -913 and the other is -805.

Most program designs that check for time-out and deadlocks will interrogate the SQLCODE for a value of -911. Stored procedures that encounter this condition and are "victimized" will be notified with an SQLCODE of -913. This means that DB2 did not execute a ROLLBACK for the application and the application needs to rollback or commit as soon as possible.

An SQLCODE of -805 when executing a stored procedure can occur for one of two reasons:

- Either DB2 cannot find the package for the application that contains the SQL CALL statement,
- or DB2 cannot find the package for the stored procedure being called.

If the -805 is returned, this should be a lot more specific to stored procedures. The first task is to determine if it is the CALL statement package that cannot be found, or if the call statement got to the stored procedure program and the stored procedure package cannot be found. The actions are different to resolve each. Often the stored procedure program is changed and rebound, so the timestamp does not match unless a WLM refresh is done.

-913

UNSUCCESSFUL EXECUTION CAUSED BY DEADLOCK OR TIMEOUT. REASON CODE reason-code, TYPE OF RESOURCE resource-type, AND RESOURCE NAME resource-name

Explanation: The application was the victim in a deadlock or experienced a time-out. The reason code indicates whether a deadlock or time-out occurred.

SQLERRD(3) also contains the reason-code which indicates whether a deadlock or time-out occurred. The most common reason codes are:

00C90088 - deadlock
00C9008E - time-out

Response: The application should either commit or roll back to the previous COMMIT. Then, generally, the application should terminate. See message DSNT376I in *DB2 Version 9.1 for z/OS Messages,* GC18-9849, for possible ways to avoid future deadlocks or time-outs.

-805

`DBRM OR PACKAGE NAME location-name.collection-id.dbrmname.consistency.token NOT FOUND IN PLAN plan-name. REASON reason`

Explanation: An application program attempted to use a DBRM or package `location-name.collection-id.dbrmname.consistency-token` that was not found. The collection ID is blank (`location-name.dbrmname.consistency-token`) if the CURRENT PACKAGESET special register was blank for the local program execution.

The REASON token is blank if the length of “location-name” is 16, the length of “collection-id” is 18, and the length of “dbrm-name” is 8 due to the length of SQLERRMT.

The DBRM or package name was not found for one or more of the following reasons:

- 01: The DBRM name was not found in the member list of the plan and there is no package list for the plan. Refer to the first SQL statement under problem determination for assistance in determining the problem.

  The package name was not found because there is no package list for the plan. Refer to the second SQL statement under Problem Determination for assistance in determining the problem.

- 02: The DBRM name `dbrm-name` did not match an entry in the member list or the package list. Any of the following conditions could be the problem:

  Bind conditions:

  – The collection-id in the package list was not correct when the application plan plan-name was bound. Refer to the second SQL statement under Problem Determination for assistance in determining the problem.

  – The location-name in the package list was not correct when the application plan-name was bound. Refer to the second SQL statement under Problem Determination for assistance in determining the problem.

  – The location-name in the CURRENTSERVER option for the bind subcommand was not correct when the application plan plan-name was bound. Refer to the third SQL statement under Problem Determination for assistance in determining the problem.

  – DB2 private protocols are not supported under the bind parameter.

  Application conditions:

  – The CURRENT PACKAGESET special register was not set correctly by the application.
  – The application was not connected to the proper location.

When using SET CURRENT PACKAGESET = :HV, be sure to use the correct encoding scheme, that matches with the :HV in ZPARM options. This statement does not require package or DBRM bound into the plan, so it uses the encoding scheme defined for system. The same applies to SET CURRENT PACKAGE PATH.

- 03: The DBRM name dbrm-name matched one or more entries in the package list and the search of those entries did not find the package. The conditions listed under reason 02 or the following conditions might be the problem.

  The DBRM of the version of the application program being executed was not bound (A package with the same consistency token as that of the application program was not found.) Refer to the fourth and fifth SQL statements under the Problem Determination section.

  The incorrect version of the application program is being executed.

- 04: The package collection-id.dbrm-name.consistencytoken does not exist at the remote site, location-name. Refer to the fifth SQL statement under the Problem Determination section.

In a native SQL procedure, if the affected SQL statement follows a SET CURRENT PACKAGESET, SET CURRENT PACKAGE PATH, CONNECT statement, or if it refers to an object on a remote server, additional package or packages will need to be bound via BIND COPY. Whenever the native SQL procedure is changed such that a regeneration is needed, the additional package also needs to be bound with the copy option.

Response: Based on the above reasons, the programmer can perform one or more of the following operations for each reason to correct the error:

- 01: Add the DBRM name dbrm-name to the MEMBER list of the BIND subcommand and bind the application plan plan-name,

  or,

  Add the PKLIST option with the appropriate package list entry to the REBIND subcommand and rebind the application plan plan-name.

- 02: Correct the dbrm-name of the entry in the PKLIST option and use the REBIND subcommand to rebind the application plan plan-name, or correct the location-name of the entry in the PKLIST option and use the REBIND subcommand to rebind the application plan plan-name, or correct the location-name in the CURRENTSERVER option and use the REBIND subcommand to rebind the application plan plan-name, or set the CURRENT PACKAGESET special register correctly, or Connect to the correct location name.

- 03: All the operations under reason 02 above might fix the problem, plus the following operations.

  Correct the collection-id of the entry in the PKLIST option and use the REBIND subcommand to rebind the application plan plan-name, or bind the DBRM of the version of the application program to be executed into the collection collection-id, or execute the correct version of the application program. The consistency token of the application program is the same as the package that was bound.

- 04: According to *DB2 Version 9.1 for z/OS Codes,* GC18-9843 all the operations under reason 02 and 03 might fix the problem.

If this error is issued for one of the situations described for a native SQL procedure, use BIND COPY to bind the additional packages.

Problem Determination: The following queries aid in determining the problem. Run these queries at the local location:

- This query displays the DBRMs in the member list for the plan. If no rows are returned, then the plan was bound without a member list:

```
SELECT PLCREATOR, PLNAME, NAME, VERSION
FROM SYSIBM.SYSDBRM
WHERE PLNAME = 'plan-name';
```

- This query displays the entries in the package list for the plan. If no rows are returned, then the plan was bound without a package list:

```
SELECT LOCATION, COLLID, NAME
FROM SYSIBM.SYSPACKLIST
WHERE PLANNAME = 'plan-name';
```

- This query displays the CURRENTSERVER value specified on the BIND subcommand for the plan:

```
SELECT NAME, CURRENTSERVER
FROM SYSIBM.SYSPLAN
WHERE NAME = 'plan-name';
```

- This query displays if there is a matching package in SYSPACKAGE. If the package is remote, put the location name in the FROM clause. If no rows are returned, the correct version of the package was not bound:

```
SELECT COLLID, NAME, HEX(CONTOKEN), VERSION
FROM <location-name.>SYSIBM.SYSPACKAGE
WHERE NAME = 'dbrm-name'
AND HEX(CONTOKEN) = 'consistency-token';
```

- This query displays if there is a matching package in SYSPACKAGE. If the package is remote, put the location name in the FROM clause. Use this query when collection-id is not blank. If no rows are returned, the correct version of the package was not bound:

```
SELECT COLLID, NAME, HEX(CONTOKEN), VERSION
FROM <location-name.>SYSIBM.SYSPACKAGE
WHERE NAME = 'dbrm-name'
AND HEX(CONTOKEN) = 'consistency-token'
AND COLLID = 'collection-id';
```

16.2 Debugging options

In this chapter, we examine classical debugging techniques and debugging options using the IBM Debug Tool on z/OS using a COBOL example. Our COBOL example using the IBM Debug Tool also applies to PL/I and C/C++ language stored procedures. The other debugging options for other languages and other platforms are discussed in Chapter 28, “Tools for debugging DB2 stored procedures” on page 735.

16.3 Classical debugging of stored procedures

We have all had occasions when our program has abnormally terminated with a system completion code and needed to be *debugged*. There have been times when it was necessary to roll up our sleeves and hunt for the data to evaluate values and addresses. There have also been times when we had to search through a sysudump for that *needle in a haystack.* Well, stored procedures that are written to run on the DB2 for z/OS server have many options available for debugging. Before delving into the tools available for assisting you with your debugging efforts, let us take a look at the *classical* approach to debugging.

16.3.1 Invoking program receives SQLCODE of -430

SQLCODE of -430 means that the CALL statement successfully *invoked* the stored procedure but the procedure abnormally terminated. Example 16-2 includes all of the displays from the invoking program, program name, contents of fields, SQLCODE from the procedure, and the DSNTIAR produced error information.

Example 16-2 Program produced displays

```
******************************** TOP OF DATA ********************************
++ BONNIC1C STARTING ++
WS-TIMESTAMP = 2003-12-03-19.37.17.697857
PEMPNO = 000250
WS-SQLCODE = - 430
 DSNT408I SQLCODE = -430, ERROR:  PROCEDURE DEVL7083.PRGTYPE1 (SPECIFIC NAME
          DEVL7083.PRGTYPE1) HAS ABNORMALLY TERMINATED
 DSNT418I SQLSTATE   = 38503 SQLSTATE RETURN CODE
 DSNT415I SQLERRP    = DSNX9CAC SQL PROCEDURE DETECTING ERROR
 DSNT416I SQLERRD    = 0  0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
 DSNT416I SQLERRD    = X'00000000'  X'00000000'  X'00000000'  X'FFFFFFFF'
          X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
******************************** BOTTOM OF DATA ********************************
```

Figure 16-1 shows the console messages that accompany the abend of the stored procedure. You may see message DSNX905I, DSNX906I, or DSNX966I, depending on the circumstances.

```
DSNX906I  -DB2G DSNX9CAC PROCEDURE OR FUNCTION 224
DEVL7083.PRGTYPE1           TERMINATED ABNORMALLY. THE PROCEDURE OR
FUNCTION HAS BEEN STOPPED. ASID= 03EB WLM_ENV= DB2GDEC1
-                                        --TIMINGS (MINS.)--
   ----PAGING COUNTS---
-JOBNAME  STEPNAME PROCSTEP    RC   EXCP    CPU    SRB  CLOCK   SERV
PG   PAGE   SWAP    VIO SWAPS
-PAOLOR53          RUN         08    272    .00    .00   3.18   2325
 0      0      0      0     0
IEF404I PAOLOR53 - ENDED - ASID=0034 - SC63
-PAOLOR53 ENDED.  NAME-RUN BONNIC1C         TOTAL CPU TIME=   .00
TOTAL ELAPSED TIME=  3.18
$HASP395 PAOLOR53 ENDED
```

Figure 16-1 Console messages for abend that resulted in SQLCODE -430

16.3.2 Searching out reasons the stored procedure abnormally terminated

Here is a testing strategy to consider for determining the cause of logic or abend errors in the stored procedure:

1. If the procedure contains DISPLAY statements, they will be written to the SYSOUT data set of the job that creates the WLM-established stored procedures address space. If you would like to separate your displays from the debugging output, use the MSGFILE(ddname,,,,ENQ) LE runtime option to specify a ddname for the LE debugging messages. Example 16-3 highlights the MSGFILE and WLM name that will be used throughout this discussion. For details on the RUN options, see 5.2, “Language Environment runtime options” on page 48.

   Example 16-3 Sample CREATE with LE runtime options

```
CREATE PROCEDURE PRGTYPE1
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME    VARCHAR(12)
,OUT PMIDINIT     CHAR(1)
,OUT PLASTNAME    VARCHAR(15)
,OUT PWORKDEPT    CHAR(3)
```

```
,OUT PHIREDATE   DATE
,OUT PSALARY     DEC(9,2)
,OUT PSQLCODE    INTEGER
,OUT PSQLSTATE   CHAR(5)
,OUT PSQLERRMC   VARCHAR(250)
,OUT PCALLCTR    DECIMAL(5,0)
)
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
WLM ENVIRONMENT DB2GDEC1
COLLID DEVL7083
PROGRAM TYPE MAIN
RUN OPTIONS 'MSGFILE(SYSDBOUT,,,,ENQ)'
COMMIT ON RETURN NO;
```

2. Compile the stored procedure with the option 'TEST(SYM)' if you would like to have a formatted local variable dump included in the CEEDUMP output of the stored procedure. For more details on this compile option, refer to *Enterprise COBOL for z/OS Programming Guide Version 3 Release 4*, SC27-1412-05.
3. Locate the stored procedure output, CEEDUMP, and program displays by:
   a. Using System Display and Search Facility (SDSF), with a prefix equal to the WLM environment name, enter one of the following commands:

      ```
      DA to display active or, ST for the status of, STC (Started Tasks)
      ```

      Example 16-4 accesses the Job Data Set panel by coding ? in the NP field.

      Example 16-4 SDSF ST display

      ```
      SDSF STATUS DISPLAY ALL CLASSES                          LINE 73-78 (78)
      COMMAND INPUT ===>                                          SCROLL ===> CSR
      PREFIX=DB2GDEC1  DEST=(ALL)  OWNER=*  SYSNAME=
      NP   JOBNAME   JobID    Owner    Prty Queue      C  Pos  SAff  ASys Status
       ?   DB2GDEC1  STC09147 STC         1 PRINT      3179
      ```

   b. Example 16-5 shows the Job Data Set Display (JDSD) accessed, and the selection of the SYSDBOUT data set referenced in the runtime options of Example 16-3.

      Example 16-5 Job Data Set Display

      ```
      SDSF JOB DATA SET DISPLAY - JOB DB2GDEC1 (STC09147)    LINE 1-5 (5)
       COMMAND INPUT ===>                                       SCROLL ===> CSR
       PREFIX=DB2GDEC1  DEST=(ALL)  OWNER=*  SYSNAME=
       NP   DDNAME   StepName ProcStep DSID Owner    C Dest              Rec-Cnt Page
            JESMSGLG JES2                2 STC      S LOCAL                  18
            JESJCL   JES2                3 STC      S LOCAL                  22
            JESYSMSG JES2                4 STC      S LOCAL                  27
            SYSOUT   DB2GDEC1          101 STC      S LOCAL                   1
        S   SYSDBOUT DB2GDEC1          101 STC      S LOCAL                   3
            CEEDUMP  DB2GDEC1          102 STC      S LOCAL                 548
      ```

   c. With Example 16-6, the stored procedure PRGTYPE1 terminated with a S0C7 at statement 331.

Example 16-6 Message in SYSDBOUT data set

```
CEE3207S The system detected a data exception (System Completion Code=0C7).
          From compile unit PRGTYPE1 at entry point PRGTYPE1 at statement 331 at
compile unit offset +000005D4 at entry offset +000005D4 at address 0001BC6C.
```

d. Next, select the **CEEDUMP** data set. Example 16-7 highlights, most importantly, the data content of the field in error, PCALL-CTR. The local variables appear in the CEEDUMP output because of the TEST(SYM) compile option.

e. Use the compiler listing to determine exactly what statement was executing at the time of the error.

Example 16-7 CEEDUMP output

```
CEE3DMP V1 R4.0: Condition processing resulted in the unhandled condition.        12/03/03
7:37:19 PM                  Page:    1

 Information for enclave PRGTYPE1

Condition Information for Active Routines
     Condition Information for PRGTYPE1 (DSA address 21CE01C0)
       CIB Address: 21CE0C68
       Current Condition:
         CEE3207S The system detected a data exception (System Completion Code=0C7).
       Location:
         Program Unit: PRGTYPE1 Entry: PRGTYPE1 Statement: 331 Offset: +000005D4

CEE3DMP V1 R4.0: Condition processing resulted in the unhandled condition.        12/03/03
7:37:19 PM                  Page:    7
Local Variables:
              287 01 PHIREDATE        X(10) DISP       '........'
              288 01 PSALARY          S9(7)V99 CMP3    *** Invalid data for this data type
*** Hex 0000000000
              289 01 PSQLCODE         S9(9) COMP       +0000000000
              290 01 PSQLSTATE        X(5) DISP        '.....'
              291 01 PSQLERRMC        AN-GR
              292  02 PSQLERRMC-LEN    S9999 COMP       +00000
              293  02 PSQLERRMC-TEXT   X(250) DISP
'..................................................
 ..................................................
              294 01 PCALL-CTR        S9(5) CMP3       *** Invalid data for this data type
*** Hex 000000
```

4. Repeating step 3 on page 330, locate the compile SYSPRINT for the stored procedure. As shown in Example 16-8, you now have the failing statement in the stored procedure source code.

Example 16-8 Compile SYSPRINT information

```
PP 5655-G53 IBM Enterprise COBOL for z/OS and OS/390 3.2.0
Invocation parameters:
QUOTE,NORENT,OFFSET,MAP,TEST(SYM),PGMNAME(LONGUPPER)
.
.
000331         012600          ADD 1 TO  PCALL-CTR
.
.
```

16.3.3 Reasons why the stored procedure abended

After analyzing the source code, you would have discovered that the PCALL-CTR field was defined in the LINKAGE SECTION of the stored procedure. Example 16-9 highlights the Linkage and the PROCEDURE DIVISION USING clause. This also confirms that PCALL-CTR is a parameter.

Example 16-9 LINKAGE SECTION of PRGTYPE1

```
LINKAGE SECTION.
01  PEMPNO        PIC X(6).
01  PFIRSTNME.
    49  PFIRSTNME-LEN     PIC S9(4) COMP.
    49  PFIRSTNME-TEXT    PIC X(12).
01  PMIDINIT      PIC X(1).
01  PLASTNAME.
    49  PLASTNAME-LEN     PIC S9(4) COMP.
    49  PLASTNAME-TEXT    PIC X(15).
01  PWORKDEPT     PIC X(3).
01  PCALL-CTR     PIC S9(05)        COMP-3.
PROCEDURE DIVISION USING PEMPNO, PFIRSTNME,PMIDINIT,
                         PLASTNAME, PWORKDEPT, PCALL-CTR.
```

The data shown in Example 16-7 on page 331 was indicative of an uninitialized value.

PCALL-CTR was defined as an output parameter according to the CREATE statement in Example 16-3 on page 329.

Whether or not the invoker initializes this field is irrelative. DB2 did not move the output parameter's data to the work area at the time of the call.

16.3.4 Solutions for this abend

There are a few options available for consideration:

On the surface, the stored procedure needs to deal with the initialization of PCALL-CTR.

But, if the stored procedure is:

- Counting the calls for the invoking program, then the invoker should take control of the initialization and define the parameter as input and output, so that the values are copied in both directions.

  Note: We think it is usually best if the invoker counts for itself. If the stored procedure is not going to further process the count, why force every user to supply a counter even if the caller is not concerned with the number of calls issued.

- Using this counter to keep track of its own utilization, it could initialize the counter when it is defined, and just increment within itself and not be a parameter at all.

  If the stored procedure is going to process this count, then you must consider the reusability of this module to develop a method that will also maintain the integrity of the counter.

16.4 Compiler and LE options for debugging

We describe the COBOL Compiler and LE options available for debugging.

16.4.1 COBOL compiler options

When using COBOL, set the SYM suboption of the TEST compiler option. The SYM suboption of TEST causes the compiler to add debugging information into the object program to resolve user names in the routine, and to generate a symbolic dump of the DATA DIVISION. With this suboption specified, statement numbers will also be used in the dump output along with offset values.

To simplify debugging, use the NOOPTIMIZE compiler option. Program optimization can change the location of parameters and instructions in the dump output.

You can use the following COBOL compiler options to prepare your program for runtime debugging:

```
LIST, MAP, OFFSET, TEST, VBREF, XREF
```

Refer to *Enterprise COBOL for z/OS Programming Guide Version 3 Release 4*, SC27-1412 for details.

16.4.2 Language Environment runtime options

There are several runtime options that affect debugging in Language Environment (LE). The TEST runtime option, for example, can be used with a debugging tool to specify the level of control the debugging tool has when the routine being initialized is started. The ABPERC, CHECK, DEPTHCONDLMT, ERRCOUNT, HEAPCHK, INTERRUPT, TERMTHDACT, TRACE, TRAP, and USRHDLR options affect condition handling. The ABTERMENC option affects how an application ends (that is, with an abend or with a return code and reason code) when an unhandled condition of severity 2 or greater occurs. The HEAPCHK option, in particular, can be handy to find the source of storage overlays. The option checks all modifications to storage to see if they are outside the application's heap.

The following Language Environment runtime options affect debugging:

ABPERC, ABTERMENC, CHECK, NODEBUG, DEPTHCONDLMT, ERRCOUNT, HEAPCHK, INFOMSGFILTER, INTERRUPT, MSGFILE, MSGQ, PROFILE, RPTOPTS, RPTSTG, STORAGE, TERMTHDACT, TEST, TRACE, TRAP, USRHDLR, XUFLOW

Refer to *z/OS Language Environment Programming Reference,* SA22-7562 for details.

16.5 IBM Debug Tool

Debug Tool combines the richness of the z/OS environments with the power of Language Environment to provide a debugger for programmers to isolate and fix their program bugs and test their applications. Debug Tool gives you the capability of testing programs in batch, using a non programmable terminal in full-screen mode, or using a workstation interface to remotely debug your programs.

16.5.1 IBM Debug Tool overview

The IBM Debug Tool is the successor to the IBM Distributed Debugger. It is sold in conjunction with the compilers or stand alone. IBM also sells the Debug Tools Utilities and

Advanced Functions separately that includes the full Debug Tool capabilities but adds a lot of extra functions and utilities.

The Debug Tool helps you test programs and examine, monitor, and control the execution of programs written in Assembler, C, C++, COBOL, or PL/I on a z/OS system. Your applications can include other languages; Debug Tool provides a disassembly view that lets you debug at the machine code level those portions of your application. However, in the disassembly view, your debugging capabilities are limited. Table 16-6 and Table 16-7 map out the combinations of compilers and subsystems that are supported.

For an updated list of supported compilers and environments, see the Debug Tool Web site at:

http://www.ibm.com/software/awdtools/debugtool

Table 16-6 Debug Tool interface type by Compiler or Assembler

Compiler or Assembler	Batch mode	Full screen mode	Remote mode
OS/VS COBOL V1.2.4 (with limitations[a])	Y	Y	
AD/Cycle® COBOL/370™ V1R1	Y	Y	
VS COBOL II V1.3.1, V1.3.2 and V1.4 (with limitations[a])	Y	Y	Y
COBOL for MVS & VM V1(with limitations[a])	Y	Y	Y
COBOL for OS/390 and VM V2	Y	Y	Y
Enterprise COBOL for z/OS V3, V4	Y	Y	Y
OS PL/I V2.1, V2.2, and V2.3 (with limitations[a])	Y	Y	
PL/I for MVS and VM V1.1.1	Y	Y	
VisualAge® PL/I for OS/390 V2.2	Y	Y	
Enterprise PL/I for z/OS V3	Y	Y	Y
AD/Cycle C/370™ V1.2	Y	Y	
C/C++ for MVS/ESA V3	Y	Y	
C/C++ feature of OS/390	Y	Y	
C/C++ feature of z/OS	Y	Y	Y
IBM High Level Assembler (with limitations[a]) V1.4 and V1.5	Y	Y	Y

a. Supported with Debug Tool Utilities and Advanced Functions

Batch mode

You can use Debug Tool command files to predefine a series of Debug Tool commands to be performed on a running batch application. Neither terminal input nor user interaction is available for batch debugging of a batch application. The results of the debugging session are saved to a log, which you can review at a later time.

Full-screen mode

Debug Tool provides an interactive full-screen interface on a 3270 device, with debugging information displayed in four windows:

Monitor window Displays the variables and their values, controlled by entering the SET AUTOMONITOR ON and MONITOR commands.

Source window Displays the source or listing file. The Debug Tool can find this for you, or you can specify where to find it.

Log window Records your interactions with Debug Tool and the results of those interactions.

Memory window Displays a section of memory, controlled by entering the MEMORY command.

The default screen displays three physical windows, with one assigned the Monitor window, the second assigned the Source window, and the third assigned the Log window. You can swap the Memory window with the Log window.

You can debug all languages supported by Debug Tool in full-screen mode. This refers to the debugging mode that requires a second terminal, a VTAM terminal, to be started and used to debug an application.

After the VTAM terminal has been started, you can optionally use the Debug Tool Terminal Interface Manager, or TIM for short, to identify that terminal to Debug Tool by using a user ID instead of a LU name. If you are use the Debug Tool Terminal Interface Manager, specify the VTAM suboption with your user ID, as in the following example:

```
Test=(,,VTAM%USERABCD)
```

Our example of Debug Tool session at 16.5.2, “IBM Debug Tool on z/OS: An example” on page 336, uses the Terminal Interface Manager.

Table 16-7 Debug Tool interface type by subsystem

Subsystem	Batch mode	Full screen mode	Remote mode
TSO	Y	Y	Y
JES batch	Y	Y	Y
UNIX System Services		Y	Y
CICS	Y	Y	Y
DB2	Y	Y	Y
DB2 stored procedures		Y	Y
IMS (TM and DB) with BTS TSO foreground		Y	
IMS (TM and DB) with BTS batch	Y	Y	Y
IMS without BTS IMS DB batch	Y	Y	Y
IMS without BTS IMS		Y	Y

Remote debug mode

In remote debug mode, the host application starts Debug Tool, which uses a TCP/IP connection to communicate with a remote debugger on your Windows workstation. Debug

Tool, in conjunction with the remote debuggers provides users with the ability to debug host programs, including batch, through a graphical user interface (GUI) on the workstation.

For example, a COBOL batch job running in MVS/JES, or a COBOL CICS batch transaction, can be interactively debugged through a TCP/IP connection to a workstation that has the Rational® Developer for System Z installed in it.

The following remote debuggers are available:

- Compiled Language Debugger component of Rational Developer for System z
- Compiled Language Debugger component of WebSphere Developer for zSeries
- Compiled Language Debugger component of WebSphere Developer for System z
- WebSphere Developer Debugger for zSeries
- WebSphere Developer Debugger for System z

The remote operating system that Rational Developer and WebSphere Developer supports are Windows 2003, Windows Vista®, Windows XP. Check the following Web sites for more information regarding the above products.

```
http://www.ibm.com/software/awdtools/rdz/support/index.html
http://www.ibm.com/software/awdtools/devzseries/support/
```

16.5.2 IBM Debug Tool on z/OS: An example

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where stored procedures run. Stored procedures run in a WLM-established address space and must execute with the LE runtime loadlib. Therefore, the default debugging tool on z/OS is the IBM Debug Tool. We used Debug Tool V8.1 in our example. The supported releases fore the Debug Tool are 6.1 - 8.1, with 8.1 being the recommended version. Version 5.1 is going out of service at the end of March 2008.

The DB2 administrator must define the address space where the stored procedure runs. This can be a DB2 address space or a Workload Manager (WLM) address space. This address space is assigned a name that is used to define the stored procedure to DB2. In the JCL for the DB2 or WLM address space, verify that the following data sets are defined in the LINK LIST or STEPLIB concatenation and that they have the appropriate RACF read authorization for programs to access them:

```
LOADLIB for the stored procedure
SEQAMOD for Debug Tool
SCEERUN for Language Environment
```

After updating the JCL, the DB2 administrator must recycle the DB2 or WLM address space so that these updates take effect.

As previously mentioned, Debug Tool gives you the capability of testing programs in batch, using a non-programmable terminal in *full-screen mode*, or using a workstation interface to remotely debug your programs.

Before demonstrating some of the debugging commands and options available with Debug Tool, we discuss the Terminal Interface Manager, or TIM, and its setup. Next, we describe the stored procedure setup, and finally, an example using the program previously debugged in the classical manner.

Refer to *Debug Tool for z/OS Debug Tool Utilities and Advanced Functions for z/OS User's Guide Version 8.1,* SC19-1196 for details of using the Debug Tool in this and other modes.

16.5.3 The Terminal Interface Manager

A feature of the Debug Tool is the Terminal Interface Manager (TIM). TIM enables a user to use full-screen mode through a VTAM terminal without having to know the LU name of the VTAM terminal. The Terminal Interface Manager provides a method to correlate a user ID to the terminal. This is useful in situations where it is cumbersome to edit the TEST runtime parameter with the LU name of the VTAM terminal.

The setup for using TIM is very similar to using VTAM MFI. There are some differences, however, namely:

- The port number you use in the Telnet session's Link parameters will be different. Your system programmer will need to do some setup to assign this port number to a TIM session.
- You will need to change the RUN OPTIONS in your stored procedure to the following:

  ```
  TEST(,,,VTAM%userid:*)
  ```

  Note that rather than an LU name, a userid is specified.

Setup for the TIM

We first need to set up a Telnet session to communicate to DB2 for z/OS from our workstation. In this example we used IBM Personal Communications (PCOMM). You can use your emulator of choice. The first session will run the application that calls your stored procedure.

1. Create a second Telnet session using your emulator of choice. (Treat the session that you use for editing and submitting JCL as your first session.) Figure 16-2 displays the customize communication panel of PCOMM to be modified to define the host.
   - On the panel where you define your host connection, in Host Name or IP Address, put in your server host name. Ours was `TLBA07ME.TOROLAB.IBM.COM`. In the Port Number field, enter the port number that your system programmer assigned for Telnet. In our environment, this was `2026`, which was specified in our server's TCPIP.TCPPARMS (TCPPROF) file. Figure 16-2 shows the Link parameters for the PCOMM session we are creating.

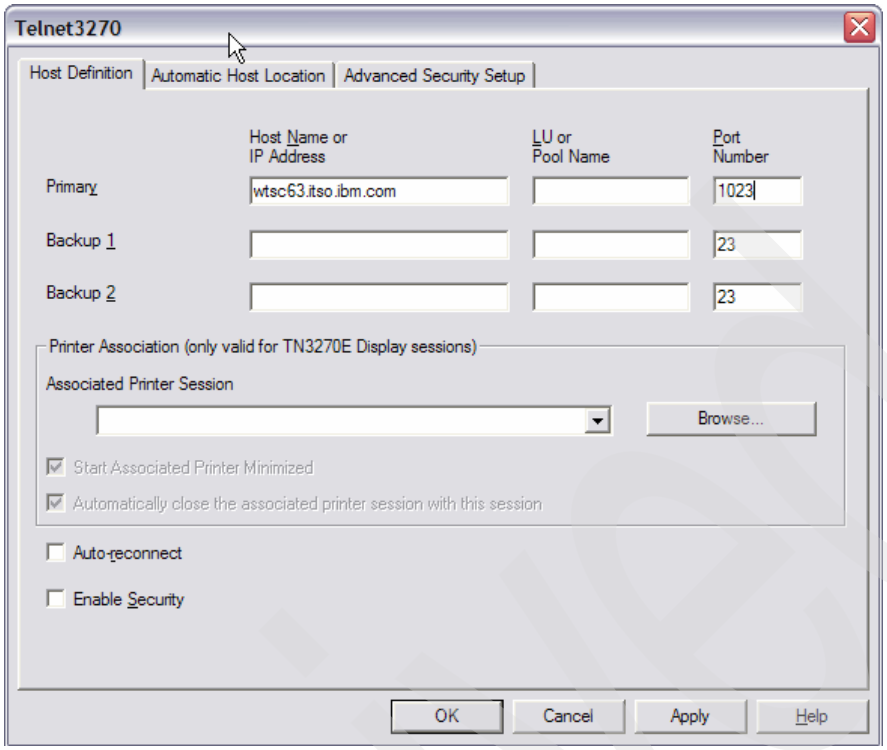

Figure 16-2 Defining the z/OS connection

2. Save this session as a Workstation Profile, and give it a meaningful name such as `DT_TIM.ws` as shown in Figure 16-3.

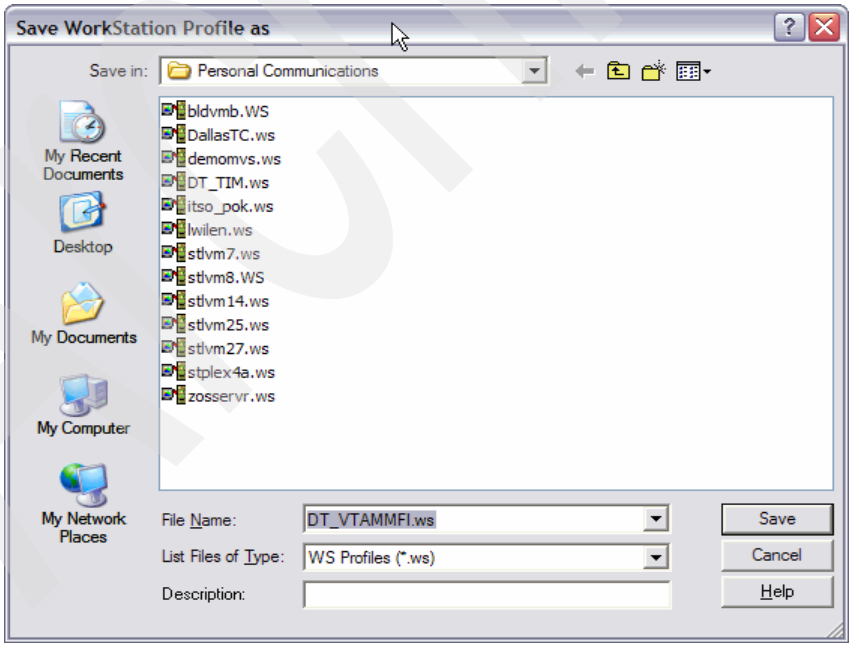

Figure 16-3 Saving session profile

3. Start the session. TIM's initial page is shown in Figure 16-4.

Figure 16-4 Terminal Interface Manager, login page

After entering your TSO userid and password, you will get a message:

```
EQAY001I Terminal S070095 connected for user <your TSO userid>
EQAY001I Ready for Debug Tool
```

You will need to call your stored procedure, either through an application or tooling. Once the stored procedure is called, this session hosts the Debug Tool full screen session, and you will be able to enter the debug operations we discuss in 16.5.5, “Demonstration of Debug Tool with TIM” on page 340.

16.5.4 Stored procedure setup

1. Using your first session, alter the runopts setting for the stored procedure you want to debug with this LU name value as follows:

   ALTER PROCEDURE procedure name RUN OPTIONS 'TEST(,,,VTAM%userid:*)'

2. Using your first session, recompile the stored procedure with the TEST(ALL) compile option, and add a SYSPRINT DD statement to the compile step to create a data set with the compile output to be available for the debugger to display in the VTAM window:

   ```
   //COB.SYSPRINT DD DISP=SHR,DSN=hiqualifier.your created library(procname)
   ```

3. Then, start your stored procedure (submit the JCL from the first session), and the Debug Tool will be launched in the above second session window.

Important: The above describes the necessary steps for the first-time debugging of a stored procedure using the MFI VTAM setup. For subsequent debugging sessions, the steps are reduced to the following:

1. Start your second emulator session and note the LU name.
2. Alter the RUN OPTIONS to reflect your new LU name.
3. Start debugging by:
   - Open your first screen and submit the run JCL.
   - *Do not* attempt to open or modify the second session in any way until the debugger begins.

16.5.5 Demonstration of Debug Tool with TIM

Figure 16-5 shows the initial screen of our sample debugging program EMPDTLC, which is a stored procedure. There are twelve default PF key assignments displayed at the bottom of the screen. There are three windows whose position or layout can be altered:

- Monitor window: Continuously displays the value of monitored variables and other items, depending on the command used.
- Source window: Displays your program source code. (We use the Debug Tool SIZE 12 line command to create a larger source window.)
- Log window: Records your commands and Debug Tools responses

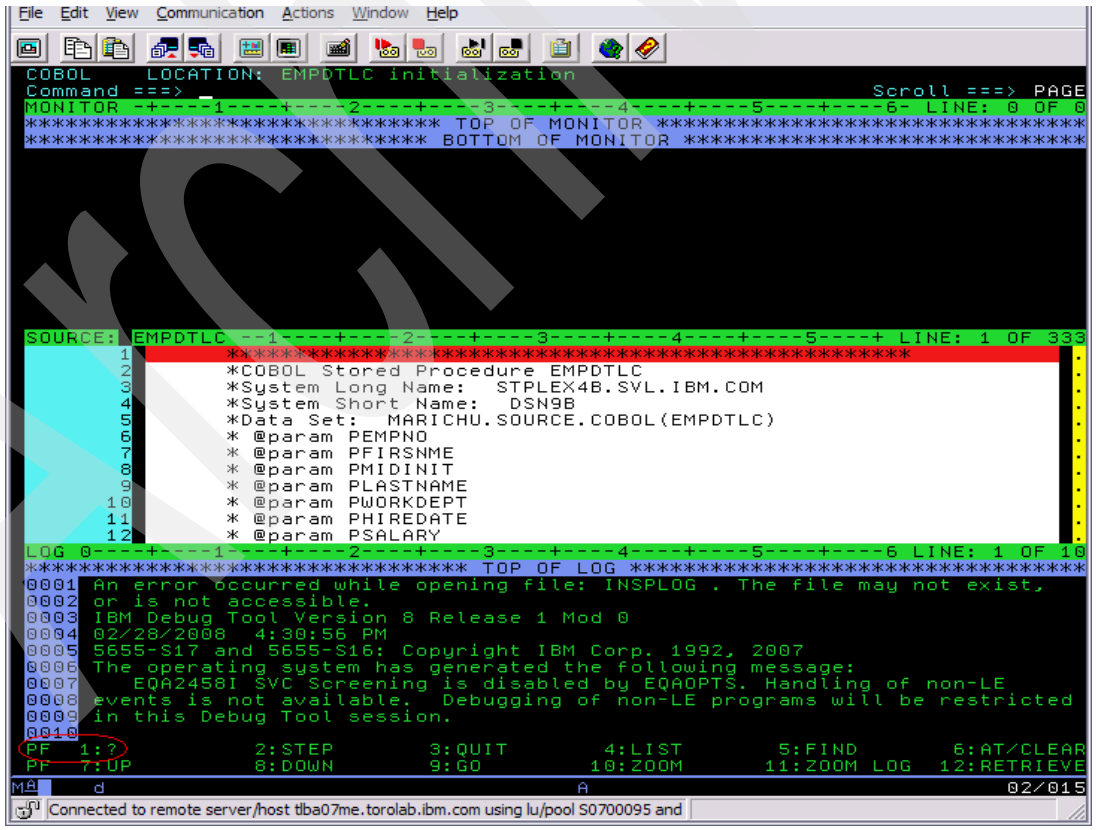

Figure 16-5 Initialization of Debug Tool stored procedure, EMPDTLC

The PF Keys

The PF1 key, ?, gives a list of the Debug Tool commands. Commonly used commands such as **`Step, Go, List`** and **`Find`** are assigned to PF2, PF4 and PF5 respectively.

Layout command

The Debug Tool **`layout`** command allows you to customize the layout of the debug session. Enter the **`layout`** on the command line. Select your layout preference as shown in Figure 16-6.

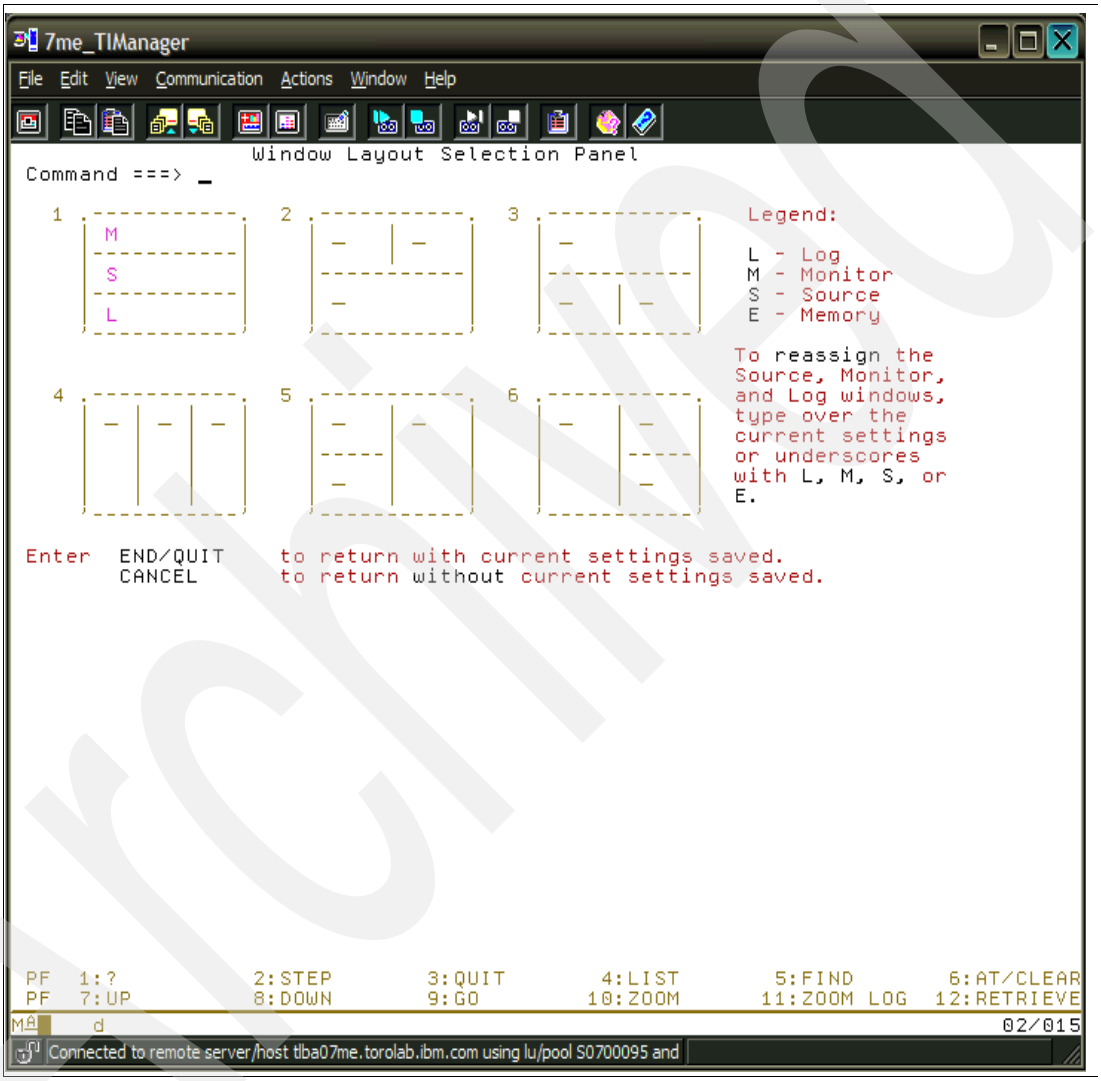

Figure 16-6 Debug Tool TIM, layout selections

Checking the Version number

To verify the version of the Debug Tool that we are using, on the command line, type:

```
CALL %VER
```

The information appears in the Log window as shown in Figure 16-7.

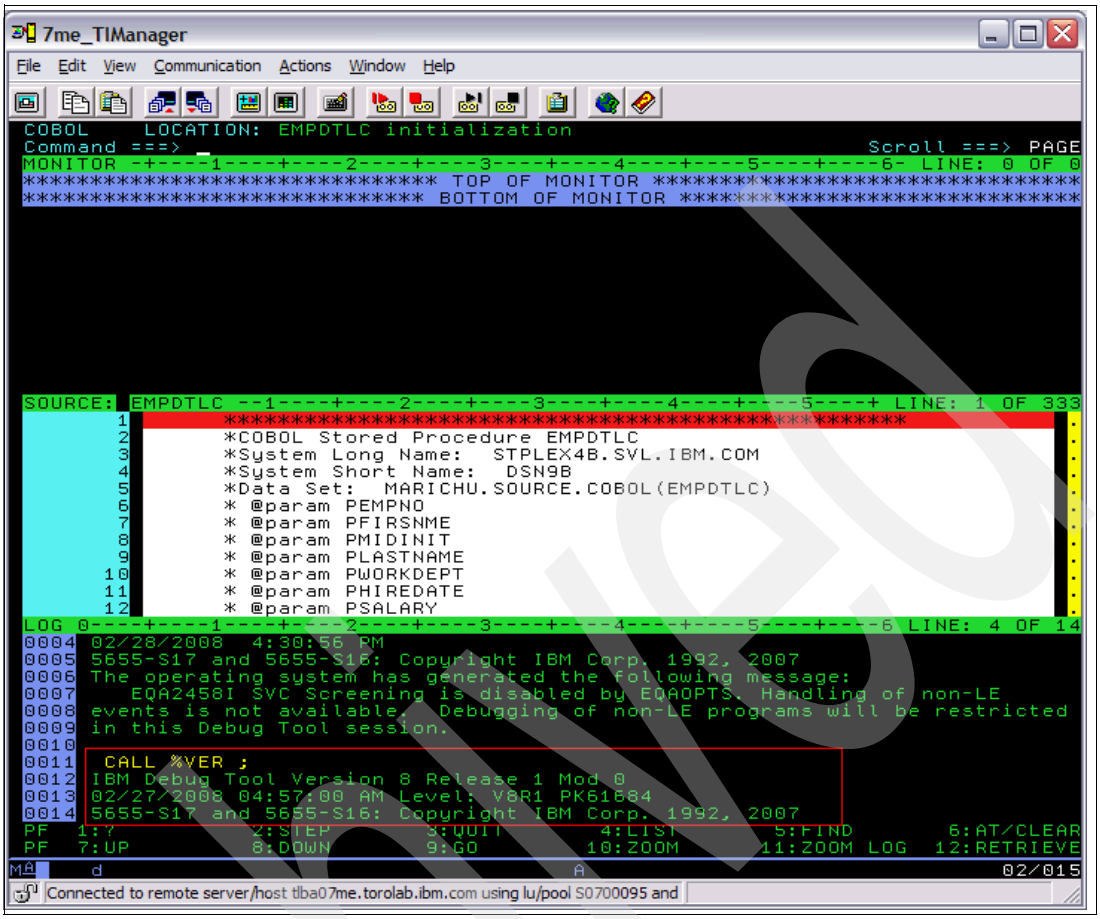

Figure 16-7 Debug Tool TIM, check version

Navigating through the source code

We navigate through the source code using the FIND command. The FIND does not appear in the Log window. To repeat a find, press PF5.

In our example, let's find the variable `PCALL-CTR`. Type `FIND PCALL-CTR` on the command line. Debug Tool will position the source window on the first executable statement that uses this variable. The variable that is found is identified by a black background, as shown in Figure 16-8.

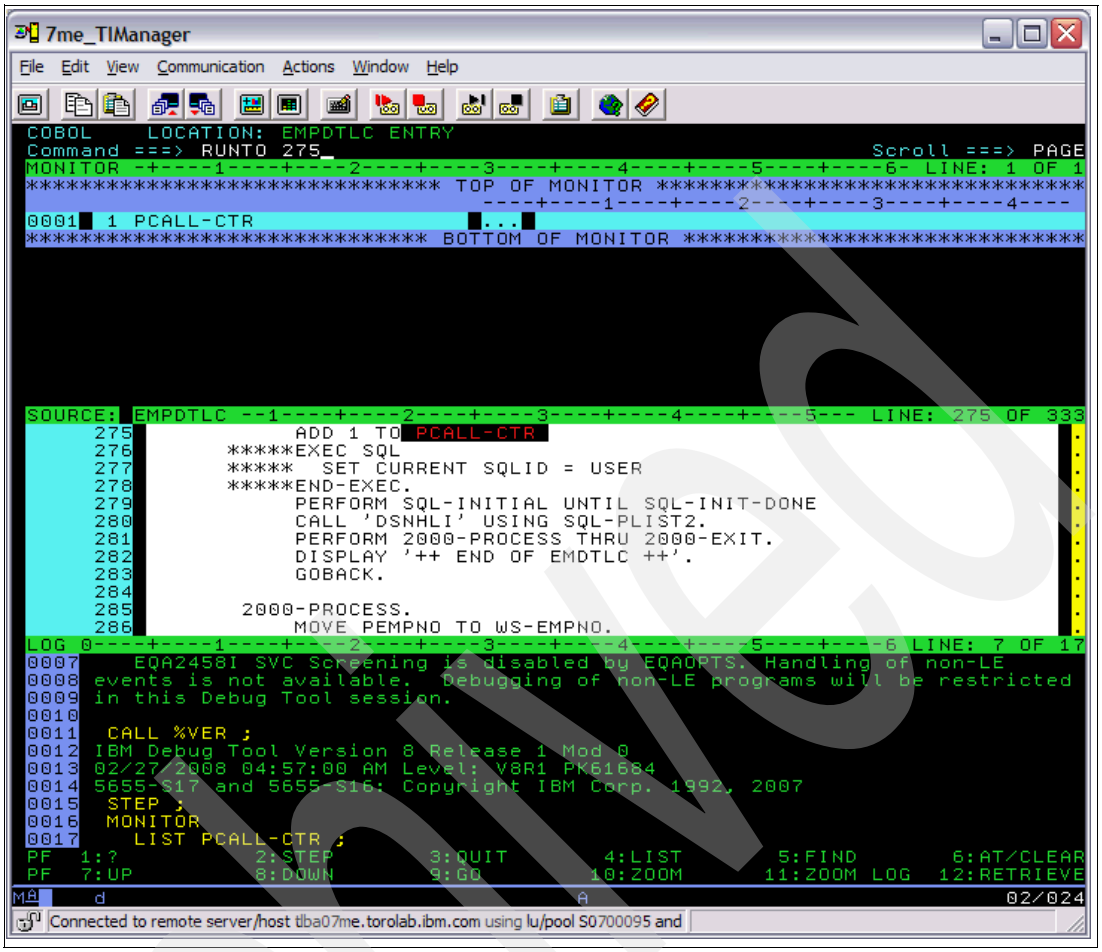

Figure 16-8 Debug Tool TIM, Find PCALL-CTR

Displaying and monitoring variables

In the Debug Tool, you use the LIST command to display the current contents of a variable. You can also “watch” or monitor this variable via the MONITOR LIST command. So, to monitor the variable `PCALL-CTR`, issue the command:

```
MONITOR LIST PCALL-CTR
```

In the Monitor window, the variable `PCALL-CTR` is displayed, and the contents of this variable are displayed as shown in Figure 16-9.

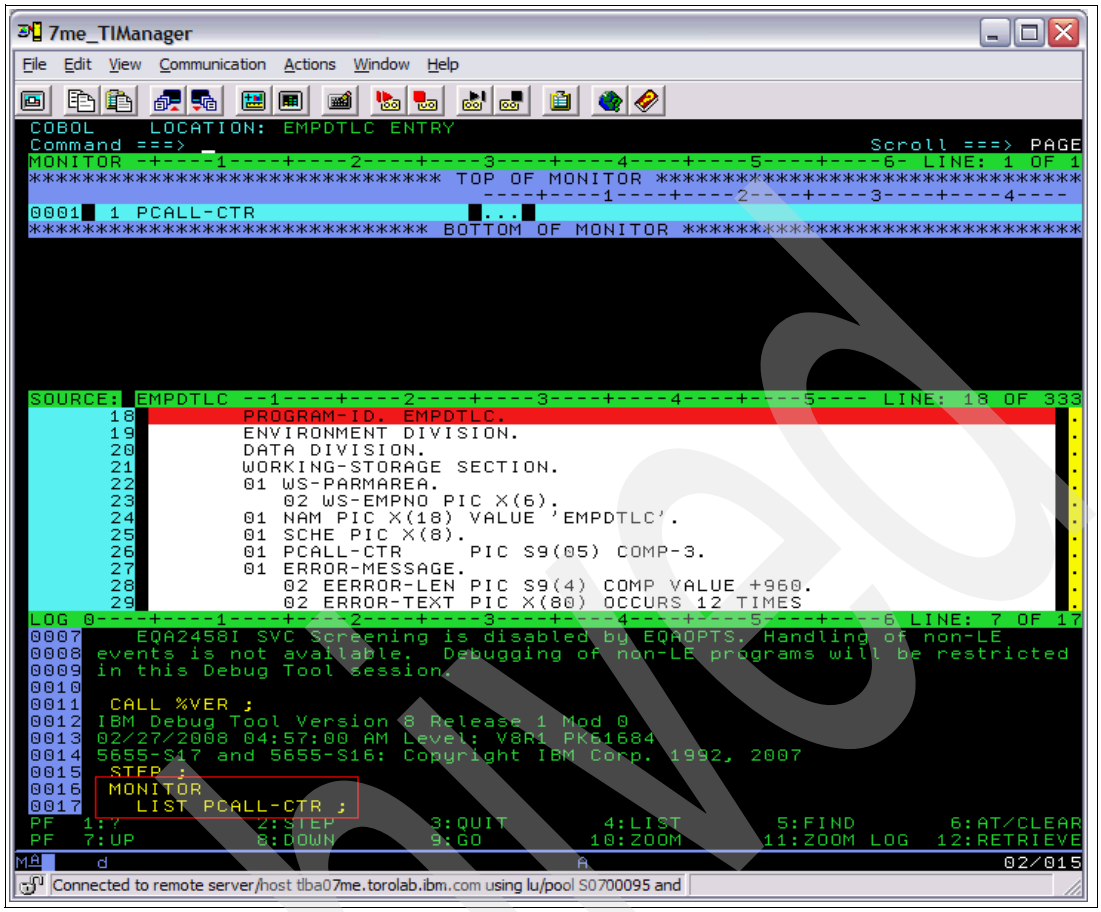

Figure 16-9 Debug Tool TIM, Monitor List command

We note that the value of PCALL-CTR is unprintable, so we want to see what's in memory.

Debug Tool can display the value of a variable in memory and displays the hex values in the Memory window. To display what's in storage starting at the register where PCALL-CTR is located, issue the following commands:

```
LIST STORAGE (PCALL-CTR);
MEMORY PCALL-CTR;
ZOOM MEMORY;
```

Debug Tool will access the address assigned to PCALL-CTR, which in this case is 32A470F8, and display what is in memory, starting at that address, as shown in Figure 16-10.

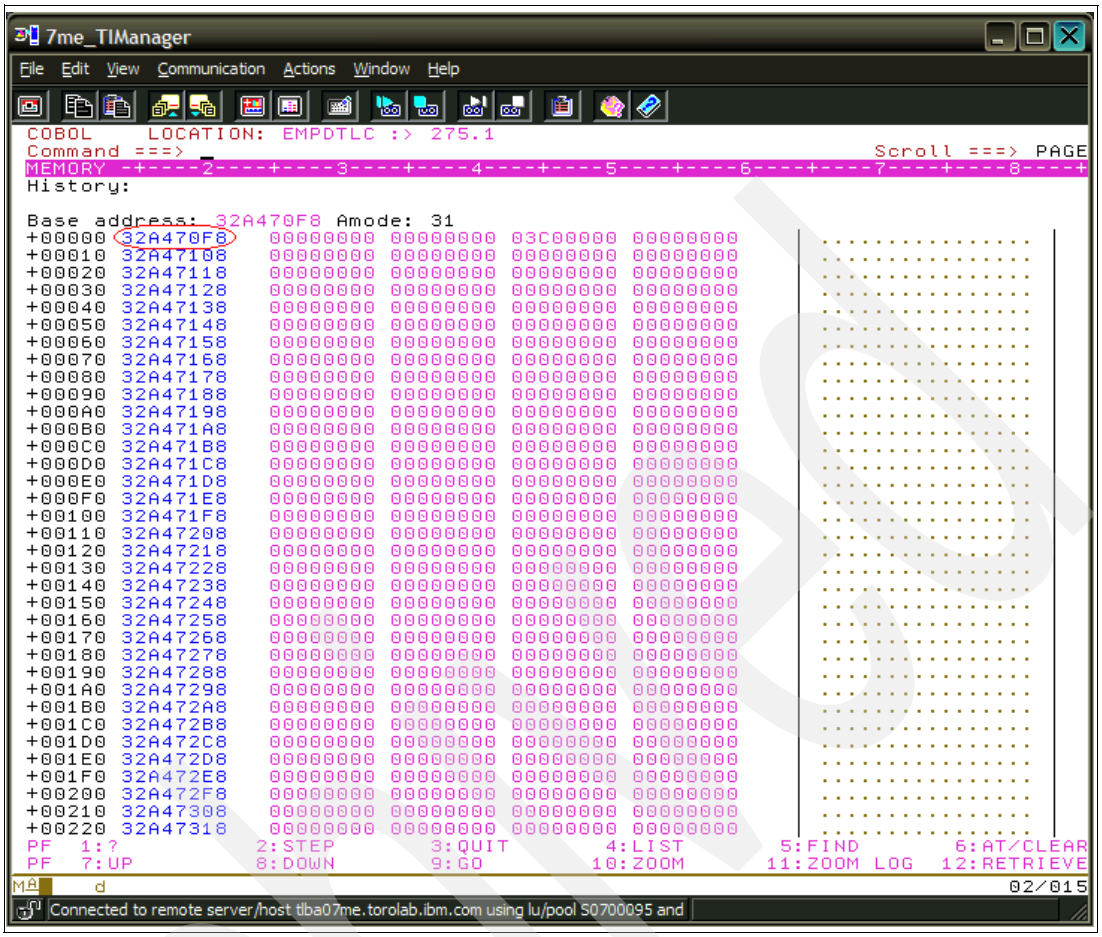

Figure 16-10 Debug Tool TIM, Memory display starting at address 32A470F8

Changing variable values

The Debug Tool allows you to change the value of a variable prior to the variable being used in a statement. So, in our example, the Debug Tool is about to execute the statement

```
ADD 1 TO PCALL-CTR
```

However, we can see that this variable is not initialized. So, let us initialize the value of `PCALL-CTR`. You can do this by simply typing over the current value in the Monitor view. In our example, we typed `0000` in the value field. This action generated the statement `MOVE 0000 TO PCALL-CTR`, as shown in Figure 16-11.

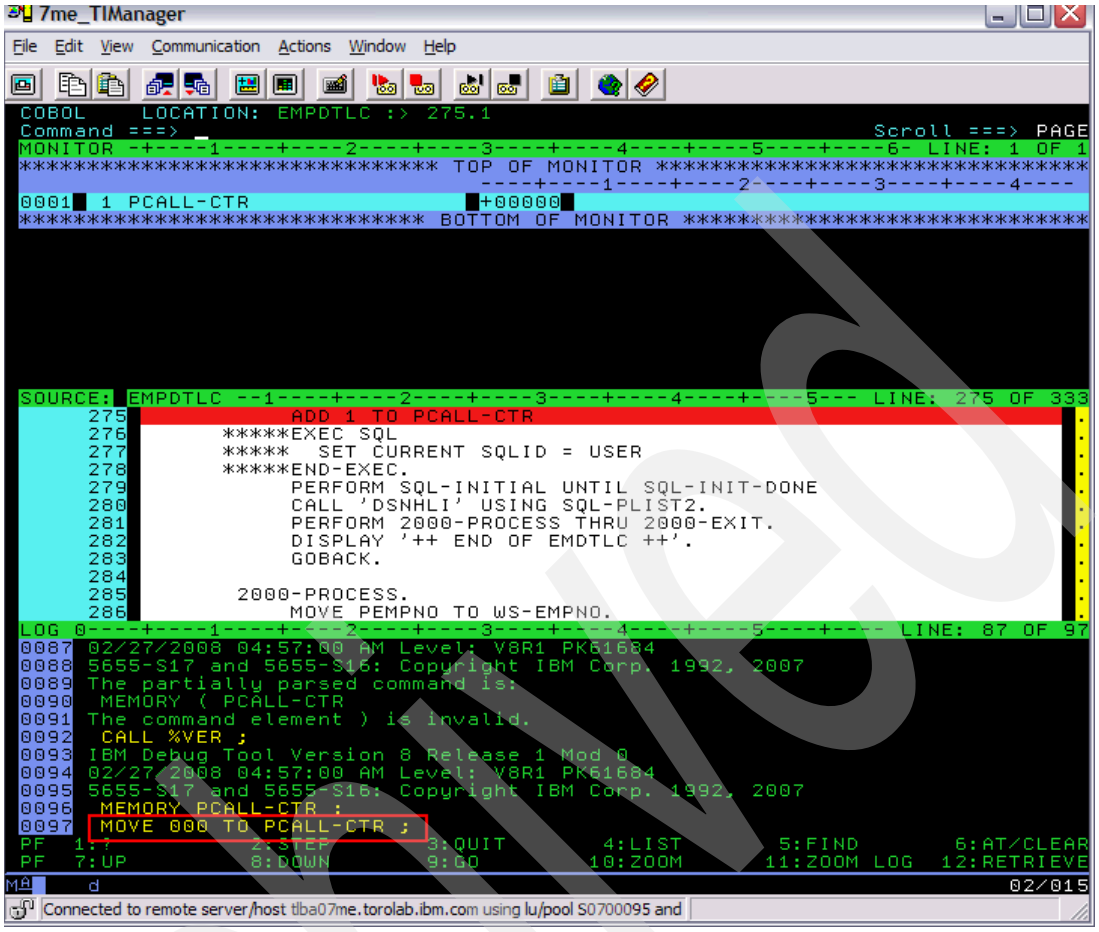

Figure 16-11 Debug Tool TIM, initialize PCALL-CTR, generated MOVE statement

Creating breakpoints

You can create breakpoints by placing the cursor at a given line and pressing PF6. If you know the line number where you want execution to stop, you can issue the AT <line number> command.

In Figure 16-12, the current statement to be executed is denoted by the red line on line 301. We create a breakpoint in line 306 by placing the cursor there, and pressing PF6 or by typing the command `AT 306`.

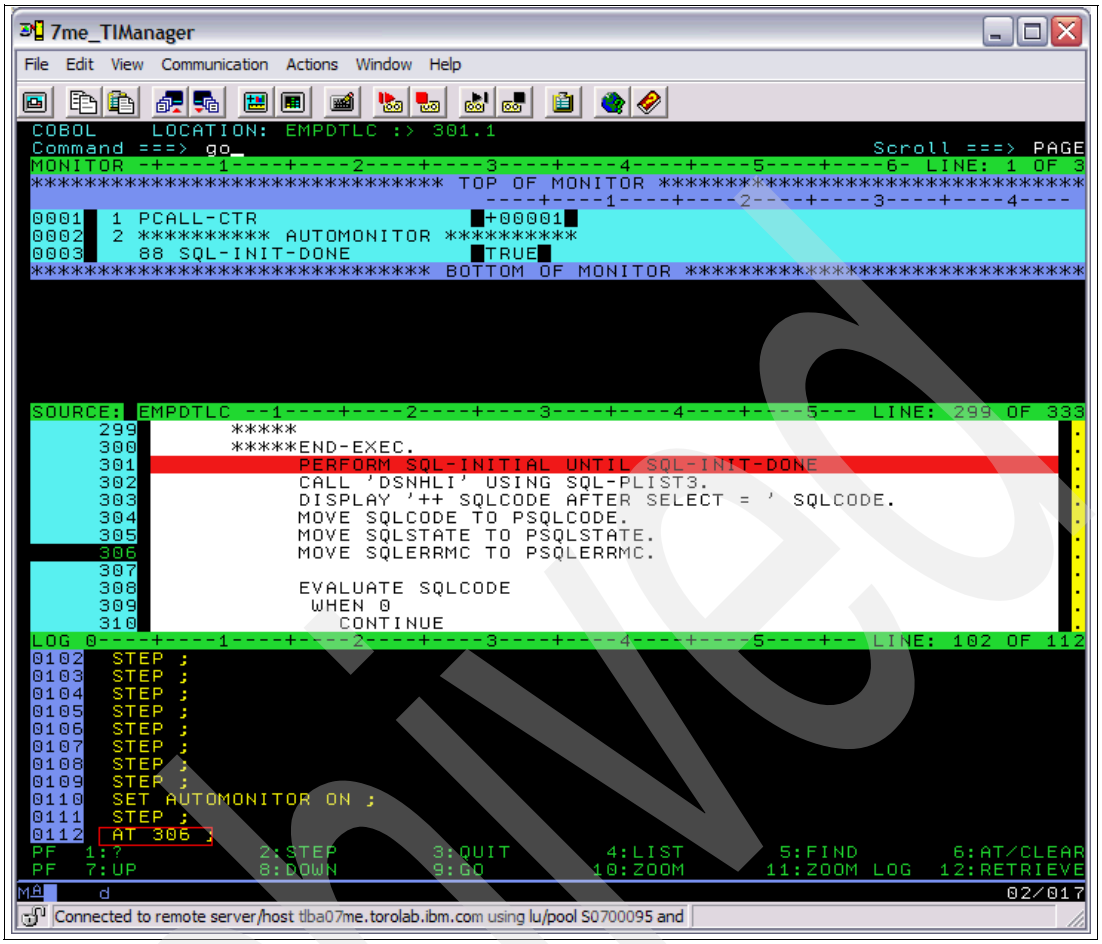

Figure 16-12 Debug Tool TIM, AT command

Resuming or terminating execution

The following Debug Tool commands control execution of a statement.

- GO or RUN starts or resumes execution of the stored procedure until the next breakpoint or until the end of he stored procedure.
- RUNTO <line number> runs the stored procedure up to the specified line number without setting a breakpoint.
- STEP INTO executes the current statement. If the current statement is a called function or routine, this command will position the cursor on the first executable statement of the called function
- STEP OVER executes the current statement. If the current statement is a called function or routine, this command will execute the call and position the cursor on the next statement after the call
- QQUIT or QUIT ABEND ends a Debug Tool session.

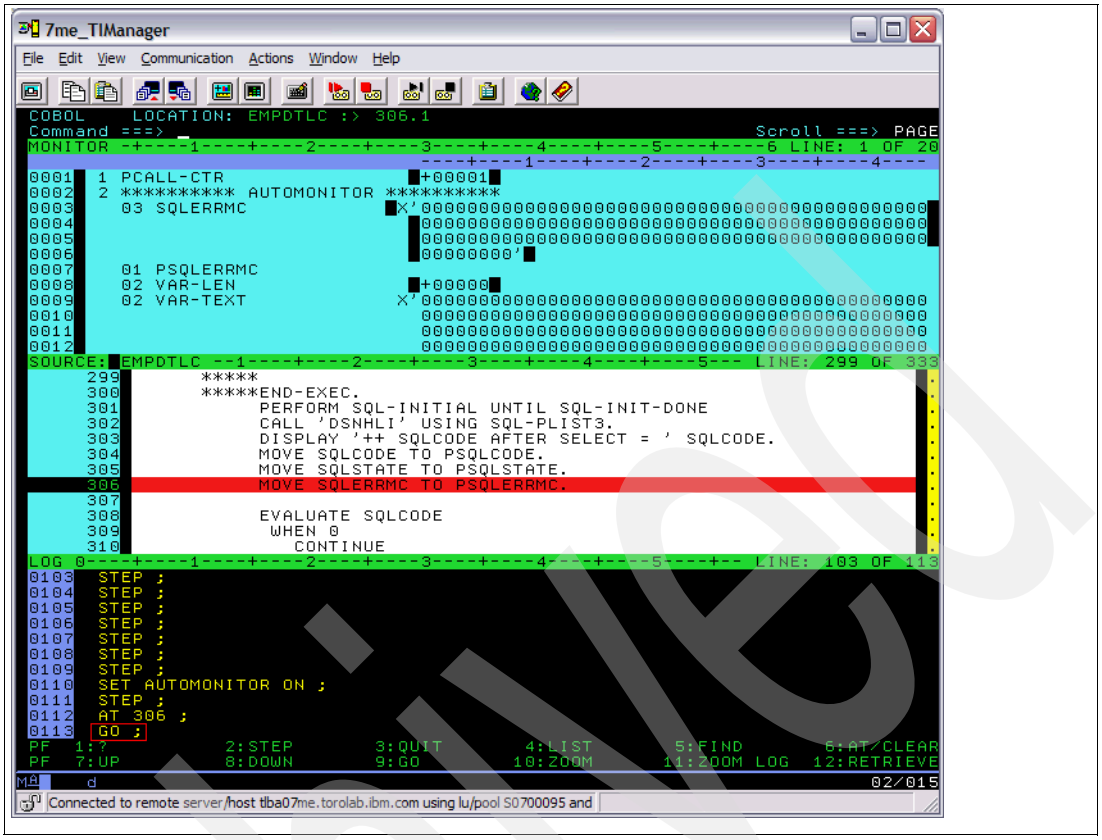

Figure 16-13 Debug Tool TIM, GO command

Obviously, you have not seen everything that is available for debugging with the Debug Tool. We barely scratched the surface. For more information about the Debug Tool commands, refer to *Debug Tool for z/OS Debug Tool Utilities and Advanced Functions for z/OS(R) User's Guide Version 8.1,* SC19-1196.

16.5.6 User-defined exit for specifying runtime options

Debug Tool Utilities and Advanced Functions, Version 8 Release 1 (5655-S16) provides a customized version of the Language Environment user exit routine (CEEBXITA) to link into the application load module. The routine returns a TEST runtime option when called by the Language Environment initialization logic.

The routine extracts the TEST runtime option from a data set with a name that is constructed dynamically from a naming pattern, which includes the user ID token &USERID. This token is replaced with the user ID of the current user during name construction. Each user can specify an individual TEST runtime option when debugging an application.

If the PROGRAM TYPE for a DB2 stored procedure is MAIN, you can specify the TEST runtime options through the DB2 catalog or Language Environment EQADDCXT exit routine. If you specify the TEST runtime option through the Language Environment EQADDCXT exit routine, you can run the stored procedure with your own set of suboptions. Another user can run or debug the stored procedure with his own set of suboptions. Therefore, multiple users can run or debug the stored procedure at the same time.

To set up this exit routine, do the following:

- Create a data set with the following requirements:

- Sequential data set (DSORG=PS)
- Record format and length requirements:
  - RECFM(F) or RECFM(FB) and LRECL >=80
  - RECFM(V) or RECFM(VB) and LRECL >=84

- Launch the Debug Tool Utilities and select **Manage TEST Run-time Data Set** as shown in Figure 16-14.

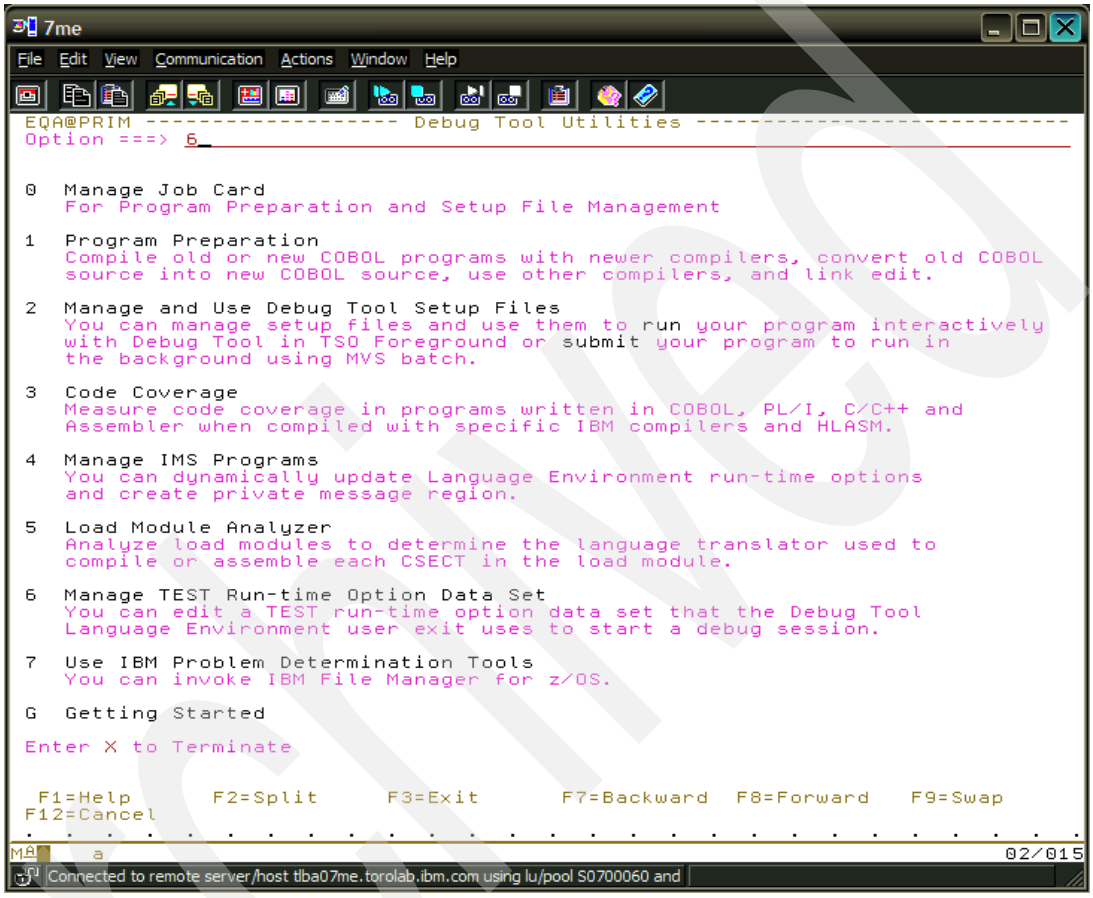

Figure 16-14 Manage TEST Run0time Option Data Set

- In the next page, specify the data set that you created in the first step as shown in Figure 16-15.

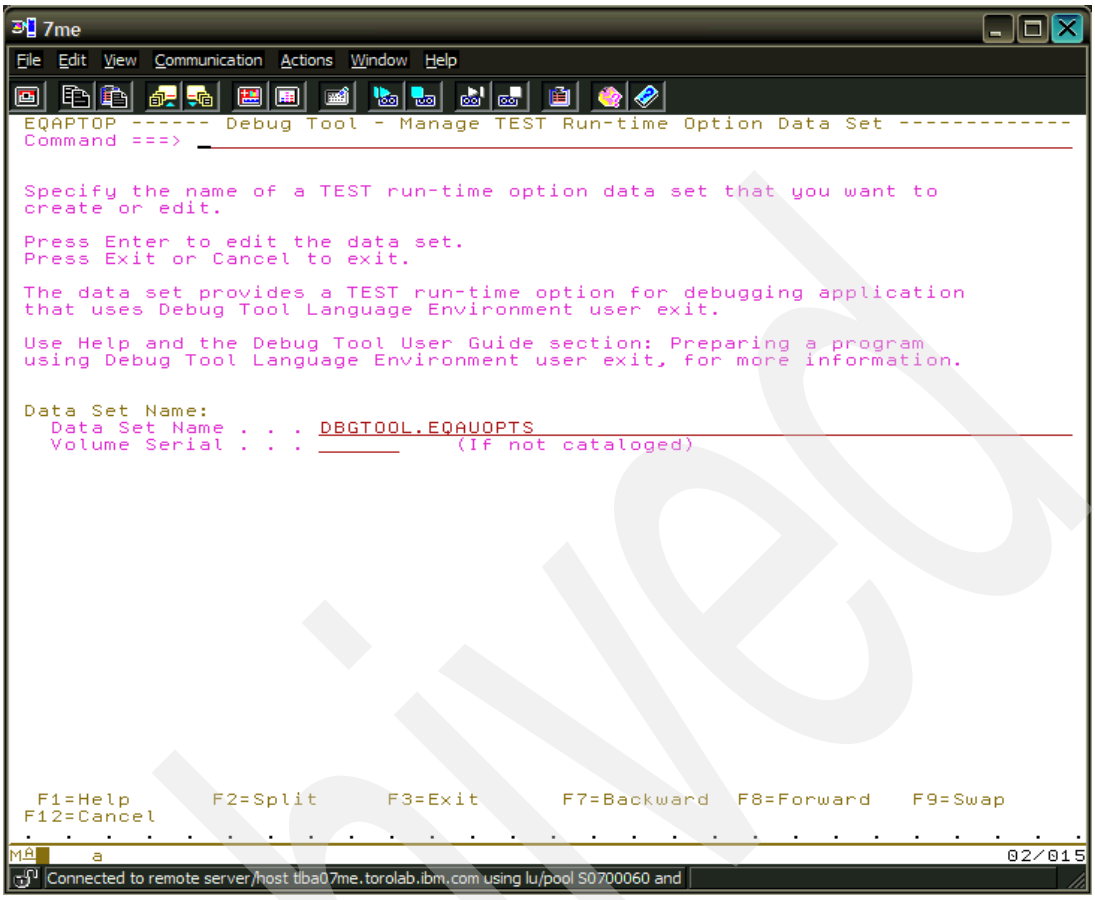

Figure 16-15 Specify the data set name

- Specify the TEST runtime option you want to use through the DB2 catalog or Language Environment EQADDCXT exit routine. See Figure 16-16.

Note: If the PROGRAM TYPE for the stored procedure is SUB, you must specify the TEST runtime option through the DB2 catalog. You are limited to specifying one specific set of suboptions, which means that every user that runs or debugs that stored procedure uses the same set of suboptions. If both methods are used, the Language Environment exit routine takes precedence over the DB2 catalog.

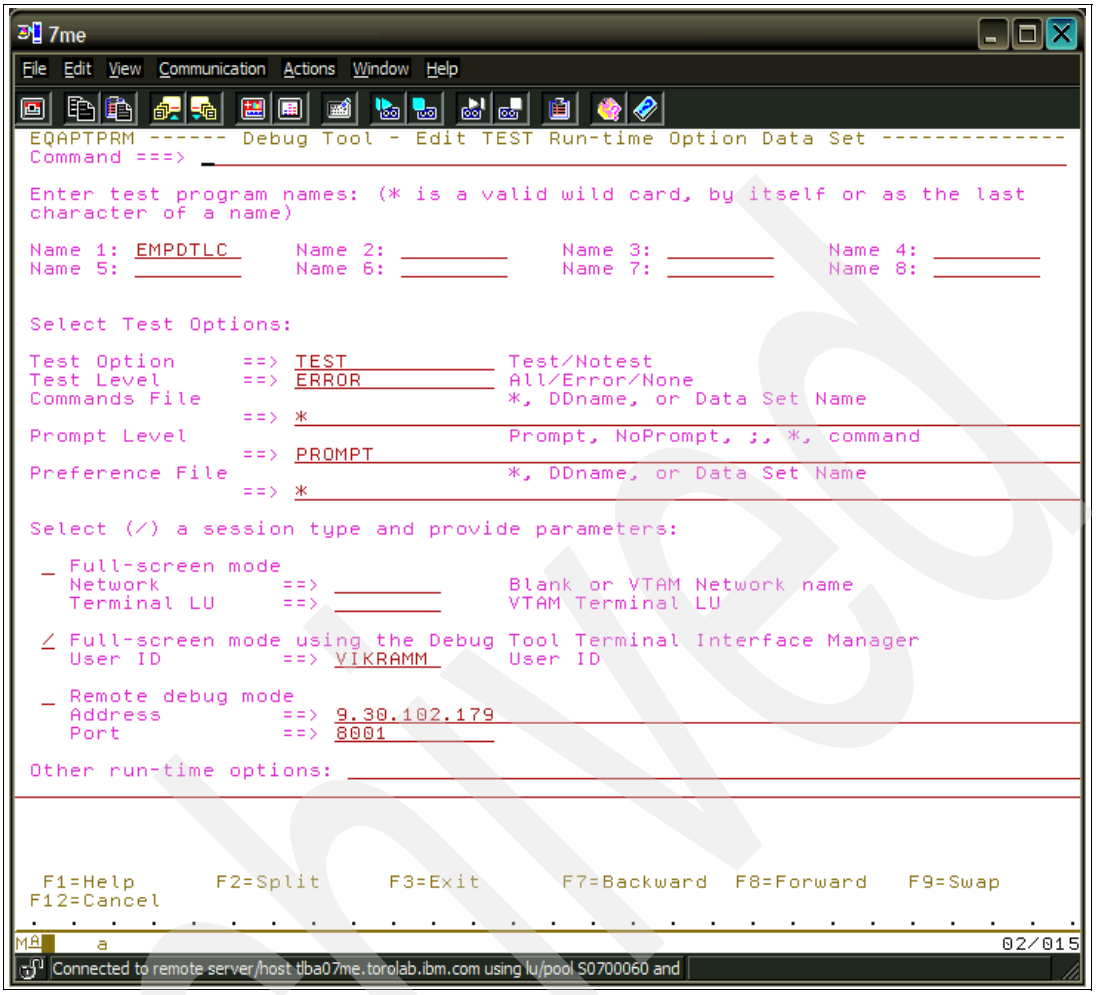

Figure 16-16 Managing your Run-time Option Data Set

16.5.7 Debug Tool references

For more information to help you get started with using VTAM MFI, Terminal Interface Manager, and the user-defined Exit routines, see the library available from the Debug Tool Web site:

http://www.ibm.com/software/awdtools/debugtool/

We found the following references to the standard manuals useful:

- *Debug Tool for z/OS User's Guide,* SC19-1196:
  - Chapter 12, "Preparing a DB2 stored procedures program"
  - Chapter 15, "Preparing a program by using the Language Environment exit routine"
  - Chapter 16, "Starting Debug Tool from the Debug Tool Utilities"
  - Chapter 21, "Starting a full-screen debug session"
  - Chapter 39, "Debugging DB2 stored procedures"
- *Debug Tool for z/OS Customization Guide*, SC19-1200
  - Chapter 5, "Enabling debugging in full-screen mode through a VTAM terminal"

- *Debug Tool for z/OS Reference and Messages*, GC19-1198
  - Chapter 1, “Debug Tool Run-time Options, TEST runtime option” page 23+

16.6 GET DIAGNOSTICS

The `GET DIAGNOSTICS statement`, available with DB2 for z/OS Version 8, enables applications to retrieve diagnostics information about statements that have been executed. This statement complements and extends the diagnostics that are available in the SQLCA. See Figure 16-17.

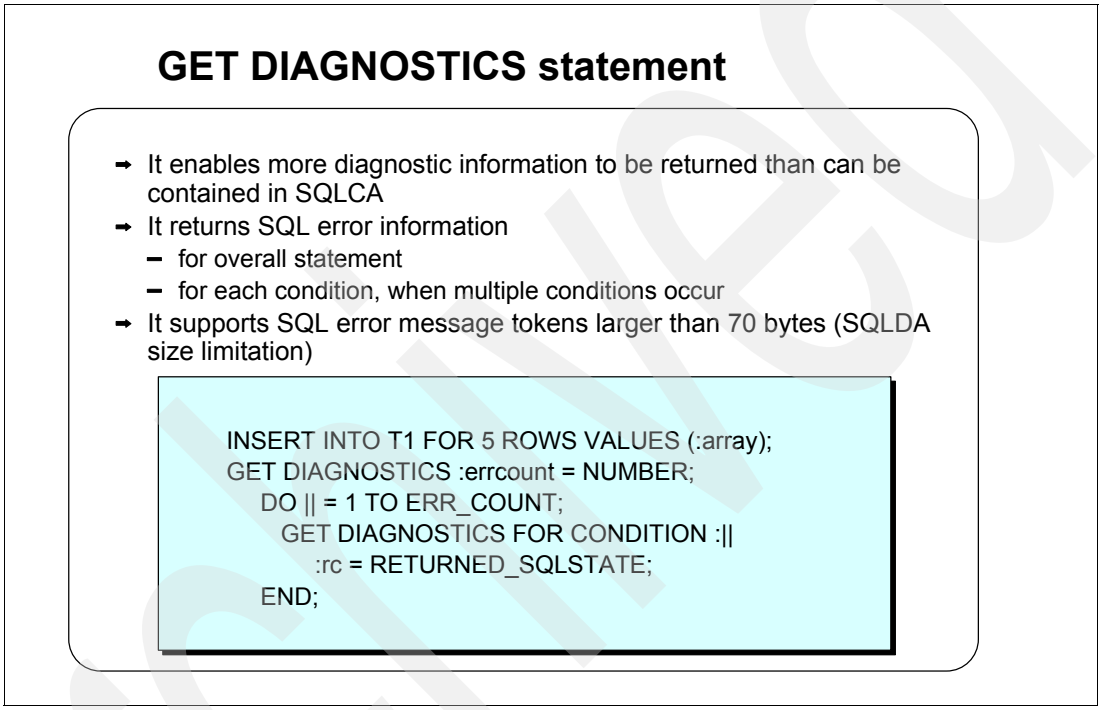

Figure 16-17 GET DIAGNOSTICS statement

In Figure 16-18 we show the syntax for the GET DIAGNOSTICS statement.

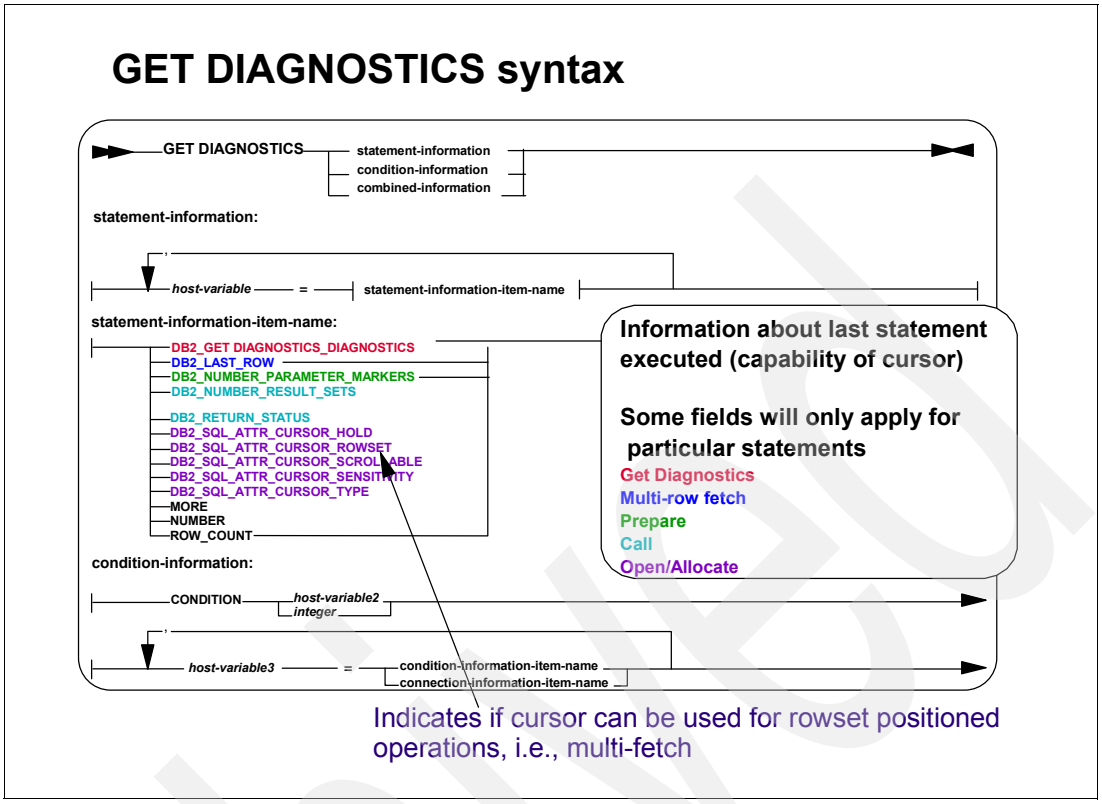

Figure 16-18 GET DIAGNOSTICS syntax

The following C language program example demonstrates the use of this new statement.

In an application, use GET DIAGNOSTICS to determine how many rows were updated:

```
long rcount;
EXEC SQL UPDATE T1 SET C1 =C1 +1;
EXEC SQL GET DIAGNOSTICS :rcount = ROW_COUNT;
```

After execution of this code segment, *rcount* contains the number of rows that were updated.

Diagnostic information for multi-row fetch

The SQLCA is used to return information on errors and warnings found while fetching from a rowset cursor. After each FETCH statement from a rowset cursor, information is returned to the program through the SQLCA as follows:

- SQLCODE contains the SQLCODE.
- SQLSTATE contains the SQLSTATE.
- SQLERRD3 contains the actual number of rows returned. If SQLERRD3 is less than the number of rows requested, then an error or end-of-data condition occurred.
- SQLWARN flags are set to represent all the warnings that were accumulated while processing the FETCH statement.

Additional information may be obtained about the fetch, including information on all exception conditions encountered while processing the fetch statement from the GET DIAGNOSTICS statement.

Consider an example where we attempt to fetch 10 rows with a single FETCH statement.

Assume that an error, SQLCODE -180 is detected on the 5th row. SQLERRD3 is set to 4 for the four returned rows; SQLSTATE is set to 22007, SQLCODE is set to -180.

This information is also available from the GET DIAGNOSTICS statement, for example:

```
GET DIAGNOSTICS :num_row = ROW_COUNT, :num_cond = NUMBER;
```

would result in num_row = 4 and num_cond = 1 (1 condition).

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

would result in sqlstate = 22007, sqlcode = -180, and row_num = 5.

There are some cases where DB2 returns a warning if indicator variables are provided, or an error if indicator variables are not provided. These errors can be thought of as data mapping errors that result in a warning if indicator variables are provided. The GET DIAGNOSTICS statement may be used to retrieve information about all the data mapping errors that have occurred.

Diagnostic information for multi-row insert

When NOT ATOMIC is specified, the inserts are processed independently. This means that if one or more errors occur during the execution of an INSERT of a row, then processing continues. The row that was being inserted at the time of the error is not inserted. Execution continues with the next row to be inserted, and any other changes made during the execution of the multiple row INSERT statement are not backed out. When ATOMIC is in effect, if an insert value violates any constraints, or if any other error occurs during the execution of an INSERT of a row, then all changes made during the execution of the multiple row INSERT statement are backed out.

The SQLCA reflects the last warning encountered. The SQLCA is used to return information on errors and warnings found during a multiple-row-insert. If indicator arrays are provided, the indicator variable values are used to determine if the value from the host variable array, or NULL, is used. The SQLSTATE contains the warning from the last data mapping error.

Additionally, when NOT ATOMIC is in effect, then status information is available for each failure or warning that occurred while processing the insert. The status information for each row is available through the GET DIAGNOSTICS statement.

As an example, assume that you are inserting multiple rows using host variable arrays for column values. The table T1 has 2 columns, C1 is a SMALL INTEGER column, and C2 is an INTEGER column. INSERT 10 rows of data into the table T1. The values to be inserted are provided in host variable arrays :hva1 (an array of INTEGER) and :hva2 an array of DECIMAL(15,0) values. The data values for :hva1 and :hva2 are represented in Table 16-8.

Table 16-8 Data values for :hva1 and :hva2

Array entry	:hva1	:hva2
1	1	32768
2	-12	90000
3	79	2
4	32768	19
5	8	36
6	5	24

Array entry	:hva1	:hva2
7	400	36
8	73	4000000000
9	-200	2000000000
10	35	88

The INSERT statement is as follows:

```
EXEC SQL
INSERT INTO T1 (C1, C2) FOR 10 ROWS VALUES (:hva1:hvind1, :hva2:hvind2)
NOT ATOMIC;
```

After execution of the INSERT statement, we have the following in the SQLCA:

```
SQLCODE = 0
SQLSTATE = 0
SQLERRD3 = 8
```

Although we attempted to insert 10 rows, only eight rows of data were inserted. Further information can be found by using the GET DIAGNOSTICS statement, for example:

```
GET DIAGNOSTICS :num_row = ROW_COUNT, :num_cond = NUMBER;
```

would result in num_row = 8 and num_cond = 2 (2 conditions)

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

would result in sqlstate = 22003, sqlcode = -302, and row_num = 4

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

would result in sqlstate = 22003, sqlcode = -302, and row_num = 8

Diagnostic information for native SQL procedures

The GET DIAGNOSTICS statement has been enhanced in DB2 9 for z/OS to include more diagnostic information for native SQL language stored procedures. For details on the changes to GET DIAGNOSTICS for native SQL language stored procedures, see 15.7.2, “GET DIAGNOSTICS” on page 310.

17

Remote stored procedure calls

In this chapter we discuss the characteristics of stored procedures that are executed in remote database management systems. The remote DBMS can be DB2 for z/OS or an instance of many other products. In this chapter, our discussion is restricted to DB2 for z/OS only.

This chapter contains the following:

- Remote stored procedures
- Remote stored procedure preparation

17.1 Remote stored procedures

An application that calls a stored procedure basically requests services from a database management system (DBMS). Therefore, the application can be called a client or a requester. As shown in Figure 17-1, if the client and the stored procedure both execute within the same DB2 subsystem (DB2L), then the stored procedure is called a *local stored procedure*.

If the client executes in one DB2 subsystem (DB2L) and the stored procedure resides and executes in another, remote DB2 subsystem (DB2R), then the stored procedure is termed a *remote stored procedure*.

Stored procedures, being reusable programmed components that are executed on the data server side, can be called both from local applications as well as remote applications.

The subsystem where the client program executes is usually referred to as the *local subsystem* (DB2L in our example). In other words, the local subsystem is the one where the application plan or package is bound. If the stored procedure is executed on a subsystem other than the local subsystem, this subsystem is then called *remote subsystem* (DB2R in our example). The remote DB2 subsystem can physically reside on the same machine, for example in a different subsystem in the same operating system, or it can even reside thousands of miles away on a different machine.

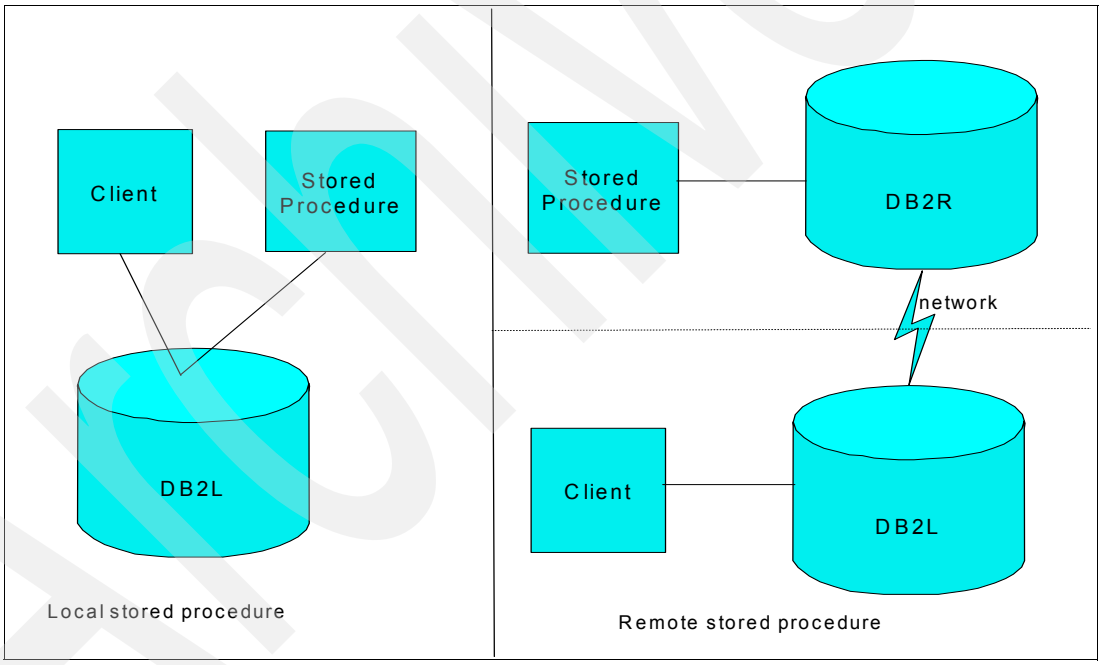

Figure 17-1 Local stored procedure vs. remote stored procedure

No matter where the remote DB2 subsystem resides, it is known to your local subsystem by its location name. The location name of the remote subsystem (DB2R) is recorded in the local subsystem's communication database (CDB).

Before calling a remote stored procedure, a client has to establish a connection to the remote subsystem. There are two ways to do this:

- Use an explicit *SQL* CONNECT statement:

Example 17-1 Explicit CONNECT statement

```
EXEC SQL connect to DB2R;
```

```
EXEC SQL CALL SYSPROC.ADMIN_INFO_SSID(?,?,?)
```

- Execute the CALL with a *qualified three-part name.* Here the connection is implicitly established.

Example 17-2 Implicit CONNECT due to qualified three-part name

```
EXEC SQL CALL DB2R.SYSPROC.ADMIN_INFO_SSID(?,?,?)
```

Either way, the information for the DRDA connectivity, and especially the location name, is found in the communication database.

Communication database

The communication database (CDB) is essentially a set of DB2 catalog tables, that let you control and configure how requests come in and leave your local DB2 subsystem. Furthermore, it configures how the conversations with a remote database management system (DBMS) are established. On DB2 for z/OS the Distributed Data Facility (DDF) uses the CDB to send and receive distributed data requests. Data at a remote DB2 subsystem can be accessed with two access protocols, DRDA or the DB2 private protocol. However, invocation of a stored procedure is supported only with DRDA, therefore the DB2 private protocol is no longer considered in this chapter. Refer to the chapter “Connecting distributed database systems” in *DB2 Version 9.1 for z/OS Installation Guide,* GC18-9846-02 for more differences between DRDA and the DB2 private protocol.

The CDB consists of the following tables in the DB2 catalog:

- SYSIBM.IPLIST
- SYSIBM.IPNAMES
- SYSIBM.LOCATIONS
- SYSIBM.LULIST
- SYSIBM.LUMODES
- SYSIBM.LUNAMES
- SYSIBM.MODESELECT
- SYSIBM.USERNAMES

The following tables have been modified with the advent of Version 9:

SYSIBM.IPLIST:

Column IPADDR revised (enabled for IPv6 addresses)

SYSIBM.IPNAMES:

Column IPADDR revised (enabled for IPv6 addresses)

Columns USERNAMES and SECURITY_OUT revised

SYSIBM.LOCATIONS:

New columns TRUSTED, SECURE added

SYSIBM.USERNAMES:

Column TYPE revised (new type ’S’ added)

If a DB2 subsystem is intended to act only as a data server, there is no need to populate the CDB tables, the defaults are sufficient. To communicate with remote DB2 subsystems in order to request data, the CDB of the requesting DB2 subsystem has to be populated. Information like the remote subsystem’s IP address, the port number, or the location name has to inserted

into some of the CBD tables. When a TCP/IP connection to the remote subsystem should be established, it is usually sufficient to only populate the following three tables:

- SYSIBM.LOCATIONS
- SYSIBM.IPNAMES
- SYSIBM.USERNAMES

Practical information with sample case studies can be found in the following two IBM Redbooks publications:

- *Moving Data Across the DB2 Family,* SG24-6905 *(especially: Section 5.1 "Coding for distributed data")*
- *Distributed Functions of DB2 for z/OS and OS/390,* SG24-6952 *(especially: "Appendix B - DB2 connectivity")*

For more information on how to establish DRDA connectivity, refer to the following manuals:

- *DB2 Version 9.1 for z/OS Installation Guide,* GC18-9846-02
- *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854-02
- *DB2 Version 9.1 for z/OS Administration Guide,* SC18-9840-02
- *Reference for Remote DRDA Requesters and Servers,* SC18-9853

CONNECT statement

The CONNECT statement, as illustrated in Example 17-1, connects the application process to a designated database server. There are two types of CONNECT statements. Both have the same syntax but feature different semantics.

CONNECT type 1 - Lets the application connect to only a single database at any time during a unit of work. This type of connection models DRDA remote unit of work processing (*DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854-02, *Chapter 1: "Distributed Data - Remote unit of work"*).

CONNECT type 2 - Lets the application connect concurrently to any number of database servers within a single unit of work. The unit of work can contain multiple SQL statements that access different database servers. However, a single SQL statement can only access one distinct database server. This type of connection models DRDA distributed unit of work processing (see *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854-02, *Chapter 1: "Distributed Data - Distributed unit of work").*

An application process can either be in a *connected* or *unconnected* state. Especially for CONNECT type 1, it can also be *connectable* or *unconnectable*, since only one connection is allowed within a unit of work. The initial state of an application process is connected, for both type 1 and type 2 connections, whereas the server of that connection is usually the local DB2 subsystem. Table 17-1 shows some more differences between type 1 and type 2 CONNECT statements. For further information and the complete state transition diagrams, refer to *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854-02 and *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

Table 17-1 Differences between type 1 and type 2 CONNECT

CONNECT (type 1 - Remote unit of work)	CONNECT (type 2 - Distributed unit of work)
Only one CONNECT statement can be executed within the same unit of work.	More than one CONNECT statement can be executed within the same unit of work. Only one connection is active (current) at a time.

CONNECT (type 1 - Remote unit of work)	CONNECT (type 2 - Distributed unit of work)
- A CONNECT statement is executed successfully only when the application process is in the *connectable* state, which implies that there is no active unit of work. - A unit of work starts only when a SQL statement accesses data, after the CONNECT was executed. - Once a unit of work is started the application process is moved to a *unconnectable* state. - A COMMIT or ROLLBACK puts the process back in a connectable state.	There are no rules about the *connectable* state. An application process is either connected or unconnected.
- If a CONNECT statement fails because the application process is not in the *connectable* state, the connection state of the application process is unchanged. - If a CONNECT statement fails although it is in a *connectable* state, the application process is placed in the *connectable and unconnected* state until the CONNECT succeeds.	If a CONNECT statement fails, the current SQL connection is unchanged and any subsequent SQL statements are executed by the currently connected server, unless the failure prevents the execution of SQL statements by that server.
A successful CONNECT ends any existing connections of the application process. Accordingly, CONNECT also closes any open cursors of the application process. (The only cursors that can possibly be open when CONNECT is successfully executed are those defined with the WITH HOLD option.)	CONNECT does not end connections and does not close cursors. It does not issue any implicit COMMIT statements.
A CONNECT to the current server is executed like any other CONNECT (Type 1) statement. The SQLRULES bind option has no effect on it.	If the SQLRULES(STD) bind option is in effect, a CONNECT to an existing SQL connection of the application process is an error. Thus, a CONNECT to the current server is an error. For example, an error occurs if the first explicit CONNECT is a CONNECT TO x where x is the local DB2. If the SQLRULES(DB2) bind option is in effect, a CONNECT to an existing SQL connection is not an error. Thus, if x is an existing SQL connection of the application process, CONNECT TO x makes x its current connection. If x is already the current connection, CONNECT TO x has no effect on the state of any connections.

Precompiler determines the type of connection

The type of connection that is prepared in a package to be run using DRDA access, is determined by a precompiler option:

CONNECT(2) - Refers to type 2 connections or “distributed unit of work“. Use this type of connection type whenever multiple data servers are involved in a single unit of work. This is the default value.

CONNECT(1) - Causes the CONNECT statements to work according to the limited “remote unit of work” definitions.

The connect rules that apply to an application process are determined by the first CONNECT statement that is executed (successfully or unsuccessfully). Programs containing multiple CONNECT statements that are precompiled with different CONNECT precompiler options cannot execute as part of the same application process.

The primary authorization of the process or the authorization ID specified on the CONNECT statement must be authorized to connect to the identified remote server or local DB2 subsystem.

17.2 Remote stored procedure preparation

There is no difference in the stored procedure preparation, whether it is to be used for local applications or for remote applications. The following steps happen at the server side for external or fenced stored procedures:

1. Develop a program to be used as a stored procedure, compile, link-edit it.
2. Define the stored procedure.
3. Bind the DBRM into a package. The stored procedure package does not have to be bound to a plan. It uses a caller's plan. This is the same for all stored procedures, local or remote.
4. Grant an execute privilege to the invoker of the stored procedure.

DB2 Version 9 for z/OS now supports native SQL procedures that are not compiled and linked to an executable program, but rather store their procedural statements natively in the DB2 catalog and directory. To create a native SQL procedure, only the following steps are required:

1. Define the stored procedure with the procedural statements embedded in the procedure body.
2. Grant an execute privilege to the invoker of the stored procedure.

Note: Native SQL procedures support a new DEPLOY keyword on the BIND statement that allows for easy deploying on a remote data server. For more information, refer to Chapter 15, "Native SQL procedures" on page 253.

17.2.1 Client program preparation

The client program has to do a few special preparation steps in order to invoke the remote stored procedures:

- Develop the calling program, which makes an unqualified call to a remote stored procedure.
- Compile and link-edit the program.
- Bind the program:
  - Bind the DBRM into a package at the local DB2, use BIND option DBPROTOCOL(DRDA).
  - Bind the DBRM into a package at the remote location (DB2R).
  - Bind all packages into a plan at the local DB2. Specify DBPROTOCOL as DRDA.

Note: If your client program has a need to access multiple servers, then you have to bind the program at each sever.

17.2.2 Sample scenarios of program preparations

Invoke a local stored procedure from a DB2 application

Let us consider a few examples where the client program calls local as well as remote stored procedures. Let us begin with Example 17-3 where we first deal with a local stored procedure call, and then move towards more complex remote calls.

Example 17-3 Client program invoking local stored procedure

```
Client program looks like:
unqualified CALL
   EXEC SQL
     CALL SP (parameter_list)
   END-EXEC.

BIND for client looks like (only important options are shown).
Local bind:
   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(BIND)

   BIND PLAN(plan_name) -
        PKLIST(col1.drpgm)
```

Example 17-3 shows the steps for preparing a client program to invoke a local stored procedure, SP. In this case the stored procedure and the client both execute against the same DB2 subsystem, for example DB2L. Notice that the PATH option determines the SQL path that DB2 uses to resolve unqualified stored procedure calls. The VALIDATE(BIND) option ensures that all objects and required privileges exist during bind time, otherwise an error is issued.

As soon as the package is bound, you can bind the local package into a plan at the local subsystem. The PKLIST option determines the packages that are bound into the plan. This is especially important when in the later examples remote packages are bound into the local application plan.

Invoke a remote stored procedure from a DB2 application

We now invoke a remote stored procedure. As shown in Example 17-4, the client program issues an explicit CONNECT statement to connect to a remote DB2 server, DB2R. For detailed information on connectivity topics, refer to *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952.

Comparing Example 17-3 with Example 17-4 on page 364 reveals that there is essentially no difference in the CALL statement to the procedure, no matter whether it is local or remote.

Before you can actually invoke the remote stored procedure, the DBRM has to be bound as a package at the local DB2 server and as a remote package at the remote DB2 server, DB2R. The package bind for the local DB2 should have the DBPROTOCOL(DRDA) option specified, due to the explicit CONNECT statement. Observe that the local package is bound with VALIDATE(RUN). This is because we make and unqualified call to a stored procedure the local DB2 does not know about. The stored procedure is only defined at the remote data server.

DB2 issues warning messages for unresolved objects during BIND with VALIDATE(RUN) option, but completes successfully. The procedure name can then be resolved during runtime

since a CONNECT statement is issued before the CALL. We could overcome these warning messages by making a qualified call to the stored procedure with a three-part name, for example *location.schema.name.* However, hard-coding the location qualifier poses some challenges on code portability. Another approach would be to pass the qualifier as a parameter to the program and then issue a dynamic call to the stored procedure. This approach causes an incremental BIND the first time the program is executed, then a cached entry is used.

Example 17-4 Client program invoking a remote stored procedure

```
Client program looks like:

   MOVE 'DB2R' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

unqualified CALL
   EXEC SQL
     CALL SP (parameter_list)
   END-EXEC.

BIND for client looks like (only important options are shown).
Local bind:
   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        DBPROTOCOL(DRDA) -
        VALIDATE(RUN)
Remote bind:
   BIND PACKAGE(db2r.col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(BIND)

   BIND PLAN(plan_name) -
        PKLIST(col1.drpgm, db2r.col1.drpgm) -
        DBPROTOCOL(DRDA).
```

Both packages, local and remote, have to be bound to a local plan. Again the PKLIST option is used for that. At the remote server, the package will be executed under the DISTSERV plan. So, if you run any performance reports like accounting data, you might be interested to know that it will appear under your local plan on local DB2 server and under DISTSERV on a remote DB2 server.

Local SQL processing and remote procedure CALL from a DB2 application

As shown in Example 17-5 on page 365, the client program issues some SQL statements that should be processed at the local DB2 server and additionally calls a remote stored procedure. The preparation looks similar to Example 17-4 except that the remote package bind also has the VALIDATE(RUN) option. This is because the tables referred to in local DB2 are not known to the remote DB2. The VALIDATE(RUN) option is therefore required to bypass the error checking and successfully perform the remote bind. Another change is the QUALIFIER option for a local package. QUALIFIER is used to qualify unqualified DB2 objects, whereas the PATH option is used to qualify stored procedures and user-defined functions.

Example 17-5 Client program with local SQL and invoking remote stored procedure

```
Client program looks like:

SQL issued against the local DB2 subsystem
   EXEC SQL
     SELECT c1, c2, c3, c4 FROM t1....
   END-EXEC.


   MOVE 'DB2R' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

Remote SP call
   EXEC SQL
     CALL SP (parameter_list)
   END-EXEC.

BIND for client looks like (only important options are shown).
Local bind:
   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        QUALIFER(qual) -
        DBPROTOCOL(DRDA) -
        VALIDATE(RUN)
Remote bind:
   BIND PACKAGE(db2r.col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(RUN)

   BIND PLAN(plan_name) -
        PKLIST(col1.drpgm, db2r.col1.drpgm) -
        DBPROTOCOL(DRDA).
```

Invoke stored procedures on multiple locations from a DB2 application

As shown in Example 17-6, the client program has a requirement to invoke stored procedures at multiple locations. In this case, explicit CONNECT statements are issued to different remote servers just before the remote stored procedure is invoked. Also observe that the client application is bound as package at the local server and at all remote servers. The local plan is then bound with a package list that includes the local package and all remote packages. Again, note the VALIDATE(RUN) option, as the stored procedures are only defined on the respective DB2 subsystem.

Example 17-6 Stored procedures at multiple remote servers

```
Client program looks like:

   MOVE 'DB2R' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

   EXEC SQL
     CALL SP (parameter_list)
```

```
   END-EXEC.

   MOVE 'DB2S' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

   EXEC SQL
     CALL SP1 (parameter_list)
   END-EXEC.

   MOVE 'DB2P' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

   EXEC SQL
     CALL SP2 (parameter_list)
   END-EXEC.


BIND for client looks like (only important options are shown).
Local bind:
   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        DBPROTOCOL(DRDA) -
        VALIDATE(RUN)
Remote bind:
   BIND PACKAGE(db2r.col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(RUN)
Remote bind:
   BIND PACKAGE(db2s.col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(RUN)
Remote bind:
   BIND PACKAGE(db2p.col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(RUN)

   BIND PLAN(plan_name) -
        PKLIST(col1.drpgm, db2r.col1.drpgm, -
               db2s.col1.drpgm, db2p.col1.drpgm) -
        DBPROTOCOL(DRDA).
```

At any point in time, the DB2 special register CURRENT SERVER indicates the location name of the server to which the application thread is currently connected.

The DB2 statement CONNECT RESET can be used to reset the connection back to the local DB2 subsystem.

17.2.3 Other considerations on preparing

Precompiler options

The following precompiler options are relevant for preparing a package to be run using DRDA access:

- CONNECT

  Use CONNECT(2), explicitly or by default.

  CONNECT(1) causes your CONNECT statements to allow only the restricted function known as “remote unit of work”. Be particularly careful to avoid CONNECT(1) if your application updates more than one DBMS in a single unit of work.

- SQL

  Use SQL(ALL) explicitly for a package that runs on a server that is not DB2 for z/OS. The precompiler then accepts any statement that obeys DRDA rules.

  Use SQL(DB2), explicitly or by default, if the server is DB2 for z/OS only. The precompiler then rejects any statement that does not obey the rules of DB2 for z/OS.

BIND PACKAGE options

Only the options relevant to program preparation are discussed here:

- DBPROTOCOL(DRDA)

  Only this option allows to invoke remote stored procedures; DBPROTOCOL(PRIVATE) must not be used for this purpose.

- ENCODING

  This option controls the encoding scheme that is used for static SQL statements in the package and sets the initial value of the CURRENT APPLICATION ENCODING SCHEME special register. The default ENCODING value for a package that is bound at a remote DB2 for z/OS server is the system default for that server. The system default is specified at installation time in the APPLICATION ENCODING field of panel DSNTIPF. For applications that execute remotely and use explicit CONNECT statements, DB2 uses the ENCODING value for the plan. For applications that execute remotely and use implicit CONNECT statements, DB2 uses the ENCODING value for the package that is at the site where a statement executes.

BIND PLAN options

- DISCONNECT

  Determines which remote connections to destroy during commit operations. The option applies to any application process that uses the plan and has remote connections of any type.

  For most flexibility, employ DISCONNECT(EXPLICIT), explicitly or by default. This option requires you to manually release connections with the RELEASE(CONNECTION) statement in order to destroy them.

  The other values of the option are also useful.

  - *DISCONNECT(AUTOMATIC)* - Destroys all remote connections during a commit operation, without the need for explicit RELEASE statements in your program.
  - *DISCONNECT(CONDITIONAL)* - Destroys all remote connections during a commit operation, except when an open cursor defined as WITH HOLD is associated with the connection.

- DBPROTOCOL

  Only DBPROTOCOL(DRDA) is accepted for programs calling stored procedures.

- ENCODING

  This option controls the encoding scheme that is used for static SQL statements in the plan and sets the initial value of the CURRENT APPLICATION ENCODING SCHEME special register. For applications that execute remotely and use explicit CONNECT statements, DB2 uses the ENCODING value for the plan. For applications that execute remotely and use implicit CONNECT statements, DB2 uses the ENCODING value for the package that is at the site where a statement executes.

18

Code level management

In this chapter we discuss the two most common issues with the management of stored procedures: How to version stored procedures in a DB2 subsystem and how to promote stored procedures from development to production. This chapter focuses on external stored procedures built in COBOL, C, PL/I, etc., and external stored procedures built with the SQL Procedures language. For management of Java stored procedures, refer to Chapter 13, “Java stored procedures” on page 181. For management of native SQL procedures, refer to Chapter 15, “Native SQL procedures” on page 253.

This chapter contains the following:

- Environments and code levels
- Versioning of stored procedures
- Promotion of stored procedures
- Notes on REXX execs

18.1 Environments and code levels

In a typical IT world, it is possible to have multiple environments, each with multiple releases of application code and stored procedures. Most organizations maintain at least three environments: development, quality and production. Within each environment there may be multiple code levels to support parallel development and testing needs of the customers. A level represents code at one particular application release.

Table 18-1 shows an example of different environments and levels for an application. As shown, each level is designated for a specific purpose.

Table 18-1 Sample DB2 environments and code levels

Environment name	Code level	Purpose
Development	D01	New code
	D02	Bug fix
Quality	Q01	Functionality test
	Q02	Integrated test
	Q03	Volume test
	Q04	Performance test
Production	P01	Release to limited users
	P02	Release to all users

Each DB2 subsystem maintains a set of catalog tables which contain details about stored procedures, packages and other pertinent information about DB2 objects. When an application invokes a stored procedure, DB2 accesses these catalog tables. Like other DB2 objects, it is recommended to make unqualified references to stored procedures within the application for easy portability. Therefore, it is important to understand the relationship between the application code at different release levels and a DB2 subsystem. Depending on the size and activity of your site, multiple environments can be mapped to a single DB2 subsystem, or each environment can be mapped to multiple DB2 subsystems. Note that the entire discussion in this section is with regard to a single application and the management of code levels for that application. Normally, DB2 subsystems are shared by multiple applications,

We now look at some of the common configurations. In each configuration we assume that there is a development, quality and production environment.

- One DB2 subsystem per environment

  In this configuration we have the three environments with one DB2 subsystem per environment. There is no sharing of code or data across environments. Figure 18-1 shows an example of this type of configuration. Notice that there can be multiple code levels within each environment.

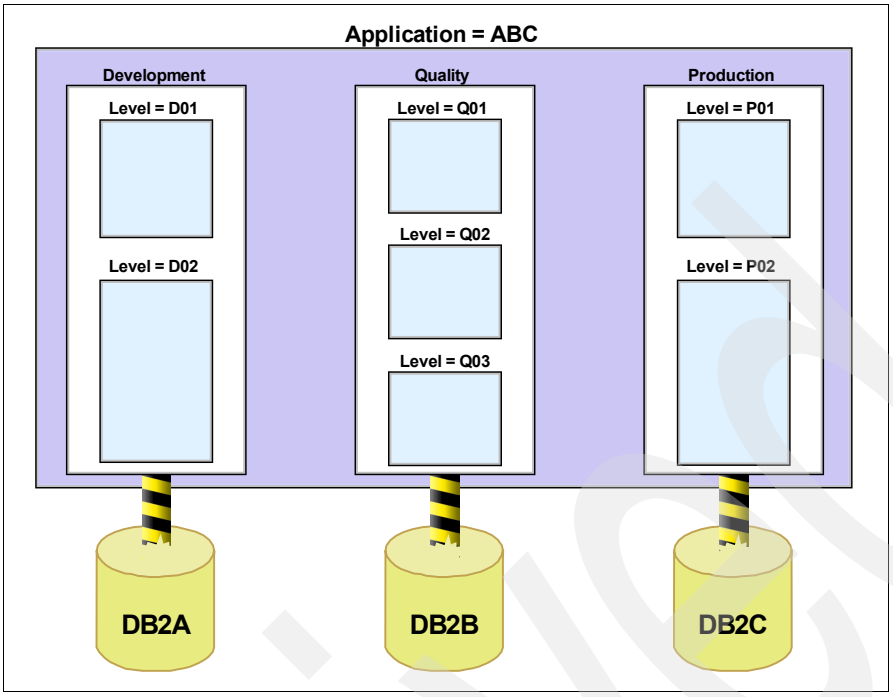

Figure 18-1 One DB2 subsystem per environment

- One DB2 subsystem for two or more environments

  In this configuration we have the same three environments, but two of the environments share the same DB2 subsystem, while the third environment has its own DB2 subsystem. There is no sharing of code across environments, but the development and quality environments use the same data. Figure 18-2 shows an example of this type of configuration. Notice that there can be multiple code levels within each environment.

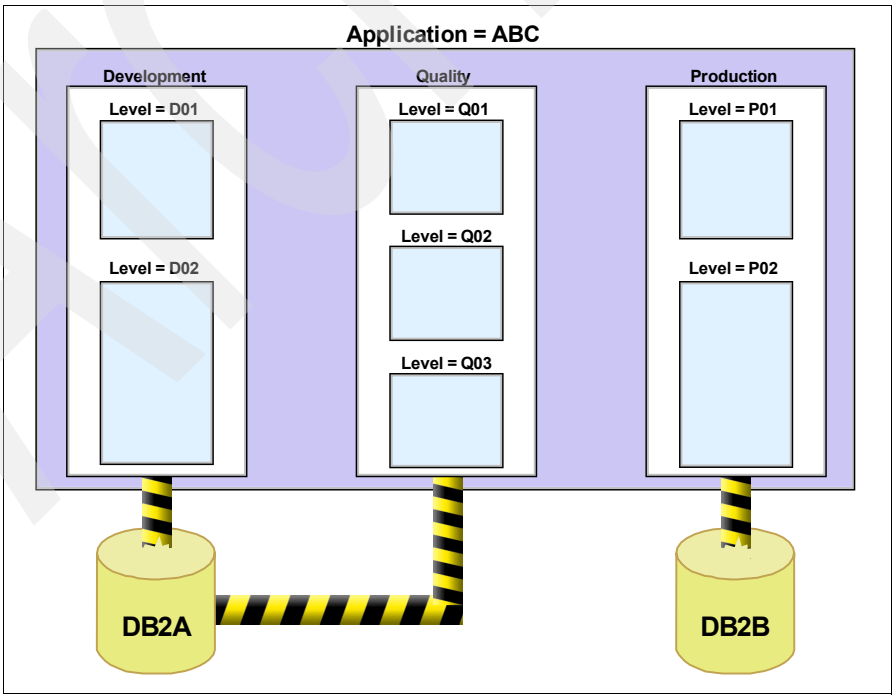

Figure 18-2 One DB2 subsystem for two or more environments

- One DB2 subsystem for one or more levels of an environment

  In this configuration we have the same three environments. Two of the environments each have their own dedicated DB2 subsystem. The third environment has two DB2 subsystems, the first one used for the first two code levels, with a second subsystem used for the third code level. In this configuration there is no sharing of code or data across environments, while there is some sharing of data across two of the code levels in the quality environment. Figure 18-3 shows an example of this type of configuration. Notice that there can be multiple code levels within each environment.

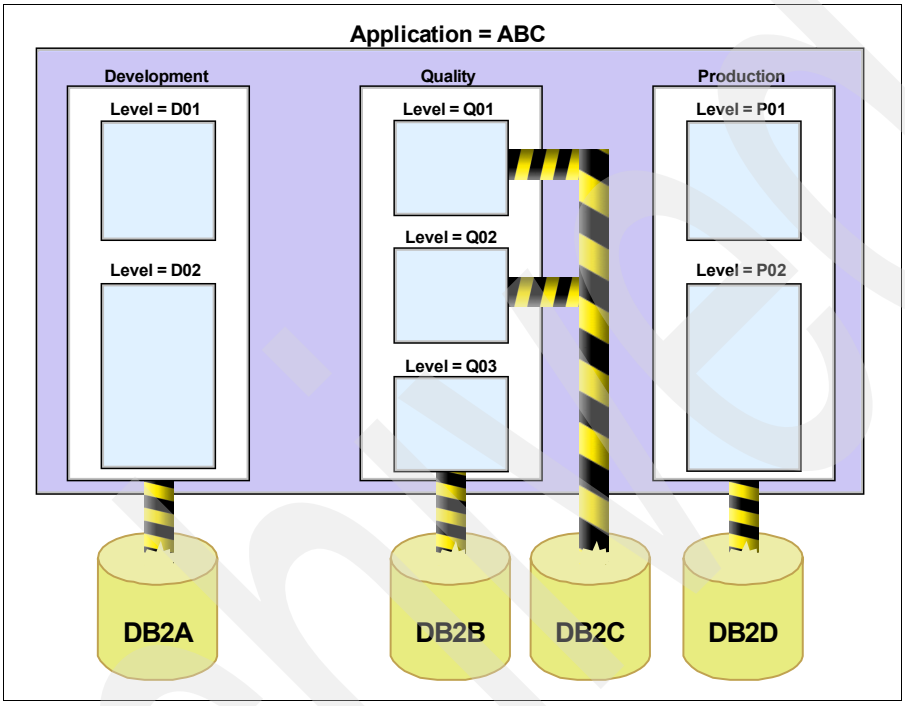

Figure 18-3 One DB2 subsystem for one or more levels of an environment

Once the code is developed and tested, it will be promoted to production to serve the intended business functionality. Depending on the complexity of the environments, configuration management of stored procedures can vary from a simple process to a most complex one. Therefore, each site should have proper change management procedures in place to ensure a smooth operation. Whatever the complexity of the environments, there are two challenges that exist for the configuration management of stored procedures:

- Maintaining different versions of stored procedures within a DB2 subsystem
- Promoting stored procedures from the development environment to the production environment

These challenges can be quite complicated if your site supports different types of stored procedures like external (written in COBOL, C, PL/I, etc.), SQL Procedures language and Java. Native SQL procedures introduced in DB2 9 for z/OS have additional versioning capabilities that we will discuss.

The following sections provide an approach to solve these two issues.

18.2 Versioning of stored procedures

DB2 uses three variables to identify and execute a stored procedure at runtime: procedure name, package name and load module name. In order to distinguish between different versions of a stored procedure, we need to “qualify” the three variables. Let us study what happens when a SQL CALL is made from an application to the point where the stored procedure gets executed. There are several logical instructions happening; in this section we just focus on the steps showing the relationship among the three variables for the following CALL statement:

```
EXEC SQL
CALL PROC1 (:PARM1, :PARM2, :PARM3)
END-EXEC.
```

Figure 18-4 summarizes the connection between the variables at CALL time.

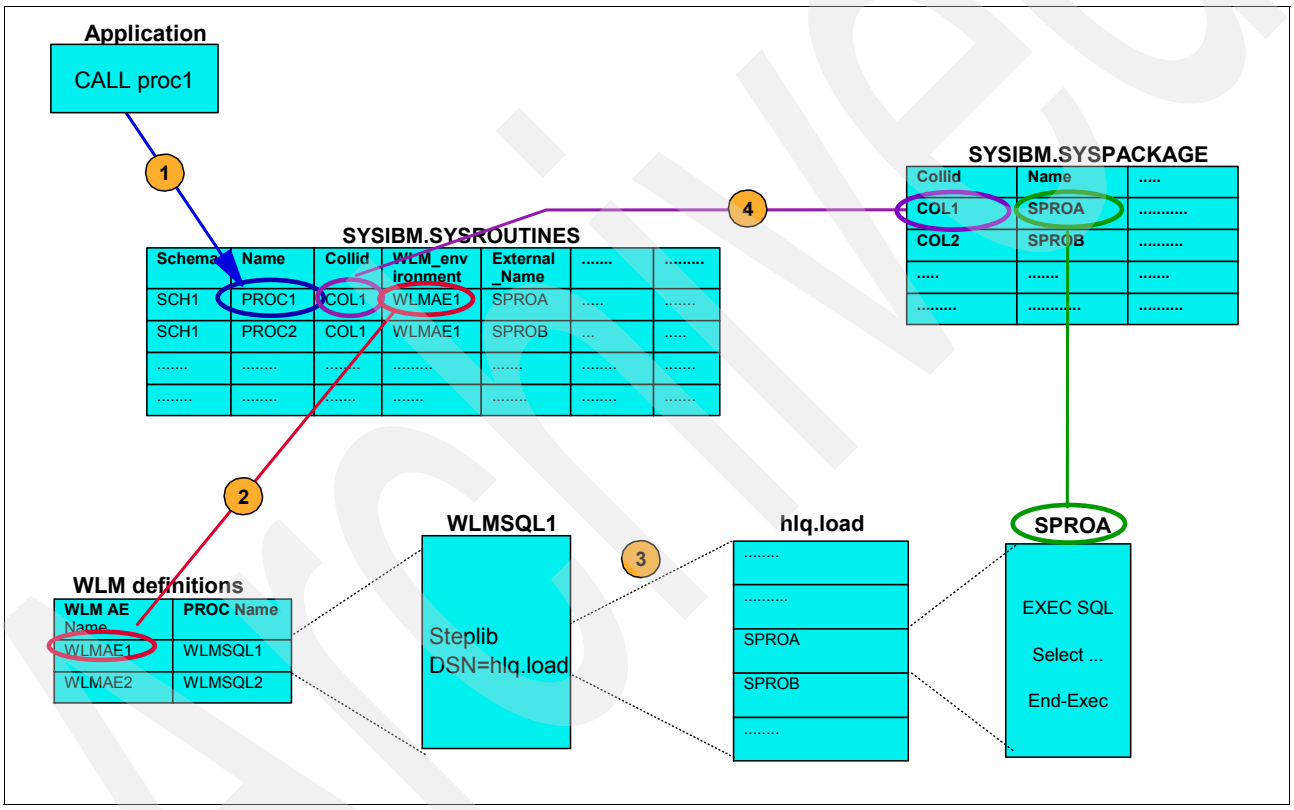

Figure 18-4 Relationship between SCHEMA, COLLID and WLM Application Environment at runtime

These are the steps to locate and execute a stored procedure:

1. DB2 searches for stored procedure PROC1 in the SYSIBM.SYSROUTINES catalog table and retrieves COLLID and WLM_ENVIRONMENT. DB2 locates PROC1 based on the SCHEMA, which can be passed either as a BIND parameter of the caller or at runtime with the SET PATH statement within the caller. In this case a COLLID value of COL1 and a WLM_ENVIRONMENT value of WLMAE1 are retrieved. The external name of the stored procedure is SPROA.
2. DB2 sends a request to WLM to schedule stored procedure PROC1 in an address space associated with the application environment identified by the WLM_ENVIRONMENT name retrieved from SYSIBM.SYSROUTINES. In this case the address space is WLMSQL1.

3. WLM address space WLMSQL1 is started, and it locates load module SPROA using the load library data sets specified in the STEPLIB (or from JOBLIB or from link-list).

> **Note:** The external name of a stored procedure cannot be greater than eight characters due to a restriction on z/OS on length of program name and member of load module. If possible, it is recommended to have the name of the stored procedure also be no more than eight characters, and the stored procedure name should be the same name as the external name. We used different names for the purpose of demonstrating the relationship between the two.

4. While executing the module SPROA, when the first SQL statement is encountered, DB2 locates the package SPROA under collection ID COL1 in SYSIBM.SYSPACKAGE.

> **Note:** If the stored procedure is defined with `NO COLLID`, then DB2 uses the collection ID of the caller.

As shown in the above steps, SCHEMA, COLLID, and the WLM_ENVIRONMENT name can be used to locate and execute a unique stored procedure.

Table 18-2 summarizes the relationship between the variables DB2 uses to locate a stored procedure and the qualifiers it uses.

Table 18-2 Stored procedure variables and their qualifiers

Variable	Qualifier	Description
Name	Schema	A schema qualifies a stored procedure.
Package Name	Collection ID	Collection ID can be used to qualify a package name associated with a stored procedure. For some reasons, if you have to use the same COLLID then packages can be distinguished by versioning them.[a]
Load module	WLM Application environment name	Load module resides in a PDS or PDSE data set. By keeping multiple versions of a load module in different load libraries and each load library concatenated to different JCL procedures associated with a WLM application environment, we can maintain multiple versions of a load module. To make it simple, the application environment can qualify the load module.

a. Package versioning is not possible for environments where compilation is done only once.

So, by defining a stored procedure with unique combinations of the above three qualifiers, multiple versions of stored procedures can be maintained within the same DB2 subsystem. Any combination of the three variables is possible. However, it is strongly recommended to have a one-to-one relationship between SCHEMA, COLLID and WLM_ENVIRONMENT of a stored procedure. The following section provides an example of how to access multiple versions of a stored procedure in the same DB2 subsystem.

18.2.1 Four release levels: Sample scenario

For example, consider a scenario where you have an application at four code release levels, say Q01, Q02, Q03, and Q04, and all the levels are accessing the same DB2 subsystem, DB2Q. The stored procedure name is PROC1 and the calling program name is CALLDR. At any point in time, the user should have the flexibility to invoke any version of the stored

procedure. To make this possible, the following preparations are required to create multiple versions of the stored procedure and multiple versions of the calling program.

Stored procedure preparation

These are the program preparation steps for the four versions of the stored procedure:

1. Make four definitions of stored procedure PROC1 with a unique combination of SCHEMA, COLLID, and WLM ENVIRONMENT name as shown in Example 18-1. Note that the stored procedure name (without qualifiers) is the same in all four cases and the external name is the same in all four cases.

Example 18-1 DDL to create four code levels of the same stored procedure

```
CREATE PROCEDURE SCH1.PROC1
(
parameter list...
)
EXTERNAL NAME SPROA
WLM ENVIRONMENT DB2QWL1
COLLID COL1 ;

CREATE PROCEDURE SCH2.PROC1
(
parameter list...
)
EXTERNAL NAME SPROA
WLM ENVIRONMENT DB2QWL2
COLLID COL2 ;

CREATE PROCEDURE SCH3.PROC1
(
parameter list...
)
EXTERNAL NAME SPROA
WLM ENVIRONMENT DB2QWL3
COLLID COL3 ;

CREATE PROCEDURE SCH4.PROC1
(
parameter list...
)
EXTERNAL NAME SPROA
WLM ENVIRONMENT DB2QWL4
COLLID COL4
```

2. Compile and link-edit the stored procedure PROC1. Ensure that the load module SPROA is stored in four different load libraries. For instance:

```
for level Q01, the stored procedure load library is 'hlq.ABC.DB2QWL1'
for level Q02, the stored procedure load library is 'hlq.ABC.DB2QWL2'
for level Q03, the stored procedure load library is 'hlq.ABC.DB2QWL3'
for level Q04, the stored procedure load library is 'hlq.ABC.DB2QWL4'
```

   The SPROA load module now exists in all the above libraries. Each library will contain the load module for the version of the source code that was compiled and link-edited to that library.

3. Bind the package SPROA with different collection IDs. The collection ID here should match the collection ID specified in the CREATE PROCEDURE statement. If the stored procedure is defined with NO COLLID, then the collection ID should match the collection ID of the caller. For instance:

```
BIND PACKAGE(COL1) MEMBER(SPROA) .....

BIND PACKAGE(COL2) MEMBER(SPROA) .....

BIND PACKAGE(COL3) MEMBER(SPROA) .....

BIND PACKAGE(COL4) MEMBER(SPROA) .....
```

4. Define four WLM application environments: DB2QWL1, DB2QWL2, DB2QWL3 and DB2QWL4. For each application environment, have a corresponding JCL procedure. In the JCL procedure, concatenate the appropriate stored procedure load library. For instance:

```
JCL procedure for DB2QWL1 will have 'hlq.ABC.DB2QWL1' in steplib.
JCL procedure for DB2QWL2 will have 'hlq.ABC.DB2QWL2' in steplib.
JCL procedure for DB2QWL3 will have 'hlq.ABC.DB2QWL3' in steplib.
JCL procedure for DB2QWL4 will have 'hlq.ABC.DB2QWL4' in steplib.
```

At the conclusion of the above steps there are now four versions of each component required for stored procedure PROC1.

Calling program preparation

These are the program preparation steps for the four versions of the calling program:

1. Compile and link-edit program CALLDR (which calls stored procedure PROC1). Ensure that the CALL statement in program CALLDR is left unqualified. Note that you only need one version of the load module for the calling program.
2. Bind program CALLDR with the PATH option. Specify the schema name of the stored procedure in the PATH option. Depending on the value specified in the PATH option, DB2 invokes the corresponding stored procedure at runtime. It is also possible to set the value of PATH at runtime using the SET PATH statement in program CALLDR. This will allow you to maintain four versions of the stored procedure simultaneously and determine which version to call based on the value of the CURRENT PATH special register at runtime.

Table 18-3 shows that the procedure name remains the same across all code levels, while the schema, which is used as the qualifier, and the collection ID change. Note that the package name and external name are also the same across all four versions.

Table 18-3 Sample naming convention for versioning of a stored procedure in an environment

Level	Procedure name	Schema	Package name	Collection ID	External name	WLM application environment name
Q01	PROC1	SCH1	SPROA	COLL1	SPROA	DB2QWL1
Q02	PROC1	SCH2	SPROA	COLL2	SPROA	DB2QWL2
Q03	PROC1	SCH3	SPROA	COLL3	SPROA	DB2QWL3
Q04	PROC1	SCH4	SPROA	COLL4	SPROA	DB2QWL4

The following SQL query will extract the different versions of a particular stored procedure from the catalog:

```
Select Name, Schema, Collid, External_name, WLM_environment
from SYSIBM.SYSROUTINES
where name ='PROC1';
```

Figure 18-5 shows how the application ABC at multiple release levels (Q01 to Q04) can access the same DB2 subsystem (DB2Q) and distinguish multiple versions of a stored procedure based on the schema.

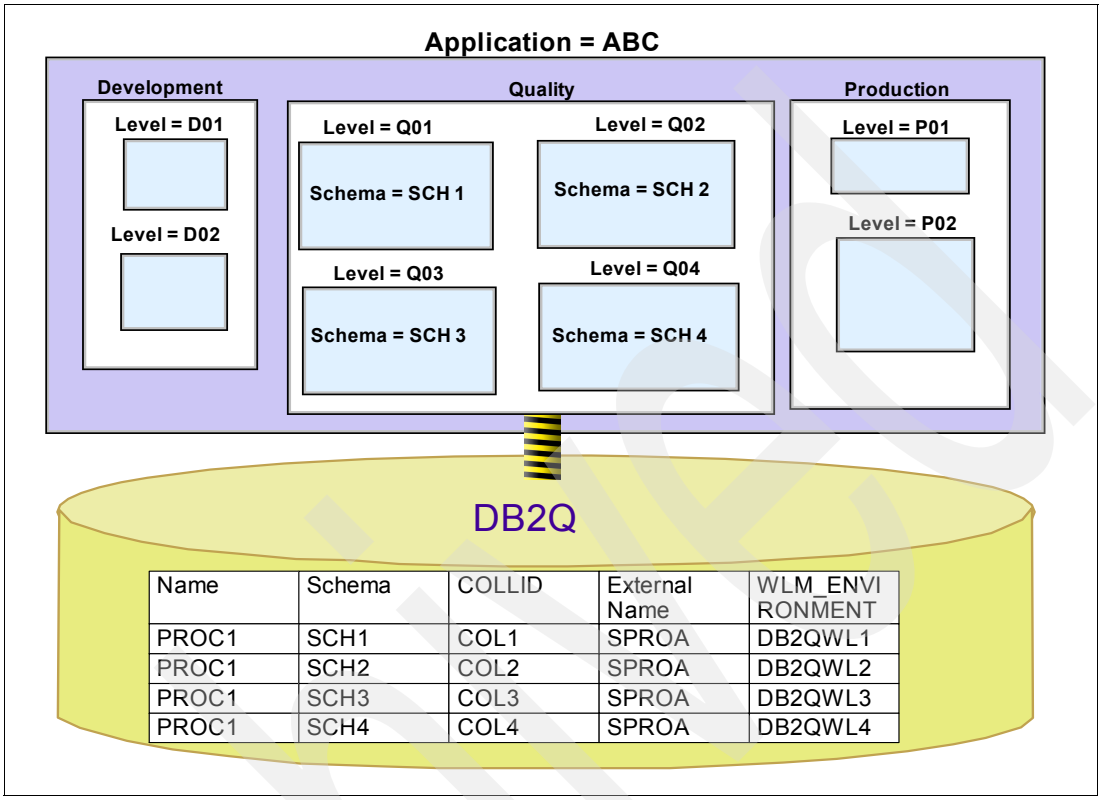

Figure 18-5 Sample versioning of stored procedures in a DB2 subsystem

18.2.2 Versioning of native SQL language stored procedures

Native SQL language stored procedures, introduced in DB2 9 for z/OS, have their own versioning capabilities. The DDL to create a native SQL procedure includes keywords to add a version, activate a version, replace the active version and drop a version, among other possibilities. DB2 9 for z/OS is able to maintain multiple versions of native SQL procedures in the DB2 catalog without having to vary the schema or collection ID. For more details on versioning of native SQL procedures, see 15.3, “Versioning” on page 291.

18.3 Promotion of stored procedures

Once the code is developed and tested, it is promoted to production. Depending on the requirements, code may need to be promoted across levels within an environment, or to the next higher environment and ultimately to production.

What makes stored procedure promotion different from other programs?

Stored procedures consist of two parts: One is source code and the other is DDL (the CREATE PROCEDURE statement). Depending on the type of stored procedure (external high-level language, external SQL language, or native SQL language) the two parts may exist as two components or one component.

- External high-level language: Source code + DDL (CREATE PROCEDURE statement).

- External SQL language: DDL only (source code is part of the CREATE PROCEDURE statement).
- Native SQL language: DDL only (source code is part of the CREATE PROCEDURE statement).

Promotion of stored procedures depends on the combination of two categories:

- Configuration management or change management policy of your site
  - Compile only once in development and copy all components
  - Compile in each environment (or code level)
- Type of stored procedure
  - External high-level language
  - External SQL language
  - Native SQL language

The following sections explain in detail the necessary steps in the promotion of stored procedures, depending on the change management policy and the type of stored procedures.

18.3.1 Compile just once

The “compile just once” promotion policy assumes that, once you have completed the final test phase, you would like to move the components for the stored procedure to the next code level. In other words, you don’t want to compile it again. We differentiate between the promotion steps for external high-level language, external SQL language, and native SQL language stored procedures.

External high-level language stored procedures

Stored procedures that are developed in high-level languages such as COBOL, C, PL/I, etc., fall in this category.

Figure 18-6 shows the development activities (on the left side of the picture) and the production promotion activities (on the right side of the picture) for the “compile just once” method for external high-level language stored procedures. We list the development activities and promotion and/or installation activities below.

Development activities

1. Define the stored procedure using the CREATE PROCEDURE statement.
2. Precompile, compile and link-edit the source code, which produces a load module and a DBRM.
3. Bind the DBRM to produce a DB2 package.
4. Refresh the WLM application environment.
5. Test and verify the stored procedure functionality.

Promotion and installation activities in target environment or code level

1. Copy the DDL (CREATE PROCEDURE statement).
2. Modify the DDL to reflect the new schema, new collection ID and new application environment (WLM AE), corresponding to your promotion code level or environment.

   A sample REXX DDLMOD and sample job DDLMODJB can be found in the Additional material. See the notes inside the DDLMOD source code file on its usage.
3. Define the stored procedure using the modified DDL. IBM-supplied programs DSNTIAD or DSNTEP2 can be used.

Ensure that SCHEMA, COLLID, and WLM AE correspond to the new environment or code level.

4. Copy the DBRM and bind the DBRM to produce a DB2 package.

   Ensure that the collection ID of the BIND statement and COLLID of the CREATE PROCEDURE statement are the same.

5. Copy the load module and refresh the WLM AE.

Tip: A batch job can be built with all the above steps for repeated executions. IBM-supplied sample job DSN8ED6 can be used to refresh WLM AE. Refer to the DSNTEJ6W job in the hlq.SDSNSAMP data set to set up the WLM_REFRESH job.

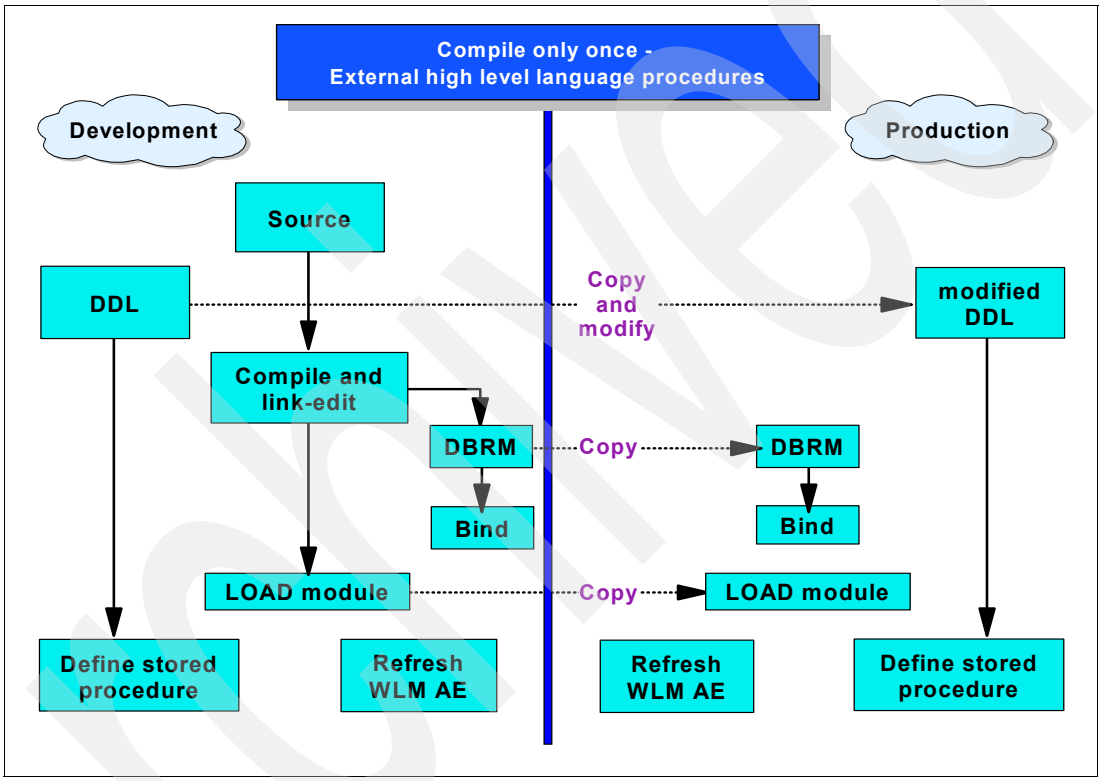

Figure 18-6 Promotion of external high-level language stored procedures - Compile only once

External SQL language stored procedures

Stored procedures built in the SQL Procedures language and then prepared to run as C programs fall into this category. External SQL language stored procedures are mainly built either using Developer Workbench (DWB) or its successor, IBM Data Studio, from workstations. It is also possible to build them on z/OS using the DSNTPSMP stored procedure in batch mode. For traditional developers on a z/OS server, it is possible to build them using an ISPF editor and then prepare, precompile, compile and link-edit the procedure using JCL. The source code of SQL stored procedures, built using either DWB or IBM Data Studio or DSNTPSMP, is stored in DB2 catalog tables (SYSROUTINES_SRC and SYSROUTINES_OPTS).

If you developed your stored procedures using the Stored Procedure Builder or the Development Center, then you could use the DB2Build utility, which executes on the client side (UNIX and Windows), to promote your external SQL language stored procedures. But this utility recompiles the program. Since your site's policy is to compile *just once in*

development, you used to need a procedure on z/OS to promote stored procedures without using Stored Procedure Builder, Development Center or DB2Build. Now you can use the Binary Deploy capability in Developer Workbench or IBM Data Studio to promote external SQL language stored procedures without recompiling. The binary deploy capability will copy all necessary components from one environment to another and perform the bind in the target environment. This way you do not need to recompile.

Figure 18-7 shows the actions that are taken by Developer Workbench or Data Studio when you are in the Deploy Wizard and click the option **Deploy using binaries**. We list the development activities and promotion or installation activities below.

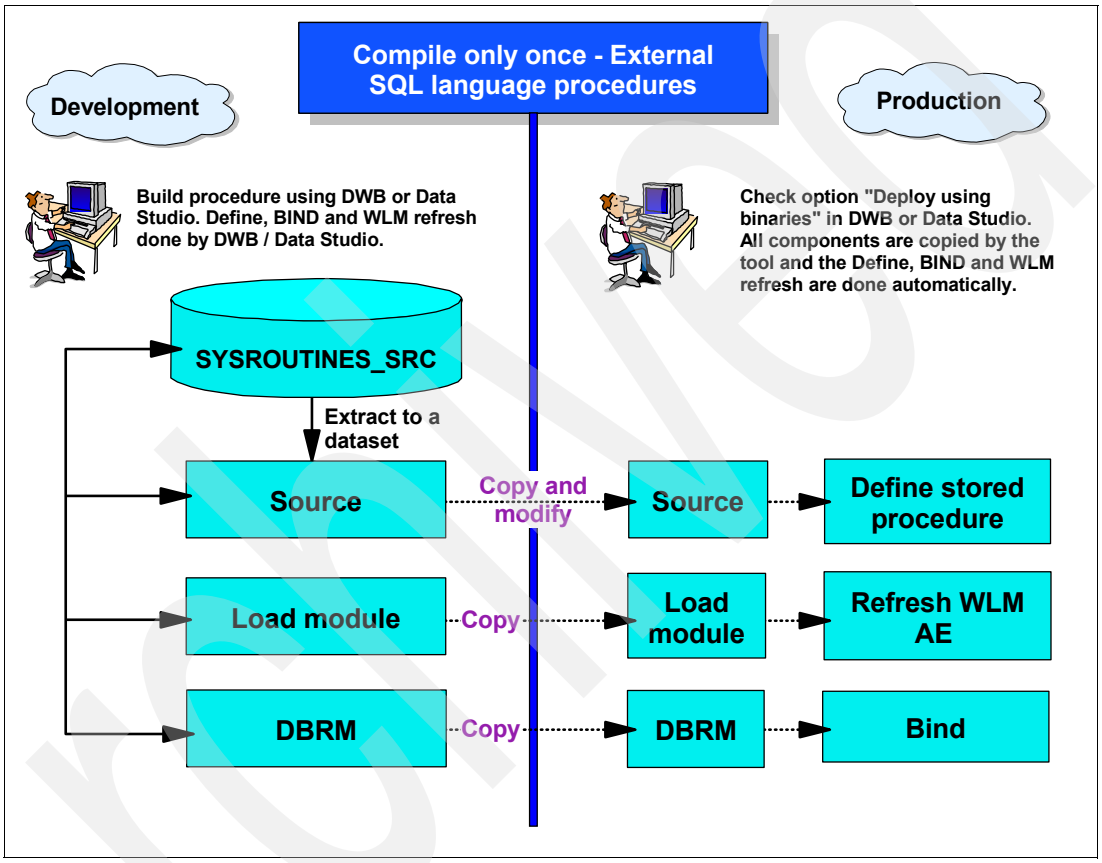

Figure 18-7 Promotion of external SQL language stored procedures - Compile only once

Development activities

1. Build an SQL procedure using either Developer Workbench or Data Studio.
2. Follow the steps in the Developer Workbench or Data Studio to deploy the stored procedure in your development environment.
3. Test and verify the stored procedure functionality.

Promotion and installation activities in target environment or code level

Developer Workbench and Data Studio can perform a function called binary deploy, which enables you to deploy external SQL language stored procedures from one environment to another without recompiling. See 27.7.8, “Deploying SQL or Java stored procedures without recompiling” on page 722 for details on the binary deploy process in Developer Workbench and Data Studio.

Native SQL language stored procedures

DB2 9 for z/OS introduced native SQL language stored procedures. These procedures, like external SQL language stored procedures, are written entirely in the SQL Procedures language. The difference is that they do not get generated into C code and they run in the DB2 engine instead of a WLM address space. Therefore, there are no load modules that need to be link-edited and there are no WLM address spaces that need to be refreshed. For details on deploying native SQL language stored procedures, see 15.5, “Deployment of a native SQL procedure to another server” on page 301.

18.3.2 Compile every time

The “compile every time” promotion policy assumes that, once you have completed the final test phase, you will use the same level of source code and go through the program preparation process to create new object code and a new DB2 package. In other words, you will go through the same program preparation steps as you did for your test environment. We differentiate between the promotion steps for external high-level language, external SQL language, and native SQL language stored procedures.

External high-level language stored procedures

Stored procedures that are developed in high-level languages like COBOL, C, PL/I etc., fall into this category.

Figure 18-8 shows the development activities (on the left side of the picture) and the production promotion activities (on the right side of the picture) for the “compile every time” method for external high level language stored procedures. We now list the development activities and promotion or installation activities.

Development activities

1. Define the stored procedure using the CREATE PROCEDURE statement.
2. Precompile, compile and link-edit the source code, which produces a load module and a DBRM.
3. Bind the DBRM to produce a DB2 package.
4. Refresh the WLM application environment.
5. Test and verify the stored procedure functionality.

Promotion and installation activities in target environment or code level

1. Copy the DDL (CREATE PROCEDURE statement).
2. Modify the DDL to reflect the new schema, new collection ID and new application environment (WLM AE) corresponding to your promotion environment/code level.

   Note: A sample REXX DDLMOD and sample job DDLMODJB can be found in the Additional material. See the notes inside the DDLMOD source code file on its usage.

3. Define the stored procedure using the modified DDL. IBM supplied programs DSNTIAD or DSNTEP2 can be used.

   Note: Ensure that SCHEMA, COLLID, and WLM AE correspond to the new environment/code level.

4. Copy the source code.

5. Pre-compile, compile and link-edit the source code, which produces a DBRM and a load module.
6. Bind the DBRM to produce a DB2 package.

> **Note:** Ensure that collection ID of the BIND statement, and COLLID of the CREATE PROCEDURE statement are the same.

7. Refresh the WLM AE.

> **Tip:** A batch job can be built with all the above steps for repeated executions. IBM supplied sample job DSN8ED6 can be used to refresh WLM AE. Refer to the DSNTEJ6W job in the hlq.SDSNSAMP data set to set up WLM_REFRESH job.

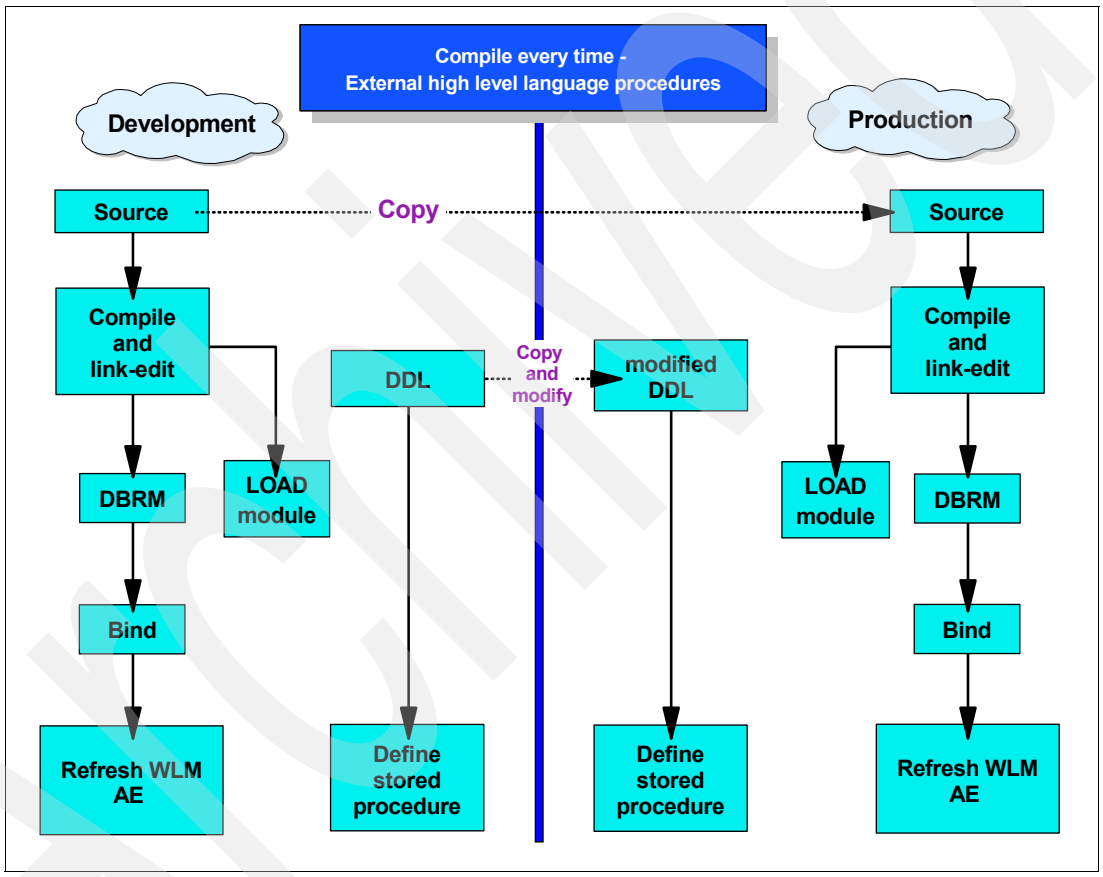

Figure 18-8 Promotion of external high level language stored procedures - Compile every time

External SQL language stored procedures

Stored procedures built in the SQL Procedures language and that are then prepared to run as C programs fall into this category. External SQL language stored procedures are mainly built either using Developer Workbench (DWB) or its successor IBM Data Studio from workstations. DWB and Data Studio provide deploy capabilities that make it easier for you to deploy your SQL language stored procedures, even when you want to recompile them in the target environment.

Figure 18-9 shows the development activities (on the left side of the picture) and the production promotion activities (on the right side of the picture) for the “compile every time”

method for external SQL language stored procedures. We list the development activities and promotion/installation activities below.

Development activities

The development activities for external SQL language stored procedures in the “compile every time” scenario are the same as in the “compile just once” scenario. The steps are as follows:

1. Build SQL procedure using either Developer Workbench or Data Studio.
2. Follow the steps in the Developer Workbench or Data Studio to deploy the stored procedure in your development environment.
3. Test and verify the stored procedure functionality.

Promotion and installation activities in target environment/code level

In the “compile every time” scenario, instead of checking the “Deploy using binary” box in the Deploy Wizard in Developer Workbench or Data Studio, check the “Deploy source to the database” box. This will copy the source code for the SQL language stored procedure from one database to another.

After you have copied the source code to the production environment, you can then use Developer Workbench or Data Studio to build the stored procedure in the production environment using the same steps you used in the development environment, as shown in Figure 18-9.

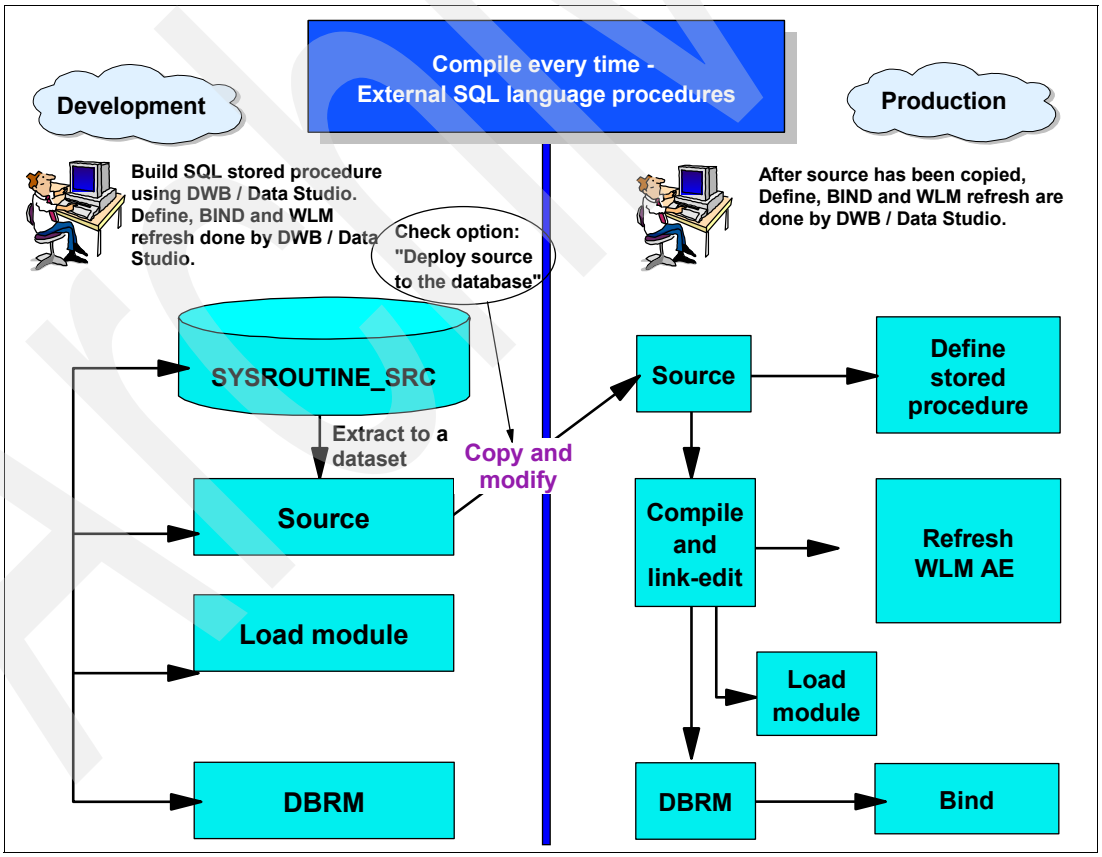

Figure 18-9 Promotion of external SQL language stored procedures - Compile every time

Native SQL language stored procedures

Since native SQL language stored procedures do not contain a load module and do not contain a TCB that runs in a WLM address space, there is no concept of “compile just once” vs. “compile every time.” For details on deploying native SQL language stored procedures, see 15.4, “Execution of a native SQL procedure” on page 299. Sample versioning and deployment scenarios for native SQL language stored procedures are shown in Chapter 15, “Native SQL procedures” on page 253.

18.4 Notes on REXX execs

Prior to the availability of Developer Workbench or Data Studio, if you had external SQL language stored procedures that you wanted to move from one environment to another, and you did not want to re-prepare the stored procedure in the production environment, you had to develop your own process to move the source code, load module, and DBRM.

In *DB2 for z/OS Stored Procedures: Give Them a CALL Through the Network*, SG24-7083, the authors describe some sample REXX execs, two of which, GETSQLSP and PUTSQLSP, assist with this process. However, since Developer Workbench and Data Studio provide the capability to deploy SQL language stored procedures using binaries, which perform the file movement process for you, there is no longer a need for using the GETSQLSP and PUTSQLSP REXX execs.

A third sample REXX exec, DDLMOD, is still useful for managing code for external high-level language stored procedures, so we include documentation for DDLMOD in this chapter.

18.4.1 DDLMOD

Functionality

The DDLMOD exec is designed to allow you to easily generate the CREATE PROCEDURE statement and supporting SQL statements for an existing stored procedure that is to be moved from one environment to one or more other environments. This REXX exec performs the source code and control statement copying function that is performed by the binary deploy function within the stored procedure development tools. DDLMOD performs the following two functions:

- Modifies the DDL based on the values specified in the configuration file.
- Generates SYSIN cards, which can be used for the WLM refresh job, the DROP PROCEDURE statement, and the SET CURRENT SQLID statement.

Input parameter

Level: A unique ID representing an environment or code level. This level is provided as part of the contents of the SYSTSIN DD statement after the name of the REXX exec. It is used to determine which record in the configuration file to use for this execution of the REXX exec.

Input data sets

- DDLINPSP: Input data set with the CREATE PROCEDURE statement to be modified
- CFGFILE: Configuration file with environment/code level details

Example 18-2 shows the sample content of the configuration file.

Example 18-2 Sample contents of the configuration file

```
Level/Env SSID      Schema   CollID    WLMAE     SQLID
```

```
--------- -----    --------  -------  ---------  --------
D01       DB9A      DSCH1    DCOL1     DB9AWLM   PROD7083
D02       DB9A      DSCH2    DCOL2     DB9AWL2   PROD7083
Q01       DB9A      QSCH1    QCOL1     DB9AWL1   PROD7083
Q02       DB9A      QSCH2    QCOL2     DB9AWL2   PROD7083
Q03       DB9A      QSCH3    QCOL3     DB9AWL3   PROD7083
Q04       DB9A      QSCH4    QCOL4     DB9AWL4   PROD7083
P01       DB9A      PSCH1    PCOL1     DB9AWL1   PROD7083
P02       DB9A      PSCH2    PCOL2     DB9AWL2   PROD7083
```

Output data sets

- DDLOUTSP: Contains the modified CREATE PROCEDURE statement
- SETSQLID: Contains the SET CURRENT SQLID statement
- DROPSP: Contains the DROP PROCEDURE statement
- WLMRFRSH: Contains SYSIN cards for the WLM refresh job

Usage notes

- DDLMOD REXX modifies the input DDL for the schema, the collection ID, and the WLM application environment name, using the values provided in the record in the configuration file that matches the level provided in the input parameter.
- If your site requires some more parameters to be modified between environments/code levels, the above REXX can be customized. What we provide is just a sample.
- The output produced in the SETSQLID, DROPSP, and WLMRFRSH data sets will help to automate the promotion process between environments. For example, SETSQLID will be useful if your site implements secondary authorization IDs, and you use them while defining the modified DDL. Similarly, DROPSP will be useful if you drop the stored procedure before you create it. WLMRFRSH will be useful if you want to refresh the target WLM address space in batch mode.

Sample job to invoke DDLMOD

Example 18-3 provides a sample job for invoking the REXX exec DDLMOD. Note that the input parameter level is D02, which could be an abbreviation for your Development Environment Level 2. This parameter value matches the Level/Env value in the second record in the configuration file. Therefore, the values in that record will be used to produce the modified statements.

Example 18-3 Sample job to invoke DDLMOD

```
//DDLMODJB JOB (999,POK),'DDL MOD JOB',CLASS=A,MSGCLASS=X,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
//*
//DDLMOD   EXEC PGM=IKJEFT01
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//*
//*INPUT DATASET WITH DDL
//DDLINPSP DD DISP=SHR,DSN=SG247083.CBL.DDL(EMPDSAMP)
//*CONFIGURATION FILE WITH ENVIRONMENT/LEVEL DETAILS
//CFGFILE  DD DISP=SHR,DSN=SG247083.DEVL.CLIST.CONFIG(APPLABC)
//DDLOUTSP DD DISP=SHR,DSN=SG247083.DEVL.DDLMOD0(EMPDTL2C)
//SETSQLID DD DISP=SHR,DSN=SG247083.DEVL.DDLMOD0(SETSQLID)
//DROPSP   DD DISP=SHR,DSN=SG247083.DEVL.DDLMOD0(DROPSP)
//WLMRFRSH DD DISP=SHR,DSN=SG247083.DEVL.DDLMOD0(WLMRFRSH)
//*
```

```
//SYSEXEC  DD DISP=SHR,DSN=SG247083.DEVL.CLIST
//SYSIN    DD DUMMY
//SYSTSIN  DD *
%DDLMOD DO2
/*
//
```

We ran the DDLMOD exec using the sample JCL in Example 18-3. The output shown in the SYSOUT from running this job is shown in Example 18-4.

Example 18-4 SYSOUT produced from running DDLMOD REXX exec

```
READY
%DDLMOD DO2
Target variables are
 DB2 subsystem ID           :  DB9A
 Stored Procedure schema    :  DSCH2
 Stored Procedure collid    :  DCOL2
 WLM application environment:  DB9AWL2
 SQL ID                     :  PROD7083
READY
END
```

To see how the REXX exec works we can show you the CREATE PROCEDURE statement as it appears before and after running DDLMOD. Example 18-5 shows the DDL that was input to the DDLMOD job.

Example 18-5 CREATE PROCEDURE statement before running DDLMOD

```
CREATE PROCEDURE DEVL7083.EMPDSAMP
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
RESULT SETS 0
EXTERNAL NAME EMPDSAMP
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB9AWLM
STAY RESIDENT NO
COLLID DEVL7083
PROGRAM TYPE SUB
COMMIT ON RETURN NO ;
```

Example 18-6 shows the modified DDL after running the DDLMOD job. You can see that the schema name, WLM environment name, and collection ID have been changed to the values specified in the configuration file.

Example 18-6 CREATE PROCEDURE statement after running DDLMOD

```
CREATE PROCEDURE  DSCH2.EMPDSAMP
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
RESULT SETS 0
EXTERNAL NAME EMPDSAMP
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
 WLM ENVIRONMENT DB9AWL2
 RESIDENT NO
 COLLID DCOL2
 TYPE SUB
COMMIT ON RETURN NO ;
```

If you have many environments and code levels you can set up a configuration file with multiple records, like the sample file shown in Example 18-2, with one record for each environment or code level to which you would promote a stored procedure.

For external high-level language stored procedures, once you promote your stored procedures to production, further new versions of stored procedures do not necessarily require you to drop and create the procedure. You can just promote the program associated with the stored procedure for logical changes. You can issue an ALTER PROCEDURE statement, wherever permissible, to change the definition. However, if you need to alter the input and output parameters, you have to drop and recreate the stored procedure.

Part 4

Performance

In this part we discuss how an installation can manage the performance of the stored procedures it executes. After a general introduction, the discussion is organized into the themes of address space management and I/O management. Each theme is discussed in detail, and instrumentation to support analysis for that theme is also described. The intent here is to identify the key performance items which differentiate a stored procedure from a *normal* DB2 transaction and provide some recommendations.

The chapters are:

- Chapter 19, “General performance considerations” on page 391

  An introduction to performance characteristics, tools, and capacity planning of DB2 stored procedures
- Chapter 20, “Server address space management” on page 423

  It deals with workload management performance issues. Read this chapter if work is missing its performance goals and there is a significant stored procedures schedule wait time according to the accounting trace data.
- Chapter 21, “I/O performance management” on page 435

  We discuss abnormal I/O time. Read this chapter if work is not performing adequately and system level performance tools suggest the server address spaces are performing a significant amount of I/O.

19

General performance considerations

In this chapter we introduce the basic concepts of stored procedure performance. We describe the main components of the stored procedure execution, the most important performance parameters, and some capacity planning formulae.

This chapter contains the following:

- Performance concepts with stored procedures
- Monitoring and measuring stored procedure performance
- Recommendations

19.1 Performance concepts with stored procedures

To give application designers flexibility in designing their client/server applications, Distributed Relational Database Architecture™ (DRDA), and DB2 provide support for stored procedures. A stored procedure is an application program that is stored at the DB2 server and can be invoked by an application, either on a remote client or on the local system, through the SQL CALL statement. These are some of the advantages of using stored procedures:

- In a distributed environment, performance can often be improved by moving part of the application business or data access logic to the server. A single send and receive operation can suffice for a series of SQL statements, thus significantly decreasing the costs of distributed SQL processing.
- Some businesses do not want every workstation in the network to have detailed knowledge of the server's database design. Instead, they would rather have the clients access the server data through an interface program supplied by the server. In this way the server can change the database design and make corresponding changes to the interface program, without requiring changes to each of the client application programs.
- Some businesses prefer to divide the application design along organizational boundaries. For example, one part of the business might specialize in end-user interface applications, and another part in database processing.
- It is often easier to manage and maintain programs that run at the server. Consider the effort required to maintain one copy of a program at the server, compared to the effort required to maintain the same program on 100 client machines.

DB2 stored procedures enable the application designer to divide the application processing between the client and the server:

- Without stored procedures

  In a client/server application, the client performs all application processing and the server performs only database request (SQL) processing. With such an application, a network send and receive operation is required for SQL statements like INSERT, UPDATE, DELETE, and SELECT, while the SQL FETCH statement may only need one network send and receive operation per block of rows returned by the server. The elapsed time of an application may increase with the number of SQL statements, and it is heavily dependent on the network connection speed.

  There is also a certain amount of overhead associated with building the DRDA request and reply messages. The network send and receive operations, and the overhead associated with building messages increase the SQL path length for distributed applications when compared to local DB2 SQL applications.

- With stored procedures

  The SQL CALL statement allows local DB2 applications or remote DRDA applications to invoke stored procedures at a DB2 server. The client only has to issue a single network send and receive operation to run a stored procedure at the server, and the stored procedure can then issue multiple SQL statements. The use of stored procedures reduces the number of network send and receive operations, thus improving the elapsed time and CPU time consumed by an application. The SQL statements issued by a stored procedure use a local DB2 interface (RRSAF), so there is no additional distributed overhead on the SQL statements.

In order to take advantage of stored procedures, you need to understand their possible impact on your current environment, and identify the key parameters that can be used to optimize stored procedure performance.

19.1.1 The address spaces

When DB2 V4 introduced stored procedures, a new address space was added to the traditional ones utilized by the DB2 subsystem. The DB2 Stored Procedure Address Space (SPAS) is an allied address space dedicated to running stored procedures; it can be stopped independently from DB2, and allows separation and protection of DB2 from application errors. With DB2 V5, multiple stored procedure address spaces were supported, providing for greater scalability and flexibility in handling applications with different priorities. The support of multiple address spaces requires that the address spaces are managed by WLM in goal mode. With DB2 V8, support for new stored procedures was only available with WLM managed address spaces, however existing stored procedures could still run in the DB2 Stored Procedure Address Space (SPAS). With DB2 9 for z/OS, support for stored procedures in the SPAS has been removed. Figure 19-1 shows the types of address spaces that are typically active when stored procedures are being executed in DB2 9 for z/OS.

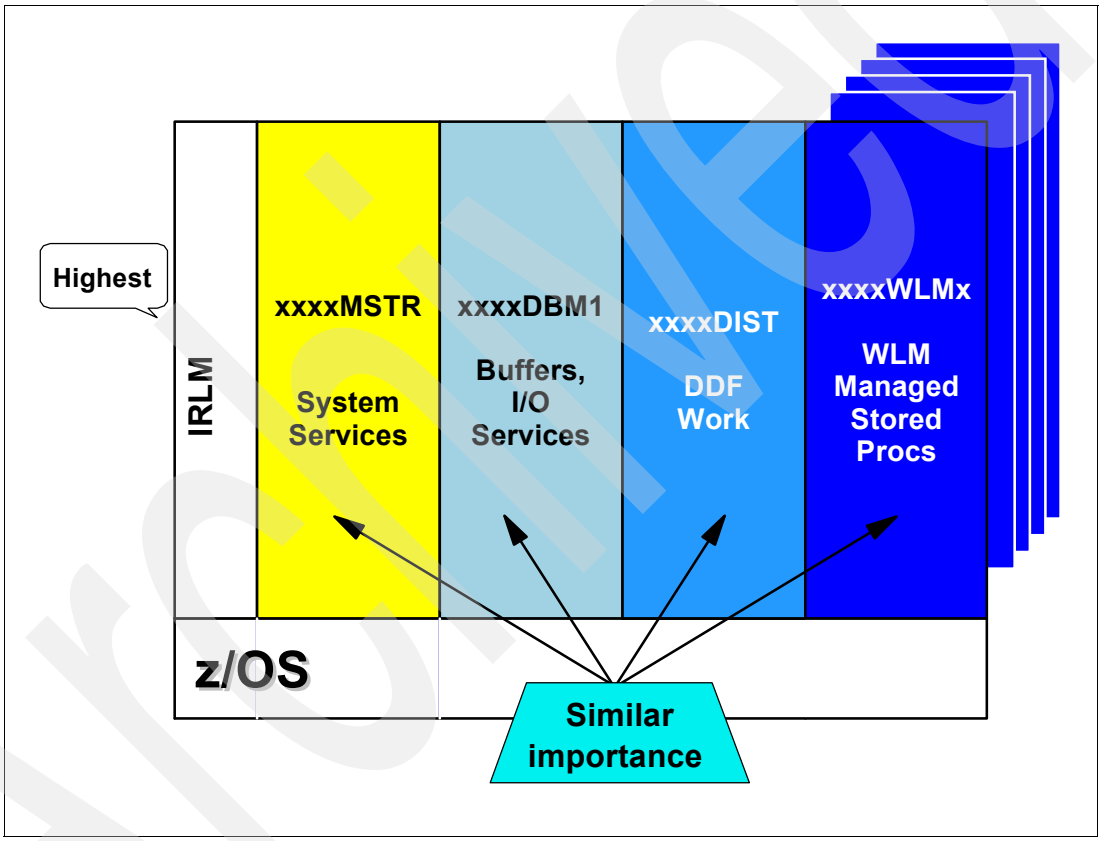

Figure 19-1 The DB2 address spaces with stored procedures

In each case xxxx represents your DB2 subsystem name.

The IRLM address space always needs to be assigned to a service class whose goal would give it one of the highest MVS dispatching priorities, usually SYSSTC will suffice. IRLM needs this priority because it manages resources that may or may not be accessed by work within the DB2 address spaces.

Next, the DB2 system services address space (MSTR), DB2 database services address space (DBM1), and the distributed data facility address space (DIST) should be assigned to service classes whose goals would be similar, but would result in an MVS dispatching priority just below the dispatching priority for the service class for the IRLM address space. The main reason for including DIST in this list is that it is considered a service address space to MVS

and thus it needs to have sufficient priority to get new work started into the address space. Once the work has been started, it will then be run under an independent WLM enclave which will be given a priority commensurate with the kind of work that it is performing. Do not make the mistake of having the DIST address space separate in priority from the other main DB2 address spaces.

Finally, the WLM-managed stored procedures address spaces themselves should be given a goal much like and probably similar to the goal for the other main DB2 address spaces (MSTR, DBM1, and DIST). These address spaces are MVS service address spaces and need the priority to get *new work* started quickly. In their case, the better the priority, the sooner a free TCB will get allocated to run a stored procedure. The lower the priority, the longer it will take. See 20.1.4, “WLM management of server address spaces” on page 427 for more information about classifying your workloads.

Note: In this chapter we only discuss performance considerations for WLM-managed stored procedures since DB2-managed stored procedures are no longer supported as of DB2 9 for z/OS.

Stored procedures that run in WLM-managed address spaces run at the same priority as the calling application. This ensures that the performance behavior of the stored procedure is synchronized with the application that calls it.

19.1.2 The execution life cycle of a stored procedure

In this section we differentiate between the life cycle of a stored procedure and that of a traditional DB2 transaction.

Figure 19-2 helps in identifying the additional steps of the stored procedure.

Assume that an application is running on a client, this application needs first to connect to DB2. DB2, then assigns a thread to the user and initiates an accounting process. Sometime during its execution, the application decides to issue an SQL CALL, providing options, and input and output parameters. The application waits while the stored procedure is executed; processing in the application will continue only when the stored procedure completes.

In the meantime, DB2 handles the CALL, retrieves information about the stored procedure from DB2 catalog table SYSIBM.SYSROUTINES (the WLM application environment name, the load module name, and the input and output parameter definitions), then passes the request to WLM. Most likely the information from the catalog will be cached, and no I/O will be necessary.

WLM puts the request in a queue for the application environment specified in the DB2 catalog. If an address space is already started and there is an available TCB, then the stored procedure is scheduled to run under one of the available TCBs. If there is no available TCB, or there is no address space started, then WLM will start another address space and the stored procedure will run under a TCB in the new address space. See Chapter 20, “Server address space management” on page 423 for more details on how WLM uses TCBs and address spaces to control execution of stored procedures.

DB2 will not have to create a new thread for the stored procedure because it runs under the thread of the caller.

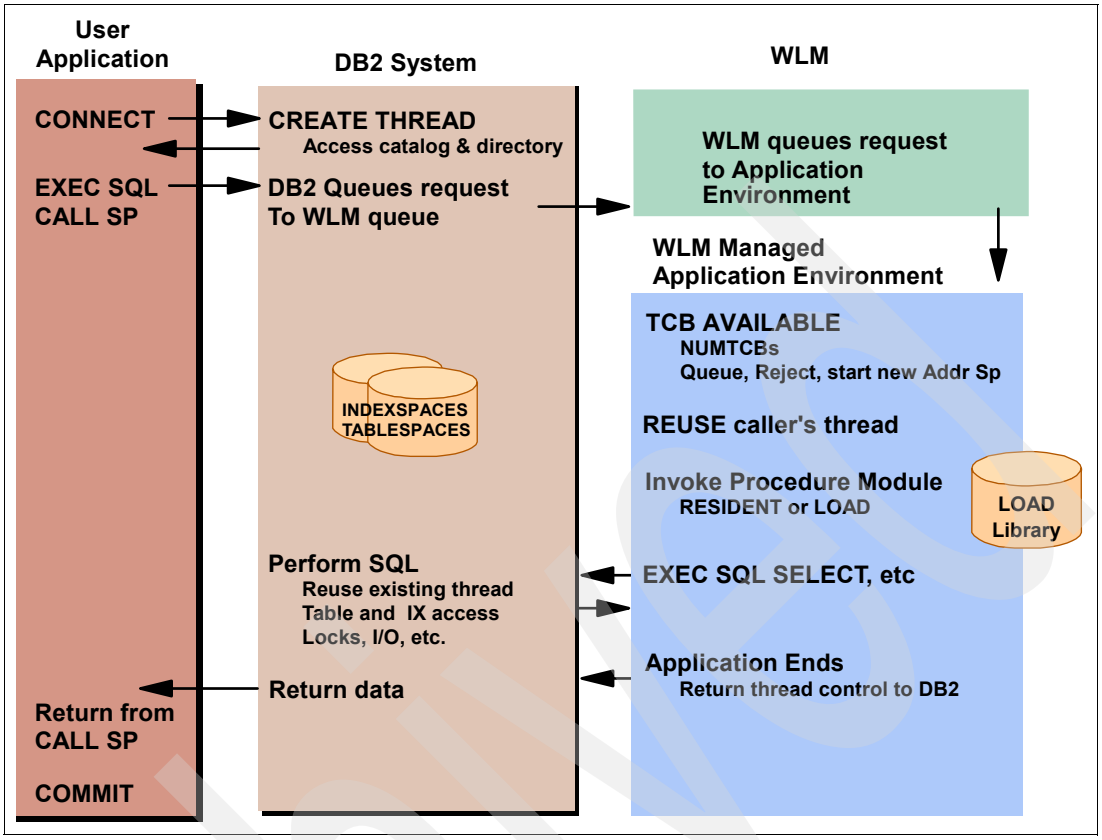

Figure 19-2 Stored procedure application life cycle

The stored procedure address space uses the Language Environment product libraries that are defined in the startup JCL for the address space to load and execute the stored procedure. If the stored procedure was defined with the STAY RESIDENT YES option, the load module may not need to be loaded.

Parameters are passed from the calling program to the stored procedure, according to the parameter definitions in DB2 catalog table SYSIBM.SYSPARMS. In addition, any LE runtime options specified when the procedure was created are in effect.

If the stored procedure contains SQL (remember, it does not have to!) then the DB2 package for the stored procedure is loaded into the EDM pool.

The stored procedure starts executing and issues SQL calls handled by the DB2 subsystem. The data returned by DB2 is moved by the Stored Procedure Manager to the output parameters. If the stored procedure is written to return result sets to the calling program, then the stored procedure will open the appropriate cursors, but not fetch any rows.

DB2 copies the output parameters received from the stored procedure to the client application parameter area, and returns control to the client application.

The calling program receives the output parameters and continues the same unit of work. If the stored procedure returned any result sets, then the calling program will fetch rows from the result sets and process those rows, all within the same unit of work. The calling program will then close the cursor for any result sets after fetching all the rows.

The client application issues a COMMIT statement, which commits the work done by the stored procedure and by the client application. A COMMIT, either issued within the stored

procedure or in the caller program upon return, commits all work up to that point in the UOW of both programs.

19.1.3 Stored procedure execution time components

Many components make up the total execution time of a DB2 for z/OS stored procedure, starting from the initial request, and ending when the thread terminates. In this section we show the components of the overall DB2 execution time for a stored procedure.

Figure 19-3 shows the components of the execution time for a stored procedure, including which components are counted in class 1 elapsed time and which components are counted in class 2 elapsed time.

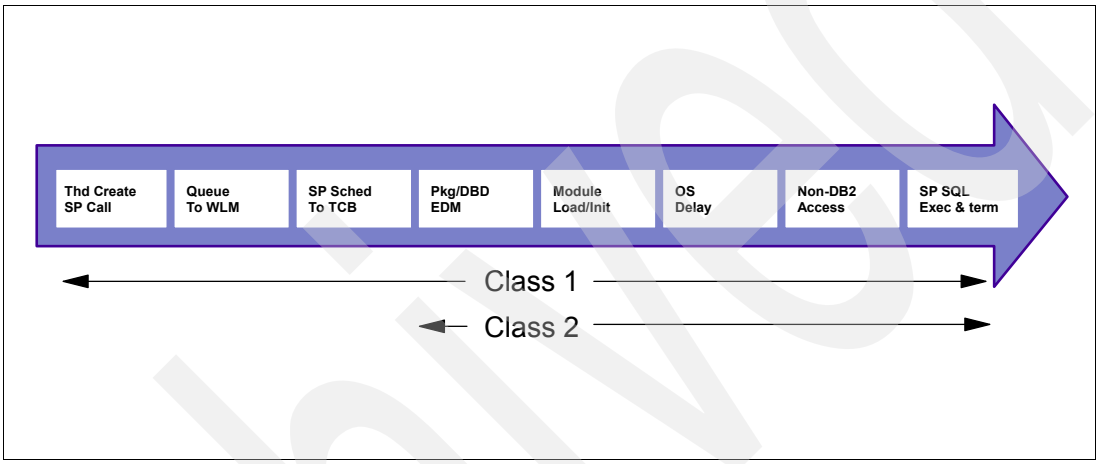

Figure 19-3 Where the execution time goes, including class 1 and class 2 breakdown

Here is a breakdown of each execution time component:

- Thread create on stored procedure call

  Every SQL statement must run under a thread in DB2. If a thread has not already been created when an SQL statement is encountered, then the thread must be created. If a stored procedure is called locally by a DB2 application that is also running on z/OS, then the stored procedure runs under the same thread as the calling program. If the stored procedure is called from a DRDA client program, then the cost of thread creation depends on whether or not a connection has already been established with DB2 for z/OS at the time of the call. If a connection has already been established, then the stored procedure runs under the same thread as the calling program. If a connection has not already been established, then there is a cost to create a thread for the stored procedure to execute under. The time to create the thread for a stored procedure called from a DRDA client can be impacted by whether you are running ACTIVE or INACTIVE threads. See “DB2 system parameter CMTSTAT” on page 417 for more details on ACTIVE and INACTIVE threads.

- Queue to WLM

  When you define your stored procedure, you specify the WLM application environment in which that stored procedure will run. The time required to queue the request to the WLM environment is included here.

- Scheduling the stored procedure into an address space with an available TCB

  If there is an available TCB in an existing address space for the associated application environment, then the stored procedure will use that TCB. If there is no available TCB, then WLM might start another address space depending on whether the application service class is meeting its performance and response time goals or not.

- Loading the package into the EDM pool

  Before your stored procedure can execute, the package for the stored procedure must reside in the EDM pool. If the package already exists in the EDM pool, then the time to locate it is very small. If the package is not found in the EDM pool then there is a delay to load it.

- Loading the stored procedure object code

  If the load module for the stored procedure has not already been loaded, then there is some delay associated with loading it. The STAY RESIDENT option can impact whether the stored procedure remains in storage. If the stored procedure is accessed frequently, then there is a high probability that it will remain in storage even if defined with STAY RESIDENT NO, and there will be no delay to load it.

- Operating system delay

  There may be some additional operating system or WLM workload delays depending on the level of CPU constraint on your system.

- Access to non-DB2 resources

  If your stored procedure accesses non-DB2 resources, such as VSAM files or IMS databases, then that also contributes to the execution time of the procedure.

- Stored procedure SQL execution

  The time spent executing SQL statements within the procedure is included here. This time will show up under package accounting time. See 19.2.2, “Reporting on DB2 accounting class 7 and 8 data” on page 402 for more details on DB2 accounting time for stored procedures.

  Note: There is also an execution time component for SQL processing associated with the stored procedure, but which occurs after returning to the calling application. This includes time spent in result set processing and time spent in commit processing, if COMMIT ON RETURN is specified. This time, as well as time associated with scheduling the stored procedure, is charged under the package name SYSSTAT. If you have stored procedures that execute a small amount of SQL and return large result sets, then it is not unusual to see the SYSSTAT package account for more time than the stored procedure package.

You can see that there are many components to the execution of a stored procedure. There are differences depending on whether the stored procedure is called from a distributed client through DDF, or if it is called locally by another application running on z/OS. You will need to monitor each of these components to ensure that your stored procedures are performing efficiently. We discuss in more detail the CPU costs of these components, and how to monitor these costs in the sections that follow.

19.1.4 Capacity planning

When planning for any new application, you need to consider the impact on system resources. In this section we provide some estimates of CPU times for an online transaction running locally on z/OS, for additional CPU when running distributed access to DB2, and for additional CPU when running a stored procedure instead of distributed access. All of these numbers were measured for DB2 for OS/390 and z/OS Version 7, and all are provided as ranges to account for variances in complexity of SQL and in bind options specified. CPU times are expressed in microseconds (µs) of zSeries 900 processor (z900), including CPU time for I/ O unless otherwise specified.

Figure 19-4 shows the improvements in performance from a single uniprocessor, expressed as a CPU multiplier. So if a process takes 1 second of CPU time on the z900, it runs .37 seconds on the z9.

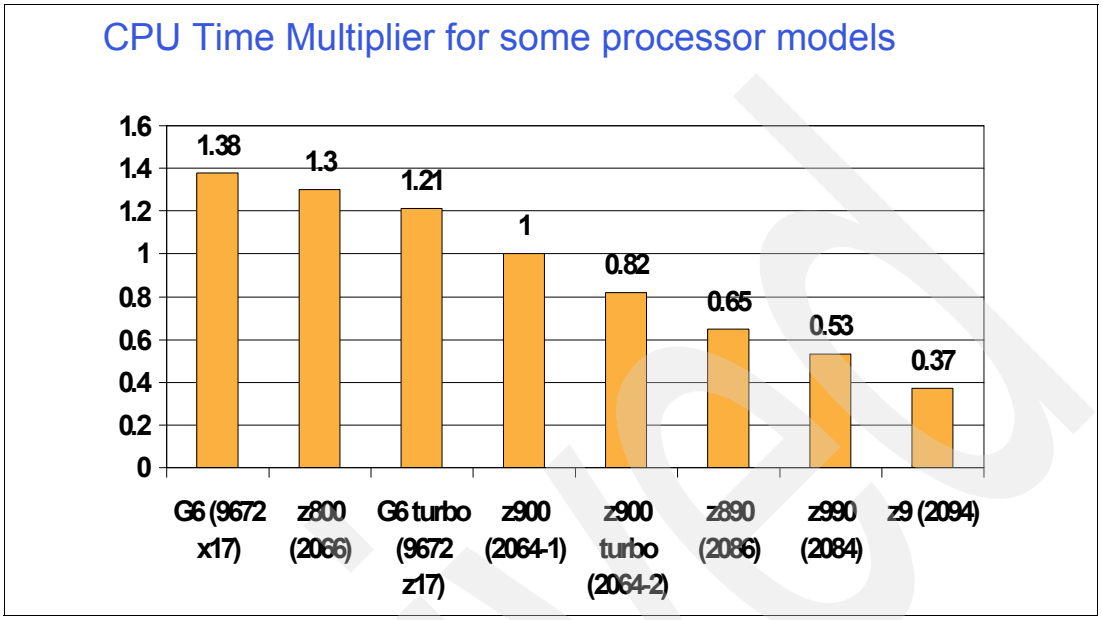

Figure 19-4 CPU multiplier across the evolution

DB2 9 for z/OS extensively uses the long-displacement instruction set that is available in z/Architecture® mode, which was introduced with 64-bit addressing. These instructions are available on the System z990, z890, and System z9 models in z/Architecture mode that exploit the long-displacement instruction set. Previous CPU models do not have such hardware support and are run in emulation mode. Therefore, some penalty in CPU overhead will occur when running DB2 9 for z/OS on System z900 and z800 models because the hardware support is simulated by microcode.

Estimates of CPU times for components of an online transaction running on the host:

- Read-only commit = 45 to 90 µs
- Update commit = 160 to 280 µs
- Create/Terminate Thread = 250 to 500 µs
  - Or, with thread reuse and release deallocate = 80 µs for signon
- Distributed Create/Terminate Thread = 2000 to 4000 µs
  - Or, inactive thread = 300 to 600 µs

What happens if the data is accessed from a distributed client? The CPU time estimates for DRDA access are:

- If block fetch not in effect = 210 µs for each SQL call
- If block fetch in effect = 5 to 10 µs for each Fetch SQL call:

  + 80 µs for each message
- + 300 to 600 µs for inactive thread scheduling per transaction (2000 to 4000 µs if Create/Terminate Thread)
- For each SQL call in DRDA non block fetch, add:

  28 to 56 µs + 170 µs message send/receive = 210 µs

- Block Fetch is enabled if:
  - The query is read-only.
  - Or, CURRENT DATA NO specified and an ambiguous cursor exists (dynamic SQL present).

What happens if the data is accessed through a stored procedure call from a distributed client? The CPU time estimates for a stored procedure access are:

- Stored procedure invocation CPU time = 220 to 560 µs

  + 170 µs for each message send/receive
- You can have between zero and two message send/receive transmissions for each stored procedure call: each message send and message receive takes 170 µs.
  - Zero messages sent or received if the stored procedure is local.
  - One message sent and none received if the stored procedure is defined with COMMIT ON RETURN and is WLM-managed (default = no commit on return): 170 µs.
  - Two messages incurred (one send and one receive) if the stored procedure is distributed, and is either defined with COMMIT ON RETUN NO or is DB2-managed: 340 µs in a distributed environment.

In our measurements, we assume the stored procedure is defined as STAY RESIDENT YES to avoid stored procedure reloading (default= NO).

The range of numbers in our measurements depends on the following factors:

- The number and size of the input and output parameters
- The language used
- Procedure defined with PROGRAM TYPE SUB instead of MAIN to reduce stored procedure invocation overhead

An example of stored procedure vs. distributed access CPU time estimation

Accessing DB2 for z/OS data through a stored procedure can reduce CPU usage and response time in a distributed environment in cases where there are multiple SQL statements that would cause multiple send/receive pairs.

For example, if a transaction initiated on a distributed client is to be modified to include a mix of 10 DML statements (SELECT, INSERT, UPDATE or DELETE), and cannot use block fetch because there is no cursor processing, the comparison of additional CPU time to code the statements as a stored procedure versus the additional CPU time to code distributed access logic in the client is as follows:

- Additional CPU time without stored procedure

  = 10 calls* 210 µs (10 SQL statements that are not blocked)
  = 2100 µs
- Additional CPU time with stored procedure

  = Approximately 600 µs based on (220 to 560 µs base) + (1 to 2) x 170 µs.
  Likely to require 2x170, since no COMMIT ON RETURN is the default.

Note that in addition to a reduction in CPU time, the stored procedure will also experience faster response times because there is only one send/receive pair versus ten for the distributed access case.

Analysis of CPU estimate for distributed access versus stored procedure

In the above example we see that adding the ten SQL statements increases the CPU time for the distributed access case by 2100 µs, while calling a stored procedure to execute the same ten SQL statements increases the CPU time by approximately 600 µs. As the number of SQL statements increases, the CPU time comparison favors stored procedures. You can use the above estimates for your V7 system and make a determination whether you can benefit from using a stored procedure. Remember that there are other benefits to stored procedures, such as security and code reuse, which should also impact your decision process.

Comparing the performance of various languages

You have many alternatives for language when it comes to coding stored procedures. In this book we describe coding stored procedures in COBOL, C, REXX, Java and both external and native SQL Language procedures. When determining which language to use you have to take into account the following considerations: language skills of your developers, available software, available development tools and available computer resources to run the procedures.

There have been numerous performance studies done for stored procedures, but in many cases the cost comparison for some of the languages is so close that it is difficult, if not impossible, to definitively state that one language performs better than another. We can conclude from these studies that COBOL and C stored procedures generally tend to be the best performers, while interpreted languages such as Java and REXX tend to be the slowest performers. Stored procedures written using external and native SQL Language fall somewhere in between and can vary based on your environment.

External SQL procedures can incur the overhead of extra SELECT statements against the SYSDUMMY1 table for some SQL statements because some SQL control statements are similar to C language statements and require DB2 to perform an extra translation. This overhead is less with native SQL procedures because all of the runtime structures are generated as a control section which contains the logic part of the program; there is no need for DB2 to perform an extra translation that requires access to the SYSDUMMY1 table because there is no need to generate C language statements. In some cases both external SQL language and native SQL language procedures need to access the SYSDUMMY1 table and in some cases only external SQL procedures have to access SYSDUMMY1. In no case is SYSDUMMY1 access required for native SQL procedures and not for external SQL procedures.

Native SQL procedures have the additional advantage of running as an enclave SRB and remote native SQL procedures are eligible to be run on a zIIP processor. This should not be overlooked when doing capacity planning and when considering the total software cost of a solution.

19.2 Monitoring and measuring stored procedure performance

Normal MVS console type information can be helpful in monitoring stored procedure environments. For instance, the JES joblog or syslog output will show the JCL for the WLM address space in the held output queue, with the details of the stopped and started address spaces. This gives details on the started address spaces. Also, if a WLM Application environment is in *stopped* state, the JCL in the held output queue can be a good starting point to investigate. WLM AE can go into stopped state if there are JCL errors or excessive abends in the application.

When a WLM application environment goes into stopped state, it sends the message IWM032I to the console. This is not a highlighted message; some proactive method should be

in place to track this message, and take an action to bring the WLM application environment up again.

In this section we briefly describe the most common functions and tools that you can use to monitor and measure a stored procedure's performance. It is important to be current on maintenance. See *Appendix A: Summary of performance maintenance for* DB2 V9 in *DB2 9 for z/OS Performance Topics*, SG24-7473 for a list of performance related APARs.

In the next sections we discuss the following topics:

- The DISPLAY PROCEDURE command
- Reporting on DB2 accounting class 7 and 8 data
- Reporting on DB2 statistics data
- RMF
- Overview of performance knobs

19.2.1 DISPLAY PROCEDURE command

The DB2 command -DISPLAY PROCEDURE shows statistics for one or more stored procedures. You can see which procedures are stopped and which are active. The report from the command shows counts for thread activity for each procedure, and it shows the WLM environment in which each procedure is run. To display information about the stored procedures in our test cases we issued the following command on our DB2 9 for z/OS system:

```
-DIS PROCEDURE (DEVL7083.EMPD*)
```

The report produced from this command is shown in Figure 19-5.

```
DSNX940I  -DB9A DSNX9DIS DISPLAY PROCEDURE REPORT FOLLOWS -
------- SCHEMA=DEVL7083
PROCEDURE      STATUS ACTIVE QUED MAXQ TIMEOUT FAIL WLM_ENV
EMPDTL1P
               STARTED     0    0    1       0    0 DB9AWLM
EMPDTL2C
               STOPQUE     0    1    1       0    0 DB9AWLM
EMPDTL2P
               STARTED     0    0    1       0    0 DB9AWLM
EMPDTL3C
               STOPREJ     0    0    1       0    0 DB9AWLM
EMPDTL4C
               STARTED     0    0    1       0    0 DB9AWLM
DSNX9DIS DISPLAY PROCEDURE REPORT COMPLETE
DSN9022I  -DB9A DSNX9COM '-DISPLAY PROC' NORMAL COMPLETION
***
```

Figure 19-5 Output of -DISPLAY PROCEDURE command

You can see from the report that all the procedures are started except EMPDTL2C and EMPDTL3C. EMPDTL3C is in STOPREJ status, which indicates that a -STOP PROCEDURE command was issued with the ACTION(REJECT) option. Any subsequent requests for the procedure are rejected. EMPDTL2C is in STOPQUE status, which indicates that a -STOP PROCEDURE command was issued with the ACTION(QUEUE) option. Any subsequent requests for the procedure are queued. You can see in the QUED column that there is one stored procedure request queued for EMPDTL2C. If there were any procedures in STOPABN status, which is also displayed in the STATUS column, that would indicate that the procedure experienced an abend and the maximum number of abends as defined by zparm

STORMXAB had been reached, or the value in STOP AFTER n FAILURES had been reached. If you see many procedures with a STOPABN status, you may want to increase STORMXAB or adjust the allowable number of failures on an individual procedure level to minimize the need to start your procedure each time you experience an abend. This can be especially helpful in a test environment for stored procedures that abend frequently during testing. See Chapter 8, “Operational issues” on page 83 for more information about restarting stored procedures and refreshing stored procedure address spaces to resolve errors.

For each stored procedure, you also see the number of active and queued threads; the maximum number of queued threads waiting concurrently since DB2 was last started; the number of times an SQL CALL statement timed out waiting for the procedure to be scheduled; and the number of times the procedure failed. All of these values are reset when you issue a START PROCEDURE command. You can use this information to monitor the behavior of each application environment and adjust the number of TCBs, or move applications to different environments as needed.

Note that native SQL procedures will not show up in the output of a -DISPLAY PROCEDURE command unless you run the procedures in DEBUG mode. If you do run the procedure in DEBUG mode, the WLM environment column in the output contains the “ WLM ENVIRONMENT FOR DEBUG MODE name” that you specified when you created the native SQL procedure. For more details on native SQL procedures, see Chapter 15, “Native SQL procedures” on page 253.

19.2.2 Reporting on DB2 accounting class 7 and 8 data

DB2 for z/OS includes an instrumentation facility component (IFC) that collects performance data at both system and application levels. In order to collect this data you have to turn on specific traces by issuing a -START TRACE command to collect data for certain trace classes. To monitor stored procedures you generally need to start an accounting trace and collect data for classes 1, 2, 3, 7, and 8. Both elapsed time and CPU time will be collected for each class. You can also sporadically activate CLASS(10), introduced by V8 APAR PK28561 (UK18090) when you need to collect package details.

See Table 19-1 for a description of the accounting classes.

Table 19-1 Description of accounting classes

Accounting class	Description of data collected
1	CPU time and elapsed time in application, at plan level
2	CPU time and elapsed time in DB2, at plan level
3	Elapsed time due to suspensions, at plan level
7	CPU time and elapsed time in DB2, at package level
8	Elapsed time due to suspensions, at package level
10	Detailed information about locks, buffers and SQL statistics at the package level, additionally collected in IFCID 239. See also informational APAR II14421.

For more details on starting traces and the appropriate trace classes to choose, see *IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS Version 4.1.0, Reporting User’s Guide*, SC18-9983-01. Another source of information on this tool is *IBM DB2 Performance Expert for z/OS Version 2*, SG24-6867-01. Example 19-1 shows a sample -START TRACE command that you can use to start an accounting trace for monitoring stored procedures.

Example 19-1 START TRACE command to monitor stored procedures

```
-START TRACE(ACCTG) CLASS(1,2,3,7,8)
```

If you have either the IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS or IBM Tivoli OMEGAMON XE for DB2 Performance Monitor on z/OS tool installed, you can start your trace using the DB2 PM Workstation Monitor or DB2 PM Online Monitor. For our test cases we used DB2 PM batch and the Online Monitor for our monitoring. If you have other performance monitoring tools, you will need to review the documentation for those tools to determine how to best monitor stored procedure performance.

DB2 PM provides monitoring capabilities for stored procedures in both batch and online mode. Batch monitoring reports on activity for a given time period. Online monitoring reports on stored procedure activity for active threads.

Batch monitoring

Since stored procedures run under the plan of the caller, you can see stored procedure activity in the accounting report for the appropriate plan, which will often be the plan for your distributed application. In our test cases, the Java stored procedures were run under a plan name of *javaw.ex*. Example 19-2 shows a stored procedures section from a DB2 PM *Accounting Long Report,* which lists information about the stored procedure activity for that *plan*.

Example 19-2 Stored procedures trace block of DB2 PM Accounting Long Report

```
PRIMAUTH: PAOLOR5  PLANNAME: javaw.ex

STORED PROCEDURES   AVERAGE     TOTAL
-----------------  --------  --------
CALL STATEMENTS         2.00         8
ABENDED                 0.00         0
TIMED OUT               0.00         0
REJECTED                0.00         0
```

The data on the report is interpreted as follows:

- **CALL STATEMENTS**: The number of SQL CALL statements executed by the plan
- **ABENDED**: The number of times a stored procedure terminated abnormally
- **TIMED OUT**: The number of times an SQL CALL statement timed out waiting to be scheduled
- **REJECTED**: The number of times an SQL CALL statement was rejected because the procedure was in the STOP ACTION(REJECT) state

The AVERAGE column represents the average number of occurrences per thread during the monitoring duration, while the TOTAL column represents the total number of occurrences for all threads during the monitoring duration.

To see accounting information for the stored procedures themselves, you need to look at the data for the stored procedure *package*. You can do a search for your *package* name within the accounting report. Information for each package executed during the reporting period is displayed within the section of the report for the associated plan name. In our test case one of the stored procedures executed is EMPRSETC, which was executed under plan javaw.ex, which is a Java plan initiated from a distributed platform. Example 19-3 shows a package identification section from a *DB2 PM Accounting Long Report,* which lists information about the activity within stored procedure EMPRSETC.

Example 19-3 Package identification trace block of DB2 PM Accounting Long Report

```
PRIMAUTH: PAOLOR5  PLANNAME: javaw.ex

EMPRSETC            VALUE
------------------  ------------------
TYPE                PACKAGE

LOCATION            DB2G
COLLECTION ID       DEVL7083
PROGRAM NAME        EMPRSETC

OCCURRENCES                          1
SQL STMT - AVERAGE                2.00
SQL STMT - TOTAL                     2
STOR PROC EXECUTED                   0
UDF EXECUTED                         0
USED BY STOR PROC                    1
USED BY UDF                          0
USED BY TRIGGER                      0
SUCC AUTH CHECK                      0
```

The pertinent counters for stored procedures in this section of the report are:

- **STOR PROC EXECUTED**: The number of stored procedures scheduled by this package
- **USED BY STOR PROC**: The number of times this package was invoked by a stored procedure

The Accounting Long Report also provides information that can help you tune your WLM address spaces. The reporting of elapsed time includes various counters for SUSPEND TIME, including SUSPEND TIME STORED PROC. This counter reports on the total elapsed time waiting for an available TCB before a stored procedure could be scheduled. Since the elapsed time counters show average times, you would expect this field to be non zero for stored procedures, because it is likely that at some point WLM would either need to start a new address space or need to add more TCBs to an existing address space. However, you would like this counter to be as small as possible.

The Accounting Long Report extract in Example 19-4 shows that the stored procedure suspend time accounts for about 38% of the total stored procedure elapsed time. This is an indication that the stored procedure request had a fairly long wait for a TCB to be scheduled.

Example 19-4 Accounting Long Report showing stored procedure suspend time

```
AVERAGE       APPL(CL.1)  DB2 (CL.2)
------------  ----------  ----------
ELAPSED TIME    0.463223    0.001396
 NONNESTED      0.040053    0.000691
 STORED PROC    0.423170    0.000704
 UDF            0.000000    0.000000
 TRIGGER        0.000000    0.000000

CP CPU TIME     0.009524    0.001300
 AGENT          0.009524    0.001300
  NONNESTED     0.005705    0.000675
  STORED PRC    0.003819    0.000625
  UDF           0.000000    0.000000
  TRIGGER       0.000000    0.000000
 PAR.TASKS      0.000000    0.000000

 IIPCP CPU      0.000000         N/A
```

```
IIP CPU TIME    0.000000    0.000000
 STORED PROC    0.000000    0.000000

SUSPEND TIME    0.177809    0.000000
 AGENT               N/A    0.000000
 PAR.TASKS           N/A    0.000000
 STORED PROC    0.177809         N/A
 UDF            0.000000         N/A
```

In this test case we were running a very small number of stored procedures, so the wait time is likely to be overstated when compared to a production environment where many stored procedures are running in the same address space and can use available TCBs. WLM's Resource Adjustment function, described in 20.1.4, "WLM management of server address spaces" on page 427 explains how WLM determines when to start new address spaces and when it is better to wait for an available TCB in an already started address space.

See *DB2 Performance Monitor for z/OS Version 7.2, Report Reference*, SC27-1647 for more details on the layout of the Accounting Long Report.

Online monitoring

You can use DB2 Performance Expert or DB2 Performance Monitor to report on the activity of currently executing threads. You can use these tools to monitor stored procedures that you know are causing you problems. We used DB2 Performance Expert for z/OS Version 1 to monitor stored procedures in our test cases. Follow these steps to view stored procedure activity for the thread being monitored:

1. From the DB2 PE main menu, select option **3**. **View online DB2 activity.**
2. From the Online Monitor Main Menu, select option **1. Display Thread Activity**.
3. The Thread Summary panel is displayed, as shown in Figure 19-6.

```
03/12/11 17:22              Thread Summary                  ROW 1 TO 9 OF 9
Command ===> ____________________________________________________________

DB2G              DB2G V7

To display a thread, place any character next to it, then press Enter.

                      Program      Connection        ------- Elapsed -------
  Primauth   Planname   name         ID         Status     Class 1      Class 2
_ STC        FPEPLAN    DGO@SDOB     DB2CALL    APPL    19:13:56.5     9.936754
_ STC        FPEPLAN    DGO@DB2I     DB2CALL    DB2     19:13:57.2     1.682438
_ STC                   N/P          IMSG       I/S            N/P          N/P
_ STC                   N/P          DB2CALL    APPL    19:13:56.8     0.008493
_ STC                   N/P          DB2CALL    APPL    19:13:53.0     0.155218
_ PAOLOR5    FPEPLAN    N/P          DB2CALL    APPL    2:11:22.20           N/P
_ PAOLOR5    DISTSERV   SYSSTAT      SERVER    *APPL    6:08.97670     0.001386
s PAOLOR5    DISTSERV   DSNJDBC2     SERVER    *APPL    4:32.25501     0.001407
_ NONE       DISTSERV   N/P          DISCONN   *DB2            N/P          N/P
-- End of Thread list --
```

Figure 19-6 Thread Summary panel of DB2 PE

4. Select the thread that you wish to monitor and press Enter. The Thread Detail panel is displayed, as shown in Figure 19-7. Elapsed and CPU times are displayed.

```
03/12/11 17:23                  Thread Detail                DB2G Top of data
Command ===> ____________________________________________________________________

For details, place any character next to heading, then press Enter.
                                                                        More:     +
_ Thread Identification
  Primauth . . . . . : PAOLOR5          Correlation Name  . . . : javaw.ex
  Planname . . . . . : DISTSERV         Connection type . . . . : DRDA
  Connection ID  . . : SERVER           Type  . . . . . . . . . : DBAT
  Requesting Location: 9.1.39.26        Status  . . . . . . . . : APPL
_ Current Package     . . . . . . . . . . . : DSNJDBC2
s Times                                            Elapsed          CPU
  Class 1  . . . . . . . . . . . . . . . . . :   4:54.216568     0.002834
  Class 2  . . . . . . . . . . . . . . . . . :      0.001407     0.002926
  Class 3  . . . . . . . . . . . . . . . . . :     26.492065          N/A
  Class 7  . . . . . . . . . . . . . . . . . :      0.000465     0.000278
  Class 8  . . . . . . . . . . . . . . . . . :           N/P          N/A
_ Locking Activity
  Timeouts . . . . . . . . . . . . . . . . . :             0
  Deadlocks  . . . . . . . . . . . . . . . . :             0
  Suspensions  . . . . . . . . . . . . . . . :             0
  Lock escalations . . . . . . . . . . . . . :             0
  Maximum page locks held  . . . . . . . . . :             0
```

Figure 19-7 Thread Detail panel of DB2 PE

- Select the **Times** option to see details on elapsed and CPU times for the thread. The Thread Times panel is displayed, as shown in Figure 19-8 and Figure 19-9.

```
0 ____________________________________________________________________
  |                          Thread Times                              |
  | Command ===> ____________________________________________________ |
  |                                                                   |
  |                                                    More:     +    |
  |                               Class 1       Class 2               |
  |                              In Appl        In DB2    Outside DB2 |
  | Elapsed time . . . . . :  4:54.216568      0.001407   4:54.215161 |
  | CPU time . . . . . . . :     0.002834      0.002926           N/C |
  |   TCB  . . . . . . . . :     0.001022      0.001199           N/C |
  |   TCB - Stored Proc  . :     0.001811      0.001727               |
  |   Parallel tasks . . . :     0.000000      0.000000               |
  | Waiting time . . . . . :          N/A      0.000208               |
  | Suspension time  . . . :          N/A     26.492065               |
  |   TCB  . . . . . . . . :          N/A     26.492065               |
  |   Parallel tasks . . . :          N/A      0.000000               |
  | Not accounted  . . . . :          N/A           N/C               |
  |                                              Time          Event |
  | Suspensions (Class 3) . . . . . . . . :    26.492065             2 |
  |   Locks and latches . . . . . . . . . :     0.000000             0 |
  |   Synchronous I/O . . . . . . . . . . :     0.000000             0 |
  |   Other read I/O  . . . . . . . . . . :     0.000000             0 |
  |   Other write I/O . . . . . . . . . . :     0.000000             0 |
```

Figure 19-8 Thread Times panel of DB2 PE (page 1 of 2)

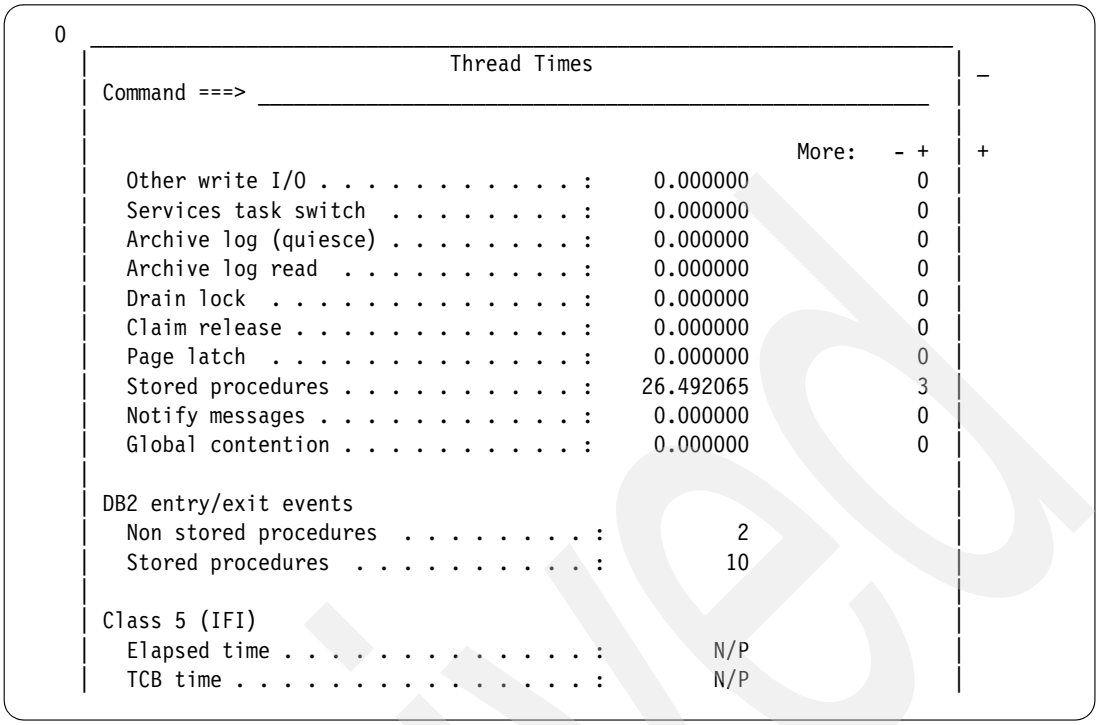

Figure 19-9 Thread Times panel of DB2 PE (page 2 of 2)

In the Thread Times panels you can see details for class 1, class 2, and class 3 times. Note that the total class 3 (suspensions) time is 26.49 seconds, and that the stored procedure suspension time accounts for all of that total. This means that there was a wait time of 26.49 seconds for an available TCB before the stored procedure could be scheduled. This is a considerably high number, which is most likely due to our NUMTCB value being set too low for our test case.

Returning to the Thread Detail panel, we can now select the option to see the SQL activity for the thread, as shown in Figure 19-10.

```
03/12/11 17:25                  Thread Detail                  DB2G DB2G V7
Command ===> ____________________________________________________________

For details, place any character next to heading, then press Enter.
                                                                  More:   - +
_ Locked Resources
_ RID List Processing
  Unsuccessful - any reason  . . . . . :            0
s SQL Activity, Commits and Rollbacks
  DML . . . :         14 Commit  . . . . . :            0
  DCL . . . :          2 Rollback  . . . . . :          0
  DDL . . . :          0 Changes/Commit  . . :        2.0
_ Buffer Manager Activity
  Getpage requests . . . . . :           34
  Buffer updates . . . . . :              7
  Prefetch requests  . . . . . :          2
  Synchronous I/O  . . . . . :            0
_ SQL Statement and Package  . . . . . :  DSNJDBC2
_ Distributed Data
  Requester elapsed time . . . . . :      N/A
_ IFI (Class 5) and Data Capture
_ Query Parallelism Data
  Data Sharing Locking Activity
```

Figure 19-10 Selecting the SQL Activity panel of DB2 PE

The SQL Activity panel is displayed. This panel shows you counts for each type of SQL statement executed within the thread. You need to scroll forward to see all the counters. The SQL Activity panels for our test case are shown in Figure 19-11, Figure 19-12, and Figure 19-13 on page 410. The SQL Activity panels show us that there were two stored procedures called during the time that this thread was monitored (SQL call = 2 on Figure 19-13) and there were nine rows fetched from cursors (Fetch = 9 on Figure 19-12).

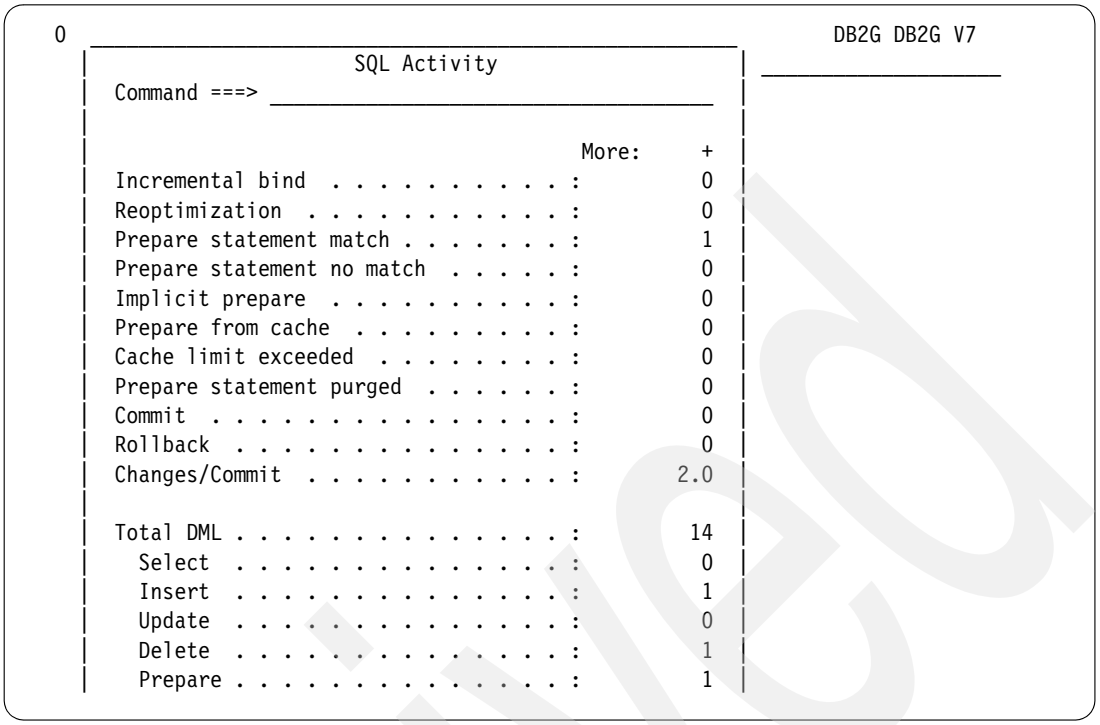

Figure 19-11 SQL Activity panel of DB2 PE (page 1 of 3)

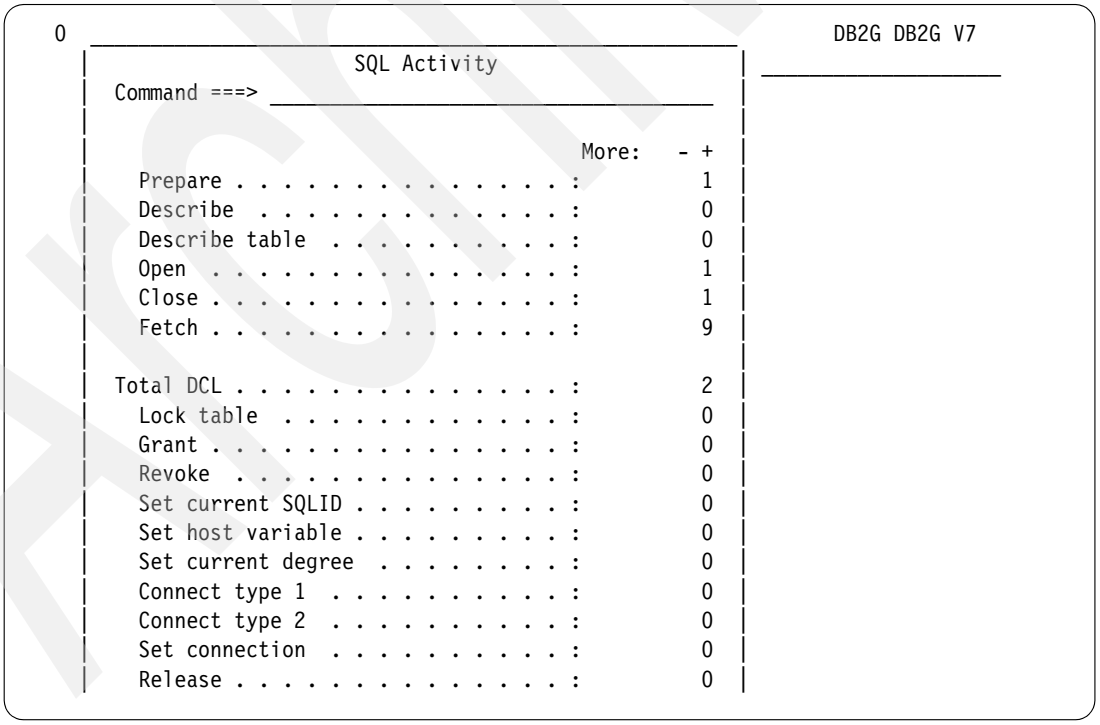

Figure 19-12 SQL Activity panel of DB2 PE (page 2 of 3)

```
O                                                          DB2G DB2G V7
                        SQL Activity
  Command ===> _____________________________________________
                                                  More:   - +
    Release . . . . . . . . . . . . . . . :          0
    Set current rules . . . . . . . . . . :          0
    SQL call  . . . . . . . . . . . . . . :          2
    Associate locators  . . . . . . . . . :          0
    Allocate cursor . . . . . . . . . . . :          0

  Total DDL . . . . . . . . . . . . . . . :          0
    Rename table  . . . . . . . . . . . . :          0
    Comment on  . . . . . . . . . . . . . :          0
    Label on  . . . . . . . . . . . . . . :          0

                       Create       Drop       Alter
    Table . . . . :         0          0           0
    Temp. Table . :         0        N/A         N/A
    Index . . . . :         0          0           0
    Tablespace  . :         0          0           0
    Database  . . :         0          0           0
    Stogroup  . . :         0          0           0
```

Figure 19-13 SQL Activity panel of DB2 PE (page 3 of 3)

As you can see there is a large quantity of performance information available for stored procedures. For more details on monitoring the performance of stored procedures during real time, see *DB2 Performance Expert for z/OS Version 1 Monitoring Performance from ISPF*, SC27-1652-02.

19.2.3 Reporting on DB2 statistics data

You can also report on system-wide statistics for usage of stored procedures. The *Statistics Long Report* includes a trace block that shows counts of stored procedures executed. Example 19-5 shows a stored procedures section from a DB2 PM Statistics Long Report that shows information about the stored procedure activity at a system-wide level. In our test case it shows that 16 stored procedure calls were executed during the statistics interval reported.

Example 19-5 Stored procedures trace block of DB2 PM Statistics Long Report

```
STORED PROCEDURES           QUANTITY  /SECOND  /THREAD  /COMMIT
--------------------------  --------  -------  -------  -------
CALL STATEMENT EXECUTED        16.00     0.02     0.64     0.62
PROCEDURE ABENDED               0.00     0.00     0.00     0.00
CALL STATEMENT TIMED OUT        0.00     0.00     0.00     0.00
CALL STATEMENT REJECTED         0.00     0.00     0.00     0.00
```

See *DB2 Performance Monitor for z/OS Version 7.2, Report Reference*, SC27-1647-02 for more details on the layout of the Statistics Long Report.

19.2.4 RMF

You can use RMF™ to monitor distributed processing, including stored procedures called from a distributed client. RMF reports on SMF type 72 records, which monitors the portions of the client's request that are covered by individual enclaves. The duration of the enclave

depends on whether the threads are active or inactive. We recommend you use type 2 inactive threads. RMF reports on the time that the thread is active. This includes any queueing time, which includes the time waiting for an existing thread or new thread to become available.

The type 72 records contain data collected by RMF monitor 1. There is one type 72 record for each service class period, report class, performance group number (PGN) period, and report performance group (RPGN) per RMF monitor 1 interval. Each enclave contributes its data to one type 72 for the service class or PGN and to zero or one (0 or 1) type 72 records for the report class or RPGN. By using WLM classification rules, you can segregate enclaves into different service classes or report classes (or PGNs or RPGNs, if using compatibility mode). By doing this, you can understand the DDF work better.

Example 19-6 shows a sample job to run an RMF monitor 1 report that shows workload by service class. The literal *DDFSP8* represents the service class we are monitoring. To monitor a different service class, change the variable. To monitor a service class period use the keyword SCPER instead of SCLASS.

Example 19-6 Sample JCL to produce RMF monitor 1 report

```
//SMFDUMPP JOB (999,POK),'COBOL C/L/B/E',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=SC63,L=9999
//STEP EXEC PGM=IFASMFDP,REGION=0M,TIME=1440
//INDD1 DD DSN=SYS1.SC63.MAN1,DISP=SHR
//OUTDD1 DD DSN=PAOLOR7.SMF.DB2SPB,DISP=SHR
//*OUTDD1 DD DSN=PAOLOR7.SMF.DB2SPB,
//*  DISP=(,CATLG),SPACE=(CYL,(30,10)),
//*  DCB=HGPARK.SMF.CASE1,UNIT=SYSDA
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
INDD(INDD1,OPTIONS(DUMP))
OUTDD(OUTDD1)
DATE(2003321,2003365)
/*
//REP20    EXEC PGM=ERBRMFPP
//STEPLIB  DD  DISP=SHR,DSN=CEE.SCEERUN
//MFPINPUT DD  DISP=SHR,DSN=PAOLOR7.SMF.DB2SPB
//MFPMSGDS DD  SYSOUT=*
//* WORKLOAD BY SERVICE CLASS
 SYSID(SC63)
 SYSRPTS(WLMGL(SCLASS(DDFSP8)))
 NOSUMMARY
```

See *z/OS V1R7.0 Resource Management Facility User's Guide,* SC33-7990-10 for details on producing RMF reports.

19.2.5 Overview of performance knobs

There are a number of parameters and controls that can be adjusted to improve the performance of stored procedures. Lab measurements have shown that client/server processing greatly benefits from this process. In this section we discuss what tuning knobs are available and how they can be used.

Block fetch

Stored procedures work best for applications with many non-blocked SQL statements. Distributed applications can take advantage of block fetch for read-only cursors. Applications

that have this type of logic may not see the performance advantages of stored procedures. Block fetching does not work for INSERT, UPDATE, or DELETE statements, so applications that change DB2 data cannot take advantage of block fetch, and will see more performance benefits when coded as stored procedures.

About block fetching, it is worth considering that prior to the addition of the FETCH FIRST clause, the OPTIMIZE FOR clause was used to control network blocking and to control access path selection. With the addition of the FETCH FIRST clause, its interaction with the OPTIMIZE FOR clause also influenced network blocking and access path selection. If both clauses are specified and the customer is using OPTIMIZE FOR for the desired blocking and access path selection, the FETCH FIRST clause can override the OPTIMIZE FOR clause if its value was less than the OPTIMIZE FOR value. DB2 V7 APAR PQ49458 has modified the behavior of the FETCH FIRST n ROWS ONLY clause as follows:

- The FETCH FIRST clause will have no impact on network blocking. If the FETCH FIRST clause is specified and the OPTIMIZE FOR clause is not specified, access path will use the FETCH FIRST value for optimization, but DRDA will not consider the value when it determines network blocking.
- When both the OPTIMIZE FOR clause and FETCH FIRST clause are specified, the OPTIMIZE FOR value will be honored even if it is greater than the FETCH FIRST value. Currently, if both clauses are specified, the lower of the two integer values is used for access path selection. However, if a customer is explicitly specifying both clauses, DB2 should use the specified values since they may have been chosen for performance reasons.

Note: With DB2 V8, DRDA internally and automatically exploits multi-row operations. This means that for remote transactions, the fetching and inserting of rows between the DDF and DBADM1 address spaces will benefit from fewer interactions.

NUMTCB

When you define each WLM application environment, you specify a NUMTCB value. NUMTCB specifies the maximum number of TCBs that can run concurrently in a WLM address space. When a request for a TCB is received, and the maximum number of TCBs for that address space is already reached, WLM needs to start another address space for that application environment. Your stored procedure has to wait for that address space to be scheduled. The wait time is shown in DB2 accounting as part of the class 1 (suspension) time (see Example 19-4 on page 404). You can adjust the NUMTCB value of each application environment to meet the performance needs for those environments. See 20.1.2, “NUMTCB” on page 425 for more details on the implications of your NUMTCB settings.

Note that installation panel DSNTIPX - ROUTINE PARAMETERS includes a field named NUMBER OF TCBS. The value of this field is used as the default NUMTCB value when address space JCL is generated.

Native SQL procedures run under an enclave SRB so they are not subject to any NUMTCB limits.

Note: Starting with DB2 for z/OS V8, Workload Manager will not automatically assign the number of TCBs that is specified in NUMTCB. Instead, WLM will consider the NUMTCB value as a maximum and will manage the number of TCBs in an address space based on resource utilization. See 20.2.4, “Recommendation - Exploit WLM server task thread management” on page 433 for more details.

CREATE PROCEDURE statement

There are a number of options on the CREATE PROCEDURE statement, which can impact the performance of your stored procedures. We discuss each of the options here.

- PROGRAM TYPE (SUB or MAIN)
  - If you specify MAIN, then your stored procedure and all its called modules run as main routines. The load modules for any programs called by your stored procedure are reloaded into memory for each execution, and then are deleted from memory at the end of each execution. Although reloading each time ensures that all work areas are initialized, there is considerable overhead incurred.
  - If you specify SUB, then your stored procedure and all its called modules run as subroutines. The load modules for any programs called by your stored procedure remain in memory after they have been loaded the first time. Specifying SUB reduces the overhead of loading modules, but it forces application developers to ensure that work areas are properly initialized.
- STAY RESIDENT (YES or NO)
  - If you specify YES, then the load module for your stored procedure remains in memory after it has been loaded the first time. This has no impact on any programs called by the stored procedure. Specifying YES reduces the overhead of loading modules, but it forces application developers to ensure that work areas are properly initialized.
  - If you specify NO, then the load module for your stored procedure is loaded into memory each time it is called, and then deleted from memory at the end of each execution, unless there are other tasks that are accessing the stored procedure. This has no impact on any programs called by the stored procedure. Although reloading each time ensures that all work areas are initialized, there is considerable overhead incurred.
- COMMIT ON RETURN (YES or NO)
  - If you specify YES, then DB2 issues a commit when the stored procedure returns to the calling program. This commits the work of the stored procedure, and the work of the calling application. This is useful for distributed applications because it releases the locks held by the client. However, if the stored procedure is called via a connection established via DB2 Connect in a sysplex environment where sysplex workload balancing is enabled, the client does not see the commit and does not know to reuse the connection. This can affect how work is distributed among members of a sysplex. If the connection is established via a Java Type 4 connection, then the client will recognize the commit and the connection can be reused.
  - If you specify NO, then DB2 does not issue a COMMIT when the stored procedure returns to the calling program. It is the calling program's responsibility to issue a commit or a rollback.
  - If you code a COMMIT inside the stored procedure, regardless of the COMMIT ON RETURN setting, the thread is not eligible to become an INACTIVE thread.
- PARAMETER STYLE (GENERAL, GENERAL WITH NULLS, SQL)
  - If you specify GENERAL, then only the parameters on the call statement are passed to the stored procedure. You have the capability to set output parameters to null by including null indicators for those parameters, but you cannot set input parameters to null. Therefore all input parameters are passed to the stored procedure.
  - If you specify GENERAL WITH NULLS or SQL, then you can set both input and output parameters to null and reduce the amount of information that is passed to and from the stored procedure. Prior to DB2 for z/OS V8 the value of DB2SQL was used instead of SQL. In DB2 V8 and V9 the value of DB2SQL is still accepted and acts as a synonym for PARAMETER STYLE SQL.

WLM address space priority

When you define your WLM application environments, you need to be aware that the WLM address space needs a *base* priority high in order to get the stored procedure set up and started. The stored procedure that executes in that environment will then run in the WLM service class of the applications that call those procedures. For example, stored procedures that are called by a transaction running on a workstation platform will execute under the classification rules for your DDF workload. Note that you also want to make sure that your WLM address space priorities are not lower than the highest priority of any thread that calls a stored procedure in those address spaces; otherwise the high priority thread may have to wait for the low priority stored procedure.

If you do not provide any classification rules for your DDF workload, then default service classes apply. Stored procedures will default to the SYSOTHER service class, which has a *discretionary goal*. This means that when the system is near capacity, your DDF work will not get the resources it needs to complete. You can prevent work from falling into a service class with a discretionary goal by making sure that all combinations of workloads are covered by the classification rules. For example, you can define one service class at the subsystem level, then another for a list of packages that start with some common characters. See Chapter 36, “Assigning procedures and functions to WLM application environments” in the *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413 for details on setting address space priorities.

Language Environment runtime library access and options

Language Environment (LE) is a component of the z/OS operating systems. LE establishes a common runtime environment for stored procedures that may be written in many different high level languages. The runtime library access needs to be facilitated for performance, and the runtime options need to be chosen for speed and efficiency.

For the library access, the LE libraries should be placed in LLA with the FREEZE option, either if allocated through LNKLST or STEPLIB. Further improvements can be obtained by placing in LPA the eligible portion of SCEERUN as listed in LPALST. For details, see the *z/OS 1.4 Language Environment Customization*, SA22-7564-4.

There is some overhead in establishing this environment, especially in regard to the amount of storage required by LE when running many stored procedures concurrently. You can minimize the storage required below the 16 MB line for LE by specifying some runtime options when you create your stored procedures. See 19.3, “Recommendations” on page 418 for more details.

Grouping of stored procedures within application environments

When you define your stored procedures, you specify an application environment in which they will run. How you group your stored procedures in application environments can have an impact on performance.

Important: If you group stored procedures of different language types, such as COBOL and C, in the same environment, each language may have different LE runtime options. When stored procedures that are defined with PROGRAM TYPE SUB with one set of LE runtime options execute in an application environment, and are followed by a stored procedure with a different set of LE runtime options in the same environment, all the stored procedures with the initial set of LE runtime options are invalidated, and must be reloaded at the next execution. When the stored procedures with the original runtime options are subsequently executed, the stored procedure that had the different options is now invalidated, and must be reloaded upon next execution. The effect is that the LE environment is refreshed, making it behave more like PROGRAM TYPE MAIN, and even worse, because all modules loaded into that environment are deleted. So, you can see that mixing stored procedures with different LE options is not a good idea.

Note that you can run into this same issue with stored procedures written in the same language with the same default LE runtime options if you specifically set runtime options for a stored procedure. For example, you can use the TEST runtime option for debugging purposes. Some customers specify an IP address in the TEST option during development, but neglect to remove that runtime option when deploying the stored procedure in production. As a result, there are multiple stored procedures running in the same application environment that have different runtime options and the LE environment is refreshed each time a stored procedure is loaded that has different runtime options than the last stored procedure that ran in that application environment. This can be very costly in terms of performance. For more details on using the TEST runtime option, see 5.2.3, “TEST and NOTEST” on page 49.

You want to separate your Java stored procedures from your other language stored procedures. Since Java stored procedures load a JVM into the application environment for each TCB, and the JVM can be quite large, you can realistically run with a NUMTCB value of no more than 8 for an application environment that runs Java code. If you mix stored procedures written in other languages with your Java stored procedures, you are limiting the number of procedures you can run for the other language. Furthermore, the whole environment is torn down between Java and non-Java, and then the JVM is recreated on the next stored procedure call whether or not a Java stored procedure is invoked there next. If the WLM environment is set up for Java, DB2 makes sure to have a JVM ready.

There are also some considerations for nested stored procedures. If you have a stored procedure running in one address space that calls another stored procedure that runs in the same address space, you can experience a situation where the nested stored procedure is waiting on a TCB, and the stored procedure that called the nested procedure is waiting for the nested procedure to complete. If you are using nested stored procedures, you should consider placing them in a separate application environment from the stored procedures that call them.

DSNTRACE facility

DSNTRACE is a facility that can be used to capture all trace messages for offline reference and diagnosis. We recommend that you do not use the DSNTRACE DD statement (not even a DUMMY DD name) in any of your stored procedures address space startup procedures, because DSNTRACE greatly increases the stored procedure initialization overhead. Also, DSNTRACE does not function in a multitasking environment because the CAF does not serialize access to the DSNTRACE trace data set.

Other debug options

You can code logic in your stored procedures to aid with debugging. For example you can use println() in Java stored procedures and DISPLAY in COBOL stored procedures. We recommend that you do not keep these debugging statements in your code when you deploy your stored procedures in production because of the added execution time as well as the volume of messages that would be written to the address space.

Authorization caching

DB2 system parameter CACHERAC controls the size of the routine authorization cache, which stores authorization IDs for stored procedures once DB2 has retrieved them from the DB2 catalog and validated them. Specifying a routine authorization cache that is too small can cause entries in the cache to be overwritten and require DB2 to re-read those entries from the DB2 catalog.

IBM Tivoli OMEGAMON XE for DB2 Performance Monitor on z/OS (DB2 PM) reports on the effectiveness of the routine authorization cache in the Authorization Management section of the Statistics Report. Example 19-7 shows an excerpt from the Authorization Management section.

Example 19-7 Authorization Management section of a DB2 PM Statistics Report

```
AUTHORIZATION MANAGEMENT     QUANTITY  /SECOND  /THREAD
--------------------------  --------  -------  -------
RTN-AUTH SUCC-W/O CATALOG      100.2K     3.84     1.09
RTN-AUTH SUCC-PUB-W/O CAT        0.00     0.00     0.00
RTN-AUTH UNSUCC-CACHE          112.3K     4.30     1.23
RTN CACHE OVERWRT - AUTH ID    111.9K     4.28     1.22
RTN CACHE OVERWRT - ENTRY        0.00     0.00     0.00
RTN CACHE - ENTRY NOT ADDED      0.00     0.00     0.00
```

In this sample report you can see that more than half of the time DB2 could not find the needed authorization in the cache, as shown by the value in RTN-AUTH UNSUCC-CACHE. In those instances DB2 had to read the authorization from the DB2 catalog. You can expect a high number of unsuccessful cache reads if every stored procedure request came in with a different authorization ID than the previous one, but that is usually not the case. The counter RTN CACHE OVERWRT - AUTH ID represents how many times an authorization ID in the cache was overwritten due to there not being enough space to store additional authorization IDs.

If you are using the default value for CACHERAC, which is 32 KB in DB2 V7 and 100 KB in DB2 V8, and you see a non-zero value in RTN CACHE OVERWRT - AUTH ID, then you need to increase the size of your cache. The default value of 32 KB in V7 is enough to hold about 370 routines. The default value of 100 KB in V8 and V9 is enough to hold about 690 routines. Note that the size of many objects in DB2 changed from DB2 V7 to V8, so if your routines fit nicely in the authorization cache in V7 it does not necessarily mean they will fit nicely in the V8 cache if you maintain a cache size of 32 KB. Note also that a maximum of five authorization IDs can be cached for each routine, so if you don't use secondary authids you won't get as much benefit from the cache. See 7.5.8, "Authorization caching" on page 81 for more details on the process DB2 follows to search for authorization IDs.

db2cli.ini file settings

If you call a stored procedure from a Java client running on DB2 for Linux, UNIX and Windows, and the Java driver on the client is the legacy JDBC driver, then the Java client will call an additional metadata procedure to perform a DESCRIBE statement for each output parameter passed in the stored procedure call. This routine will show up in your DB2

Accounting data as package SQLPROCEDURECOLS. If you are using the IBM Data Server Driver for JDBC and SQLJ, also known as the IBM Universal Driver, on your client platform, then you will not see the calls to this routine. The IBM Universal Driver assumes the maximum size for output parameters and never tries to call any metadata procedures.

If your client platform is still using the legacy JDBC driver (not supported with DB2 V9), then there is a db2cli.ini option provided in DB2 for LUW Version 8, fixpak 12 called "DescribeCall" which, when set to 0, will block the DESCRIBE from happening on CALL statements.

There is also an issue with the legacy JDBC driver when there is a data type mismatch between the parameters passed and the DB2 column definitions on SET statements. This data type mismatch will also result in a call to SQLPROCEDURECOLS to DESCRIBE the parameters. There is a db2cli.ini parameter called “DescribeParam” which, when set to 0, will block the DESCRIBE from happening.

You can actually prevent calls to the metadata procedure SQLPROCEDURECOLS for both of the above situations by setting either of the two above parameters to zero. What happens is that setting either parameter to zero will tell CLI that SQLDescribeParam is not supported, and the DESCRIBE will not happen for any function triggering DESCRIBE information, regardless of the type of SQL statement.

There is another db2cli.ini setting that you can use to minimize the impact of the DESCRIBE of the stored procedure parameters in a client environment that uses the legacy JDBC driver. Parameter SPCache, when set to 1, will cache DESCRIBE information returned from metadata procedure SQLPROCEDURECOLS. If a subsequent call to the same stored procedure occurs on the same connection, then the DESCRIBE does not need to happen. This parameter is disabled by default.

If you reference LOBs or distinct types in your stored procedure, you need to perform the metadata calls to SQLPROCEDURECOLS or your application will receive an error. This is the case where SPCache=1 should be used to minimize the overhead.

We recommend that you migrate your DB2 for LUW client environment to use the IBM Universal Driver so you do not have to worry about adjusting db2cli.ini file settings.

Non-resettable JVMs

DB2 for z/OS Version 8 APAR PK09213 provides new function that allows Java applications to run in non-resettable mode. Prior to this APAR all Java stored procedures ran in a Java Virtual Machine (JVM) in resettable mode, which limited what Java routines could do in order that the JVM did not become corrupted. Running in resettable mode also limited how many stored procedures you could run in an address space because of memory constraints.

After applying this APAR, which requires the IBM Universal Driver, you can run Java stored procedures in non-resettable mode by specifying a value of less than zero for Java environment variable RESET_FREQ. This variable specifies the number of routine invocations between JVM resets. Setting RESET_FREQ to a negative value in effect prevents the JVM from being reset. This lets you specify a larger amount of memory for Java stored procedures, therefore allowing you to run more stored procedures in an address space. Where you previously could only run 5 to 7 TCBs in an address space, you can now run 20 to 40 TCBs, although this depends on the size of your Java stored procedures and the settings you use for the non-system heap (-Xmx) and for the middleware heap (-Xms). For more details on making use of this feature, see the APAR text for APAR PK09213.

DB2 system parameter CMTSTAT

System parameter CMTSTAT controls whether DDF threads run as inactive or active database access threads (DBATs). This parameter specifies whether to make a thread active

or inactive after it successfully commits or rolls back and holds no cursors. The possible values are ACTIVE or INACTIVE. If you specify ACTIVE then the thread remains active, which provides the best performance, but consumes the most system resources and therefore may have a negative impact on performance if you run a large number of concurrent DBATs; in that case you should specify INACTIVE. Most customers should be specifying INACTIVE.

DB2 9 for z/OS supports two types of inactive threads: an inactive DBAT and an inactive connection. An inactive connection allows the thread to be pooled and reused for other connections. Therefore a smaller number of threads can be used to service a larger number of connections. A CMTSTAT value of INACTIVE is recommended whether you use stored procedures or any other processing that requires a database access thread. Note that prior to DB2 for z/OS V8 the default for this system parameter was ACTIVE. For DB2 V8 and beyond the default is INACTIVE.

19.3 Recommendations

Here, we group some recommendations for parameter definition of the following:

- For the CREATE PROCEDURE statement
- For the Language Environment
- For nested stored procedures
- Handling result sets from DB2-supplied stored procedures

19.3.1 For the CREATE PROCEDURE statement

The CREATE PROCEDURE statement uses the following recommended options. See 9.1, “CREATE or ALTER PROCEDURE parameters” on page 92 for more details on each option.

- PROGRAM TYPE SUB

  We recommend that you specify SUB to avoid the overhead of loading subroutines into memory every time they are called, and deleting subroutines from memory every time they complete execution. Subroutines include not only the programs that are called by your stored procedures, but also any other modules that are included at linkedit time. You should avoid MAIN unless you have applications that do not effectively initialize work areas, and you have no control over your source code. This may be true of vendor packages for which changing the source code is not an option. In all other cases you should use SUB.

- STAY RESIDENT YES

  We recommend that you specify YES to avoid the overhead of loading the load module for the stored procedure into memory each time it is called, and deleting the load module from memory every time it completes execution. Be aware that even if you specify NO, it is possible that your stored procedure load module will remain in memory if there are many tasks calling that procedure.

- COMMIT ON RETURN (YES or NO)

  We recommend that you specify YES for stored procedures that are called from a distributed client application. Specifying YES will ensure that locks are released when the stored procedure returns to the calling application. We also recommend that you commit within the client after returning from the stored procedure call. This will allow the connection to be reused and will allow for better workload distribution in a sysplex environment in which sysplex workload balancing is enabled and the connection is established via DB2 Connect rather than via the Type 4 Java driver. See “COMMIT ON RETURN (YES or NO)” on page 413 for more details.

We recommend that you specify NO for stored procedures that are called locally.

We recommend that you do not specifically code a COMMIT statement in a stored procedure if it is called from a distributed client application because the thread will not be eligible to become an INACTIVE thread.

- PARAMETER STYLE (GENERAL WITH NULLS or SQL)

  We recommend that you specify either GENERAL WITH NULLS or SQL. Either of these options will give you the capability to set IN, OUT, and INOUT parameters to null in your calling programs and stored procedures by setting the associated indicator value to a negative value. Nullifying parameters that are not used during either a call to or a return from a stored procedure reduces the amount of data that is passed. For example, output parameters do not have data in them during the call to the stored procedure, so you can nullify the output parameters in the calling program. For stored procedures that are called by distributed applications this can result in a savings in the amount of data transferred across the network, thus a reduction in network transmission time. Prior to DB2 for z/OS V8 the value of DB2SQL was used instead of SQL. In DB2 V8 and V9 the value of DB2SQL is still accepted and acts as a synonym for PARAMETER STYLE SQL.

- Use WLM-managed stored procedures

  Although you will be able to maintain existing DB2-managed stored procedures in DB2 for z/OS Version 8, you will not be able to create any new DB2-managed stored procedures. In DB2 9 for z/OS you must convert all of your DB2-managed stored procedures to WLM-managed stored procedures as DB2 9 does not support DB2-managed stored procedures at all. WLM-managed stored procedures provide much more flexibility with regards to setting priorities on stored procedure workloads, and segregating workloads to lessen the impact that one inefficient stored procedure can have on the rest of the stored procedures in your environment. You need to make sure that your WLM classification rules have been defined for your stored procedure workloads to prevent them from running in a default service class that has a discretionary goal. See Chapter 20, “Server address space management” on page 423 for more information on managing your WLM address spaces.

- Consider the cost of invoking a stored procedure versus the cost of network transmission for a distributed application.

  When developing distributed applications, the amount of SQL you expect to execute should be a factor in deciding whether to use stored procedures. One of the main advantages of stored procedures is that multiple SQL statements can be executed in one call to the mainframe, rather than issuing many calls over the network to do one SQL statement for each call. The trade-off for the minimum number of SQL statements where it is more efficient to call a stored procedure varies by the amount of data being selected and transmitted, but a good rule of thumb is that you should have four or more SQL statements to see a performance benefit from stored procedures. Remember that you may also have other reasons besides performance for using stored procedures, such as code reuse and security.

19.3.2 For the Language Environment

Optimize the SCEERUN library for access by exploiting the LPA and LLA MVS options for data in memory. See *z/OS 1.4 Language Environment Customization*, SA22-7564-4 for details. At a minimum, make sure the library is listed in LNKLST.

Use the Language Environment (LE) runtime options to minimize storage usage.

There are a number of LE runtime options that you can specify to minimize storage usage below the 16 MB line. They are documented in Chapter 25, *DB2 Version 9.1 for z/OS*

Application Programming and SQL Guide, SC18-9841. We repeat them here for your reference:

- HEAP(,,ANY) to allocate program heap storage above the 16 MB line
- STACK(,,ANY,) to allocate program stack storage above the 16 MB line
- STORAGE(,,,4K) to reduce reserve storage area below the line to 4 KB
- BELOWHEAP(4K,,) to reduce the heap storage below the line to 4 KB
- LIBSTACK(4K,,) to reduce the library stack below the line to 4 KB
- ALL31(ON) to indicate all programs contained in the stored procedure run with AMODE(31) and RMODE(ANY).

You can list these options in the RUN OPTIONS parameter of the CREATE PROCEDURE, or the ALTER PROCEDURE statement if they are not Language Environment installation defaults. For example, the RUN OPTIONS parameter can specify:

```
H(,,ANY),STAC(,,ANY,),STO(,,,4K),BE(4K,,),LIBS(4K,,),ALL31(ON)
```

See Appendix B, “Additional material” on page 887 for details on accessing the DDL for the CREATE PROCEDURE statement for example stored procedure EMPODB1C, which includes the above RUN OPTIONS settings.

19.3.3 For nested stored procedures

For better performance of nested stored procedures, you can force them to run in the same WLM environment with the following syntax:

```
WLM ENVIRONMENT (xxx,*)
```

DB2 uses the Workload Manager (WLM) to schedule every stored procedure that is invoked, or every UDF that is the first UDF of the cursor that is being accessed. Whether the stored procedures are nested or not is not a factor in terms of performance. The cost of using the WLM to schedule the stored procedure is the same whether the stored procedure is the highest level stored procedure in the nesting or the lowest.

You declare the Workload Manager environment in which you need to run a particular stored procedure. DB2 honors that declaration, because it assumes that your program is dependent on certain things in that WLM procedure. For instance, your program might be dependent on the STEPLIB concatenation to get the right program loaded into memory. Your program might also be dependent on certain DD cards in the procedure that provide access to specific data sets. There is a wide variety of other possible dependencies for a particular WLM procedure.

So, the question is: If I have stored procedure A defined in WLM environment 1, and stored procedure B defined in WLM environment 2, how can I force them both to run in WLM environment 1? DB2 has no mechanism for that situation. DB2 assumes that you put stored procedure B into WLM environment 2, because you had a dependency on that WLM environment, so DB2 honors that association for the life of the stored procedure. Scheduling the stored procedures in the same address space does not offer a significant performance advantage.

19.3.4 Handling result sets from DB2-supplied stored procedures

Stored procedures, including the DB2-provided and documented, and the schema stored procedures that are used by DB2 clients such as JCC, use created global temporary tables. By default, the workfile (DSNDB07) data sets to support these global temp tables are often too small, which causes synchronous I/O and contention problem. When you have a lot of

stored procedures on your subsystem, and run clients that use the JCC such as WebSphere type 2 and 4 JCC drivers, make sure that primary space is large enough and secondary is 0 for optimal performance.

20

Server address space management

In this chapter we describe how Workload Manager (WLM) manages the life cycle of server address spaces.

Read this chapter if your work is missing its performance goals and there is reason to believe that stored procedures are not being scheduled quickly enough. Criteria for establishing this are discussed in 20.2.1, “When to adjust WLM’s management of server address spaces” on page 429.

This chapter contains the following:

- WLM-established server address spaces

  This section describes in detail how WLM-established server address spaces work.
- Managing server address spaces

  This section describes how you can monitor and control WLM management of server address spaces.

20.1 WLM-established server address spaces

Note: In this chapter we only discuss the server address spaces that Workload Manager (WLM) manages. The previous implementation (the DB2-established stored procedures address space) is not discussed. We also assume that WLM is operating in *goal mode*, as *compatibility mode* is no longer available starting with z/OS V1R3.

All work in a system is assigned a service class (SC). A service class is a group of work with similar performance goals, resource requirements, or business importance. Each stored procedure or User Defined Function (UDF) is assigned a specific Application Environment (AE). A work queue is created for each combination of SC and AE. When an application invokes a stored procedure, its SC and the stored procedure's AE determine which work queue it joins.

Each work queue has at least one server address space to service its requests. WLM creates additional server address spaces as required, based on the arrival patterns of work on that queue. WLM also deletes server address spaces when they are no longer needed.

20.1.1 Task Control Blocks usage by stored procedures and UDFs

To understand how server address spaces are created and destroyed it is necessary to understand how Task Control Blocks (TCBs) are used by stored procedures and UDFs.

Each server address space contains a number of TCBs, which are ATTACHed when the address space is started. Each concurrently executing stored procedure requires at least one TCB:

- If the stored procedure does *not* call another stored procedure or invoke a UDF, a single TCB is required.
- If the stored procedure *does* call another stored procedure or invokes a UDF, additional TCBs are required. If these in turn call other stored procedures or invoke more UDFs, additional TCBs are required.

TCBs are acquired for the life of the stored procedure execution:

- When the stored procedure starts, it is scheduled to run on a TCB.

  When a top level stored procedure starts, the TCB it runs on joins the caller's WLM enclave. This is true even if the caller is not distributed. If this stored procedure calls other stored procedures or UDFs, their TCBs also join the enclave. An enclave is an independent dispatchable unit of work, which is basically a business transaction that can span multiple address spaces, and can include multiple SRBs and TCBs.
- When the stored procedure finishes, it frees up the TCB for another stored procedure to use.
- When one stored procedure calls another, the caller's TCB is suspended while the called stored procedure runs.

UDFs behave slightly differently. For a row level UDF, the TCB is retained until all rows have been processed. This means a UDF can require a TCB for the life of a unit of work, whereas a stored procedure might release it, decreasing the utilization of TCBs.

A very simple example

A batch job step calls a stored procedure, which does not call any other stored procedures or invoke any UDFs.

In this case a single TCB is required, joining the enclave created from the batch job's TCB.

A more complex example

A CICS transaction calls a stored procedure, which calls another one, which in turn calls a third one. In this case, three TCBs are required, one for each level in the hierarchy. These TCBs all join the CICS transaction's enclave.

An even more complex example

A DDF transaction calls a stored procedure. This stored procedure calls two other stored procedures, one after the other, so both of the called stored procedures are running at the second level of the hierarchy.

In this case the top-level stored procedure requires a TCB, and each of the stored procedures it calls requires its own TCB. These nested (or second-level) stored procedures run at different times, so the most TCBs required at any one time is two, not three.

The population of required TCBs depends on the stored procedures workload. In particular, the degree of concurrency and the nesting complexity of the stored procedures determine how many TCBs are needed at any time. In the nested stored procedures' case, many of these TCBs might be suspended waiting for other stored procedures they call to end.

20.1.2 NUMTCB

Each application environment (AE) is assigned its own NUMTCB value, which defaults to 8. NUMTCB specifies the maximum number of TCBs a server address space can use to concurrently process stored procedure requests for the AE it supports. Because each work queue contains requests for stored procedures from a single AE, each address space servicing that queue is subject to the same NUMTCB value, which is the same maximum number of TCBs.

The higher the NUMTCB value, the more concurrent stored procedures can be processed in parallel by a single server address space. More concurrent stored procedures in a single server address space potentially means:

- More CPU cycles consumed per second by the server address space

  Each concurrent TCB can be dispatched independently on a processor, so a larger NUMTCB allows more concurrent execution.

- More virtual storage allocated and used by the server address space

  Stored procedures may allocate and use virtual storage both below the 2 GB line and above the 2 GB line within a server address space. Each stored procedure allocates and uses its own virtual storage, to store the load module and for work areas, so a higher NUMTCB value leads to more concurrent virtual storage usage.

- More real storage used by the server address space

  An increase in virtual storage leads to an increase in real storage.

- A higher Input/output (I/O) rate to non-DB2 data accessed by the server address space

  Individual stored procedures can perform I/O at the same time, so an increase in NUMTCB can lead to a higher I/O rate.

In general, a larger NUMTCB value for a server address space means the server address space consumes more resources.

For some types of stored procedures, the NUMTCB value specified for its AE is constrained by specific considerations:

- A stored procedure written in REXX must be run in a server address space with a NUMTCB value of 1 because the REXX interpreter cannot be called from more than one TCB in an address space.
- A stored procedure written in Java must run in a server address space with a NUMTCB of *at most* 8 if you are running in a JVM started in resettable mode; otherwise, the server address space might exhaust virtual storage. If you are running in a JVM started in non-resettable mode you may be able to support a NUMTCB value of between 20 and 40, depending on the size of your procedures and memory constraints. See “Non-resettable JVMs” on page 417 for more details.
- Many DB2-supplied stored procedures, such as DSNUTILS must be run in a server address space with a NUMTCB value of 1 to ensure correct serialization of access to resources. For example, DSNUTILS uses data sets that are allocated in the JCL procedure for the WLM application environment. If more than one instance of DSNUTILS were allowed to run in the same address space, the instances would overwrite each other's data sets, leading to indeterministic behavior. Any LANGUAGE REXX stored procedures must also run in a server address space with a NUMTCB value of 1.

20.1.3 How TCBs drive the demand for server address spaces

TCBs must reside in an address space. When stored procedures run, they acquire TCBs in server address spaces. A server address space can only contain a limited number of TCBs, determined by the NUMTCB value for the queue. See 20.1.2, “NUMTCB” on page 425 for more information on NUMTCB.

For a single work queue (combination of SC and AE) there are dedicated server address spaces, and hence a predetermined number of TCBs, available. Demand for server address spaces depends on the arrival pattern of stored procedures:

- When a stored procedure request arrives, it requires a TCB to be able to start. If all the TCBs in the existing server address spaces for its work queue are already processing stored procedures, another server address space is needed.
- When a stored procedure ends, the TCB it acquired is relinquished. If there were no other TCBs in use in its server address space, there will be no TCB in use when the stored procedure ends.

WLM manages the creation and termination of server address spaces. It might not immediately create a server address space if creating a new address space would negatively impact other work in z/OS. Similarly, WLM might not immediately terminate a server address space when all of its TCBs become unused if there is a statistical likelihood that new work will arrive shortly that will acquire a TCB in the address space. See 20.1.4, “WLM management of server address spaces” on page 427 for details on how WLM's Resource Adjustment function determines when to create or terminate a server address space.

Because the creation of a server address space may be delayed, the acquisition of a TCB may be delayed, and the stored procedure might not immediately run. This has two potential consequences:

- If a stored procedure does not immediately run, the elapsed time of the calling application will be increased, perhaps to an unacceptable extent.
- Another stored procedure might terminate before the new server address space is created. If so, its TCBs become available for the waiting stored procedure to use. In this case, the new server address space will not be created, and the waiting stored procedure will reuse these newly-released TCBs.

Note: With *nested stored procedures* (where one stored procedure calls another) the number of TCBs concurrently required might be quite large, especially if the level of nesting is quite large. For this reason, nested stored procedures are particularly intensive in their use of TCBs, and particularly bursty. That is, a large number of TCBs might suddenly be acquired when a stored procedure executes. Similarly, a large number of TCBs might be relinquished almost simultaneously when a nested stored procedure terminates.

If a calling stored procedure is defined to use a different AE from the stored procedure it calls, the two stored procedures are in different work queues. In such a case, the two TCBs to run these stored procedures may be created in different server address spaces. If you define your stored procedures with the WLM ENVIRONMENT parameter of the form (name,*) then the nested stored procedure will run in the same AE as the calling stored procedure; they may or may not run in the same address space, depending on available TCBs.

20.1.4 WLM management of server address spaces

All work in a system is assigned a service class (SC) by a process called *Work Classification*. Work is classified according to such attributes as job name or transaction identifier using the WLM ISPF application. Service classes are defined using the same application. Two service class attributes are of particular importance in this discussion:

- Service class periods

  As a transaction runs, it is provided with resources, such as CPU, according to the service class to which the work is assigned. You have the capability to break down a service class into multiple time periods and assign different goals to each time period. These time periods are called service class periods.

  If you define periods for the service class, you define durations for all periods except the last. While a transaction is executing for up to the length of time defined in the duration of the first service class period, WLM will supply resources based on the definition for that period. If the transaction does not complete by the time the duration of the first service class period has been reached, then the transaction will fall into the next period and WLM will supply resources based on the definition for that period. This continues until the elapsed time of the transaction causes it to fall into the last service class period. The final period has no duration, so any transaction that executes long enough so that it falls into the last period will complete in that period, regardless of the service the transaction accumulates in the final period.

- Goals

  Each service class (or service class period, in the case of multi-period service classes) is defined to have a goal. Two types of goals are used for most DB2 work:

  – Velocity

    A velocity goal broadly specifies the ratio of the time work uses the CPU to the time work waits for the CPU.

  – Response time

    A response time goal broadly specifies how long a transaction should take to complete. Percentile response time goals are specified in terms such as *80% of transactions should complete in less than one second*. Average response time goals are expressed in such terms as *the average response time of transactions should be less than half a second.*

An important characteristic of work queues is that they service requests from a single SC. Therefore, an individual server address space services stored procedures running in one SC. With the exception of an SC with multiple periods, all work in a server address space is subject to the same goal. WLM uses the term goal to describe targets it attempts to achieve for individual service class periods.

Note: The caller of the stored procedure determines what classification rule the stored procedure runs under. The default service class assigned to that classification rule will be used. However, you do have the capability to define the service class the work will run under at a granular level. For DB2 work, you can categorize work by such criteria as package name, stored procedure name, userid and correlation ID, among other criteria. Once categorized, you can define a different service class definition for each category or individual stored procedure. This is especially useful if you have a situation where some procedures are more mission critical than others and require more stringent response time or velocity goals.

WLM manages work in the system by trying to meet two conflicting objectives:

- Meeting the goals of work running in SCs
- Optimizing the use of resources

With stored procedures, this translates into a trade-off between minimizing the delay to start new server address spaces, and minimizing the number of server address spaces:

- The faster WLM starts a new server address space, the less delay there is to the work waiting for the address space to start its stored procedure's TCB.
- The slower WLM is in starting a new server address space, the greater the chance it will not have to start it, and hence the more optimal the use of resources.

WLM's Resource Adjustment function runs every two seconds, but will only add one server address space in a 10-second interval. During resource adjustment, WLM checks goal attainment and system conditions.

Under the following circumstances WLM may create another server address space:

- Adding a server address space would help an SC meet its goals better.

  This might be the case if there is a build up of stored procedures waiting for a TCB, caused by a shortage of server address spaces.
- Adding a server address space would not impact other work in the system.

  WLM uses the resource profile of existing server address spaces servicing the work queue to determine the likely impact of adding another server address space. The main resource considered is CPU, but memory is also considered.

WLM will never start more than one new server address space for a work queue in any 10-second interval. If system conditions are unfavorable, it might take more than ten seconds for an additional server address space to be created, even if a stored procedure is delayed by this additional wait for a server address space (and hence a TCB). The 10-second minimum interval introduces latency. The 10-second cycle was chosen when WLM was developed to minimize the resources required to run WLM algorithms, and to provide enough performance data for WLM to make good decisions.

Because of this latency, it is desirable to avoid the need to create additional server address spaces. The main control for minimizing the need to create additional server address spaces is NUMTCB. By specifying a higher value of NUMTCB, more concurrent stored procedures can be run in each server address space. But, the likelihood of WLM delaying starting a new

server address space is increased the more resources the server address space is expected to consume. In other words, the larger the NUMTCB value, the more resources WLM expects the address space to consume, so WLM may delay in starting that address space to see if work will finish in the already started address space. In 20.1.2, “NUMTCB” on page 425, we describe how a server address space’s resource consumption is related to the server address space’s NUMTCB value.

The larger the NUMTCB value, the smaller the likelihood that a server address space will have no TCBs processing stored procedures. So, a large NUMTCB value reduces the need to start and stop server address spaces as the stored procedures workload fluctuates. But a large NUMTCB value may inhibit WLM from being able to adjust the number of server address spaces as the workload fluctuates.

20.2 Managing server address spaces

For most workloads server address space management is not a major effort. For some workloads an installation may need to adjust WLM’s management of server address spaces. This tuning effort draws on skills from both WLM specialists and DB2 systems programmers. It may also require changes to DB2 applications and to subsystem definitions.

Workloads that are likely to require more active management have some of the following characteristics:

- The workload is bursty rather than unvarying.
- The workload is highly complex.
- The workload has particularly stringent performance requirements.
- The workload is high volume.

20.2.1 When to adjust WLM’s management of server address spaces

To determine if you need to adjust WLM’s management of server address spaces:

1. Establish whether the work is missing business goals, or is likely to do so in the near future.

   If important work is not missing its business goals, active management of WLM’s control of server address spaces is unlikely to be required.

   Note: Business goals are not necessarily the same as WLM goals. In a well managed environment, WLM goals *do* reflect business goals. If business goals and WLM goals are in alignment, examining the attainment of WLM goals is sufficient.

2. Establish if stored procedure scheduling delays are a significant factor in the work not meeting business goals:
   - If important work is missing its business goals and *stored procedure scheduling delays are a significant factor,* you should actively manage server address spaces.
   - If important work is missing its business goals, but *stored procedure scheduling delays are not a significant factor,* you should direct your tuning efforts elsewhere.

You can check if a WLM service class is meeting its goals using Resource Management Facility’s (RMF) Workload Activity report. RMF also writes SMF Type 72 records containing the same information. Most installations use reporting software to track WLM goal attainment, using SMF Type 72 records. If the performance index for a goal is greater than 1, the goal is not attained. A value of 1 or less for the performance index means the goal was attained.

Example 20-1 shows a service class that is not meeting its goals. A value of 1.1 for PERF INDX, which is how RMF prints Performance Index, indicates the goal is not being missed by much.

Example 20-1 Portion of sample RMF Workload Activity report

```
W O R K L O A D   A C T I V I T Y

      z/OS V1R4              SYSPLEX ABCDPLEX          DATE 11/30/2003
                             RPT VERSION V1R2 RMF      TIME 00.30.00

                                        POLICY ACTIVATION DATE/TIME 11/14/2003

------------------------------------------------------------------------------

REPORT BY: POLICY=MYPOL1     WORKLOAD=PRDBATCH   SERVICE CLASS=ABCDEFGH   RESO
                                                 CRITICAL     =NONE

TRANSACTIONS    TRANS.-TIME  HHH.MM.SS.TTT   --DASD I/O--   ---SERVICE----   -
AVG      2.26   ACTUAL              35.555   SSCHRT 157.2   IOC    104774    A
MPL      2.10   EXECUTION           33.503   RESP  3891.0   CPU    174744    T
ENDED      62   QUEUED               2.052   CONN  3889.4   MSO         0    T
END/S    0.07   R/S AFFINITY             0   DISC     0.8   SRB     31796    S
#SWAPS     22   INELIGIBLE               0   Q+PEND   0.5   TOT    311314    R
EXCTD       0   CONVERSION           2.367   IOSQ     0.3   /SEC      346    I
AVG ENC  0.00   STD DEV             18.793                                   H
REM ENC  0.00                                                                A
MS ENC   0.00

VELOCITY MIGRATION:   I/O MGMT  36.2%    INIT MGMT  6.2%

        ---RESPONSE TIME---  EX   PERF   AVG   --USING%--  ------------ EXECUT
        HH.MM.SS.TTT         VEL  INDX  ADRSP   CPU   I/O  TOTAL   MPL  I/O  C
GOAL                        10.0%
ACTUALS
SYSA                        9.0%  1.1    2.4   1.0   7.6   15.2  10.0  4.6  0
```

You can check if DB2 work is delayed waiting for a server address space to be scheduled by examining two fields in DB2 Accounting Trace (SMF Type 101):

QWACCAST Wait time for a stored procedure to be scheduled

QWACUDST Wait for a User Defined Function (UDF) to be scheduled

These fields are available in the SMF 101 record only if Accounting Trace Class 3 is on.

Example 20-2 shows a DB2 application that is delayed waiting for a server address space to be created so that a stored procedure can run. The SUSPEND TIME STORED PROC field shows this application was delayed for 0.177809 seconds. As the ELAPSED TIME field is only 0.463223 seconds, this delay is a significant contributor. In short, 38% of the elapsed time is due to waiting for a stored procedure to be scheduled. In this example, the time is dominated by Class 1 time in stored procedures, the STORED PROC value being 0.423170 seconds.

Most of the time was caused by the fact that we were running a very small number of stored procedures. If we had run additional stored procedures in this test it is likely that subsequent stored procedures would not have to wait for a TCB until we had enough activity so that WLM determined it had to start another address space.

Example 20-2 Sample DB2 Performance Monitor Accounting Report listing

```
AVERAGE       APPL(CL.1)  DB2 (CL.2)
------------  ----------  ----------
ELAPSED TIME    0.463223    0.001396
 NONNESTED      0.040053    0.000691
 STORED PROC    0.423170    0.000704
 UDF            0.000000    0.000000
 TRIGGER        0.000000    0.000000

CP CPU TIME     0.009524    0.001300
 AGENT          0.009524    0.001300
  NONNESTED     0.005705    0.000675
  STORED PRC    0.003819    0.000625
  UDF           0.000000    0.000000
  TRIGGER       0.000000    0.000000
 PAR.TASKS      0.000000    0.000000

 IIPCP CPU      0.000000         N/A

IIP CPU TIME    0.000000    0.000000
 STORED PROC    0.000000    0.000000

SUSPEND TIME    0.177809    0.000000
 AGENT               N/A    0.000000
 PAR.TASKS           N/A    0.000000
 STORED PROC    0.177809         N/A
 UDF            0.000000         N/A
```

The Accounting Report summarizes transactions by interval, so it averages out the values and tends to mask bursts of activities. Often, an Accounting Trace might be needed in order to have details on all the transactions, but this will of course cause large amounts of data. Starting with DB2 V8, the classification of stored procedure and UDF wait times in the Accounting Report were changed from class 3 suspensions to class 1 suspensions, as shown in Example 20-2.

Note: Stored procedures schedule wait times and UDF schedule wait times are not calculated at the package level. The fields QPACCAST and QPACUDST always contain zeroes. The reason that these wait times are not calculated at the package level is that once the stored procedure or UDF package is loaded, the wait for scheduling is already completed. Therefore, the schedule wait time only occurs at the plan level.

20.2.2 Adjusting WLM control of server address spaces

In 20.2.1, "When to adjust WLM's management of server address spaces" on page 429, we describe when you need to tune WLM's control of server address spaces. In this section, we describe how you might do this.

You can affect WLM control of server address spaces in a number of different ways:

- You can change the NUMTCB value for the application environments (AE) your most important stored procedures run in.

  By increasing the NUMTCB value you can reduce the need to start additional server address spaces. By decreasing the NUMTCB value, you may be able to decrease the time to start another server address space. Installations will need to establish how this trade off works in their environment.

- You can modify WLM goals of the service class of the calling application.

  A more aggressive WLM goal, such as a higher execution velocity specification, might cause WLM to start additional server address spaces more quickly.

- You can change application behavior or the stored procedure definition to consume fewer resources, such that the stored procedure and the server address space are viewed as lighter weight.

  An alternative expression of this objective is that a larger NUMTCB value can be sustained for a server address space, without increasing its weight.

  Techniques to reduce the weight of a stored procedure are discussed in 20.2.3, “Reducing the resource profile of stored procedures” on page 432.

- You can separate stored procedures into different AEs.

  Separating stored procedures into different AEs ensures a larger pool of TCBs because you have more work queues. This means:

  – Before WLM creates any additional server address spaces, you have more TCBs available.

  – WLM can create more server address spaces in any ten second interval.

  If you don’t specify a WLM ENVIRONMENT when you create a stored procedure, then the stored procedure runs in the default WLM-established stored procedure address space specified at installation time via system parameter WLMENV. You should always specify a WLM ENVIRONMENT; otherwise, you have minimal control over how your stored procedure workload is managed since they will run in the default address space.

Taken together, these measures can have a significant impact on how WLM starts and stops server address spaces, and hence application performance.

20.2.3 Reducing the resource profile of stored procedures

Reducing the resources a stored procedure consumes reduces WLM’s view of its server address spaces’ weight. This increases the likelihood of a quick start when another server address space is needed. WLM deems a lighter weight server address space to be less likely to cause a resource constraint.

Techniques to reduce the weight of a stored procedure include:

- Tuning the stored procedure itself

  This will reduce the runtime of the stored procedure, even without the weight reduction benefit. A significant reduction in runtime can in turn reduce the stored procedure’s memory requirement.

  Techniques to tune the stored procedure itself include:

  – SQL tuning

  – Non-SQL application logic tuning

  – Implementation using a more efficient programming language

  – I/O tuning

  – Reducing the level of nesting

    Techniques to do this include replacing SQL CALL statements with native language subroutine calls.

  – Using subprograms rather than a main program

When successful, this technique reduces the Language Environment (LE) enclave initiation and termination cost for the stored procedure.

- Program life cycle management

  Keeping frequently reused programs resident in virtual storage (by specifying STAY RESIDENT YES), rather than continually reloading them, will reduce CPU cycles and reduce the elapsed time of the stored procedure execution. But keeping infrequently executed programs in memory increases the server address space's virtual storage requirement. For non-critical, infrequently executed programs you may wish to define them with STAY RESIDENT NO.

- Virtual storage tuning

  This can mean either reducing virtual storage usage, or increasing it. Normally, it is better to reduce virtual storage usage:

  – To reduce the requirement for real storage

  – To allow more concurrent stored procedures to be supported in a server address space without abnormal terminations

  Where garbage collection is CPU intensive it might be better to use more virtual storage to reduce the cost of garbage collection, provided you have the real storage to back the virtual storage.

These techniques often interact with each other, sometimes negatively. So select the techniques carefully.

To help analyze the use of resources by different types of stored procedures, you should name the server address spaces in such a way that it is clear which AE they serve. With this naming convention SMF Type 30 Subtypes 2 and 3 records can be used to determine the resource consumption by each server address space. This information can be used to observe the starting and stopping of WLM application environments. And furthermore, you can see the weight of the address space - in terms of (non-DB2) I/O, memory and CPU from the Type 30 records. Recall that the weight feeds into WLM decisions about whether it can afford to start another address space that services the same queue. Therefore, understanding the weight of each AE can help you in your WLM environment.

20.2.4 Recommendation - Exploit WLM server task thread management

DB2 for z/OS V8 exploits z/OS workload manager functions that allow z/OS System Resource Manager and Workload Manager to determine the appropriate resource utilization and recommend changes in the number of tasks operating inside a single WLM managed stored procedure address space.

You may wish to specify a NUMTCB value other than 1 in cases where you previously specified 1, since WLM will determine the appropriate number of tasks to run in an address space and will also determine when to start a new address space. If you specify a NUMTCB value of 1 you will not be able to take advantage of this capability. If you have sufficient available memory, NUMTCB can easily be around 60 for languages such as COBOL or C, or 20 to 40 for Java stored procedures that are run in non-resettable mode.

This recommendation only applies for stored procedures that are able to share the resources in one address space. There are still some stored procedures, for example DSNUTILS and REXX programs, which require a value of 1 in NUMTCB.

21

I/O performance management

In this chapter we discuss how to manage stored procedure I/O performance and contention issues.

Read this chapter if you believe non-DB2 I/O time and contention are a significant factor in the performance of your stored procedures. This chapter does not talk about DB2 I/O because that is not a topic specific to stored procedures. The importance of normal DB2 I/O tuning applies to stored procedures.

This chapter contains the following:

- Stored procedures I/O and ENQs

  This section describes some of the scenarios where I/O could become a significant factor in stored procedures performance.

- Managing stored procedures I/O and ENQs

  This section describes how you can monitor and control WLM management of server address spaces.

This chapter is not a primer on I/O tuning. This subject has been covered in detail in many other books. Its purpose is to highlight the nature of I/O and ENQs that can delay stored procedures.

21.1 Stored procedures I/O and ENQs

Stored procedures can perform many of the same kinds of input and output (I/O) as other types of programs. So, I/O performance might be important. In particular, stored procedures can:

- Load programs (indeed, the stored procedure itself is a loaded program).
- Access VSAM files
- Access sequential files.
- Write to SYSOUT and SYSPRINT DDs.
- Transfer control to CICS and IMS transactions to access their data.
- Access IMS data.

If your stored procedures do perform significant I/O, and performance objectives are not being met, consider tuning stored procedures access to data. Also, consider the need to tune program loading.

Because stored procedures server address spaces are multitasking, special considerations apply for accessing non-DB2 data. Some access methods do not tolerate more than one task accessing data at the same time, even if they are in the same address space. In some cases you can share the same physical resource using ENQs.

Note: If a stored procedure issues an ENQ for a resource that another stored procedure holds, the requester will be delayed until the holder issues a DEQ for that resource.

21.2 Managing stored procedures I/O and ENQs

An application-specific knowledge of the data stored procedures access is important. But knowing that a particular data set is accessed is not enough to determine whether its performance is important. There are two approaches that can be used to determine its importance:

- Assess whether a specific application's response time is impacted by the time its stored procedures take to perform I/O.

  This is impossible to do with certainty. However, a large value of Unknown Class 1 time might indicate an I/O time problem. Unknown Class 1 time can be calculated from Accounting Trace (SMF Type 101) data by the following formula:

  ```
  Unknown Class 1 time = Total Class 1 Time - Total Class 2 Time - Non DB2 CPU Time
  ```

  where

  ```
  Non DB2 CPU Time = Total Class 1 CPU Time - Total Class 2 CPU Time
  ```

  Unknown Class 1 time can contain other things, such as CPU Queueing, so it is not a precise measure of I/O time. But it might give an indication.

- Assess whether specific server address spaces demonstrate much I/O time.

  The standard measure of an address space's I/O time to disk data sets comes from SMF Type 42 Subtype 6 data. Suitably processed, these SMF records give the I/O count and I/O time to specific disk data sets, such as load libraries or VSAM files. Furthermore, I/O response time components and a cache hit ratio estimate are available for each file in this SMF record.

Notes on SMF Type 42 Subtype 6 data

This data will not give any information for cases where another address space accesses the data set. An example of this is where a stored procedure invokes a CICS transaction.

Estimates of cache hit ratio are based on sampled disconnect time, rather than a cache hit count in the disk controller. A value of less than half a millisecond is regarded as a cache hit. More than half a millisecond is regarded as a miss. This technique is susceptible to cases where disconnect time is significant, other than cache misses. An example of this is synchronous remote copy, where much of the disconnect time for a write I/O is waiting for the second copy to be written to another disk.

The easiest approach to tuning the I/O from a server address space is to establish which data sets are being the most heavily used. Summarizing the SMF Type 42 Subtype 6 data by data set across the work's peak, sorting by total I/O time produces a prioritized list of data sets to tune. Your z/OS performance analyst probably already has a job to do this reporting.

Example 21-1 is the output of such a reporting job. It is a simplified version of the PMDB2 service offering's Top Data Set report. In this case the breakdown of the response times into their components has been removed. In this example, it is a DB2 subsystem's DBM1 address space whose data sets are shown. As this is a large DB2 subsystem, it is not surprising that the top ten data sets only represent 5.7% of the I/O subsystem's I/O time.

Example 21-1 Sample top 10 data set impact report

Data Set Name	Total Minutes	Cumul Percent	Response MS	Hit %
***** Total *****	12279.6	100.0	15.8	59.7
DB2PROD.DSNDBD.ABC.ABC000.I0001.A013	96.6	0.8	11.2	58.4
DB2PROD.DSNDBD.ABC.ABC000.I0001.A016	79.0	1.4	9.3	71.0
DB2PROD.DSNDBD.ABC.ABC000.I0001.A014	77.5	2.1	9.0	63.4
DB2PROD.DSNDBD.ABC2.CIS001.I0001.A005	74.8	2.7	17.8	55.4
DB2PROD.DSNDBD.ABC.ABC000.I0001.A011	71.8	3.3	8.3	69.2
DB2PROD.DSNDBD.ABC.XABC083B.I0001.A001	63.1	3.8	7.9	52.7
DB2PROD.DSNDBD.ABC.ABC000.I0001.A010	62.8	4.3	7.3	66.7
DB2PROD.DSNDBD.ABC.ABC000.I0001.A012	60.2	4.8	7.0	71.0
DB2PROD.DSNDBD.ABC.ABC000.I0001.A015	58.6	5.2	6.9	70.2
DB2PROD.DSNDBD.ABC.ABC000.I0001.A008	49.6	5.7	5.8	71.0

Having established a list of data sets to work on, tune each data set based on its specific characteristics. Some frequently encountered stored procedures data sets are:

- Load libraries

  Each stored procedure execution requires a load module to run. If the load module is not already in server address space memory, or if the stored procedure is unable to use an in memory copy, the load module must be fetched. If the STAY RESIDENT YES and PROGRAM TYPE SUB options have been specified, and the load module has been link edited with the RENT option, the likelihood is very high that the load module will be in the server address space's memory and usable by the stored procedure. In production environments, with little change to the stored procedure's program logic, these options are recommended.

  If the load module must be fetched, the z/OS Library Lookaside (LLA) function can be used. LLA buffers load modules and load library directories in memory using the Virtual Lookaside Facility (VLF). VLF tracks the number of times each module is loaded. Frequently loaded modules are buffered in VLF data spaces.

It may be necessary to increase the size of VLF's cache. VLF is setup for LLA using statements in a COFVLFxx member in SYS1.PARMLIB. Example 21-2 shows how many installations have set up LLA. MAXVIRT(4096) specifies that the LLA module cache will occupy 4096 4 KB virtual storage pages, requiring 16 MB of memory. This is the default. If you have not specified MAXVIRT, a 16 MB cache will be used. A good minimum value with z/Architecture machines is 64 MB, requiring MAXVIRT(16384) to be specified. With stored procedures, many more load modules may need caching. So, MAXVIRT(32768) may be better. Check with your z/OS performance analyst about what size the LLA module cache should be in your environment, as memory constraints or other users of LLA may determine an appropriate value.

Example 21-2 Sample LLA definition to VLF

```
CLASS NAME(CSVLLA)
      EMAJ(LLA)
      MAXVIRT(4096)
```

- SYSPRINT data sets

  To prevent abends for each stored procedure that uses the SYSPRINT DD, specify Language Environment (LE) runtime option MSGFILE(SYSPRINT,,,,ENQ). This causes writes to the SYSPRINT DD to be serialized. Reduce the probability of stored procedures contending with each other by minimizing their use of this DD. Writes to SYSPRINT that were coded in development should be removed where possible in production.

- VSAM and non VSAM data sets

  There may be problems if more than one stored procedure uses these data sets concurrently.

  Ensure appropriate VSAM and non VSAM data set tuning options, such as buffering, are used. It is beyond the scope of this book to describe these options. There are many books that provide data set tuning guidance.

Part 5

Extending the functions

In this part we provide additional information on functions that extend the reach of standard procedures. These topics are probably for the more advanced users of stored procedures.

This part contains the following chapters:

22

Multi-threaded stored procedures in the C language

In this chapter we explore the possibility of multi-threading in stored procedures. If the function executed in a stored procedure is complex and can be split and assigned to multiple concurrently running threads, then, as for any other case of concurrent actions, multi-threading can largely reduce execution time and improve performance.

C is the only high-level programming language that allows you to write a stored procedure with secondary threads, where each thread can establish its own connections to the same or different subsystems.

This chapter contains the following:

- Purpose of multi-thread stored procedures
- Which style threads to use
- Case study: Stored procedure that runs RUNSTATS in parallel
- Compiling the stored procedure
- Authorization issues - Best practices
- Improvements
- Common design problems using multiple threads

22.1 Purpose of multi-thread stored procedures

If the function performed in your stored procedure is complex enough and can be split into several, concurrently executing threads of execution, then you may consider implementing a stored procedure that uses multiple threads, thus dramatically reducing execution time and boosting performance.

There are two advantages to implementing multiple threads in the stored procedure rather than on the client:

- The complexity of the code on the client is reduced to a single SQL CALL statement.

  Even programming languages that do not support multi-threading can make use of a multi-threaded stored procedure, with the potential of dramatically improving performance.
- Stored procedures running on a z/OS server benefit from the computing power and scalability of the platform.

  Creating threads on the server performs considerably better than creating threads on the client because there is no network communication overhead.

The primary thread of the stored procedure typically acts as the scheduler: it creates secondary threads and synchronizes with them to exchange data using shared variables. Synchronization of the control flow of the primary and secondary threads as well as serialization of access to shared variables is accomplished using interprocess communication mechanisms including semaphores and condition variables.

The primary thread implicitly uses RRSAF calls. You do not have to establish a database connection explicitly any more. If you include explicit attachment facility calls in the primary thread, DB2 rejects the calls. When you enter the main routine of the stored procedure you are in the unit of work from the client that called the stored procedure.

However, when you create secondary threads from your main routine, these threads have no implicit database connection. If required, you have to explicitly establish a database connection using RRSAF from every thread that has to execute SQL statements.

Since DB2 considers the WLM-established stored procedure address space an allied address space, it is of critical importance that the termination of these secondary threads occurs in a coordinated manner, when all SQL activity in the secondary threads has ended. The case study demonstrates how to do this.

Also, it is important to understand the accounting trace consequences. The attached secondary threads are serviced by their own separate DB2 agents, so they have their own accounting records. They are not be rolled up into the caller's accounting records. In general, a design where all SQL is contained in the primary thread and the secondary threads are used to speed up complex computations is recommended over making RRSF connections in secondary threads in a stored procedure.

You can have all your threads connect to the subsystem the primary thread is connected to or different subsystems e.g. different members on a data sharing system for load balancing.

If you are not familiar with terms relating to the development of multi-threaded applications (such as critical section, mutex, condition variable, or semaphore), consult the chapter “Using threads in z/OS UNIX System Services applications” of *z/OS C/C++ Programming Guide*, SC09-4765-08.

Examples of how to use mutex objects and condition variables can be found in *z/OS C/C++ Programming Guide*, SC09-4765-08 and *z/OS C/C++ Run-Time Library Reference*, SA22-7821-09.

22.2 Which style threads to use

On z/OS you can use MTF or POSIX-style threads. If your threads have to connect to DB2, you *have to* use POSIX-style threads. POSIX style threads are available on almost any platform that makes your multi-threaded C stored procedure code portable.

22.3 Case study: Stored procedure that runs RUNSTATS in parallel

Our sample multi-threaded stored procedure is a simple utility scheduler that accepts a list of table spaces defined in an input table, and runs the RUNSTATS utility on them in parallel threads. This results in faster execution compared to running RUNSTATS sequentially.

Example 22-1 shows the definition of the created global temporary table in which the calling application inserts the names of the table spaces.

Example 22-1 CREATE global temporary table

```
CREATE GLOBAL TEMPORARY TABLE DEVL7083.RSP_TBL
  ( DBNAME CHAR(8) NOT NULL,
    TSNAME CHAR(8) NOT NULL)
    CCSID
EBCDIC;
```

After inserting the table spaces, the calling application calls the stored procedure RUNSTATP defined in Example 22-2.

Example 22-2 CREATE RUNSTATP

```
CREATE PROCEDURE DEVL7083.RUNSTATP
  ( OUT UTILITIES_EX INTEGER
  , OUT HIGHEST_RETCODE INTEGER
  , OUT RETCODE INTEGER
  , OUT MESSAGE VARCHAR(1331) CCSID EBCDIC)
  RESULT SETS 1
  EXTERNAL NAME RUNSTATP
  LANGUAGE C
  PARAMETER STYLE GENERAL WITH NULLS
  MODIFIES SQL DATA
  WLM ENVIRONMENT WLMENVR
  STAY RESIDENT NO
  COLLID DEVL7083
  PROGRAM TYPE MAIN
  RUN OPTIONS 'TRAP(OFF),STACK(,,ANY,),POSIX(ON)'
  COMMIT ON RETURN NO
  SECURITY USER
  ASUTIME NO
LIMIT;
```

The stored procedure does not require any input parameters. After successful execution, UTILITIES_EX contains the number of utility executions; the HIGHEST_RETCODE is the highest DSNUTILS return code, the RETCODE from RUNSTATP itself (in case there was a runtime or SQL error) and a message area.

It is required to use the run option POSIX(ON), otherwise the POSIX calls fail.

Note: The programmer needs to be sure that all threads complete, otherwise the “undub” call we make will fail and the stored procedure will be reported as failing.

Every thread the stored procedure creates requires a TCB in the WLM address space. Ensure that NUMTCB of the WLM application environment equals or is greater than the maximum number of threads that your stored procedure creates plus 1 (for the main thread). Otherwise, thread execution will be serialized.

Like DSNUTILS, RUNSTATP inserts the output from all utilities into a created global temporary table, and opens a cursor on it before it returns. Example 22-3 shows the definition of that table.

Example 22-3 Creating a global temporary table for SYSPRINT

```
CREATE GLOBAL TEMPORARY TABLE DEVL7083.RSP_SYSPRINT
  ( SEQNO INTEGER NOT NULL,
    TEXT VARCHAR(254))
    CCSID EBCDIC;
```

The code snippet from the calling Java application (shown in Example 22-4) helps to better understand how to use RUNSTATP. After getting the connection, the calling program sets AutoCommit to false, so that after inserting into the global temporary table the instance of the table does not get reset by a COMMIT. We insert the table space names into the parameter table before calling RUNSTATP. In our example, a list of two table space names is inserted into the table. After that, RUNSTATP is called to run RUNSTATS on them in parallel.

Example 22-4 Handling the parameters

```
        con.setAutoCommit(false);
        // Prepare the statements for the RSP_TBL parameter tables
        ps = con.prepareStatement("INSERT INTO DEVL7083.RSP_TBL VALUES (" +
                                  "?, ?)");

        ps.setString(1, "DSN7D71A");               // Database name
        ps.setString(2, "DSN7S71E");               // Table space name
        ps.executeUpdate();
        ps.setString(1, "DSN7D71A");
        ps.setString(2, "DSN7S71D");
        ps.executeUpdate();
        ps.close();

        // Execute utilities in parallel
        cs = con.prepareCall("CALL DEVL7083.RUNSTATP(?, ?, ?, ?)");
        cs.registerOutParameter(1, Types.INTEGER);  // Utilities executed
        cs.registerOutParameter(2, Types.INTEGER);  // Highest DSNUTILS return code
        cs.registerOutParameter(3, Types.INTEGER);  // Return code
        cs.registerOutParameter(4, Types.VARCHAR);  // Message area
        hasResultSet = cs.execute();
```

Example 22-5 shows how to check for errors after the RUNSTATP call. The result set contains utility messages and those should be printed if available. Even when the execution stopped after just one utility and the return code is higher than 0, it is important to print whatever output the utilities produced.

Next, the RUNSTATP return code rc is queried. If rc is greater than zero, an error occurred in the stored procedure and we need to print the error message, which indicates the location

where the error occurred. If RUNSTATP executed successfully, we still check if the utilities ran to completion by checking if the highest DSNUTILS return code is greater than 4.

The execution needs operator attention and intervention if either the RUNSTATP return code was greater than 0, or the highest DSNUTILS return code was greater than 4.

Example 22-5 Error checking

```
        if (hasResultSet)
        {
          rs = cs.getResultSet();
          while (rs.next())
            System.out.println(rs.getString(2));
          rs.close();
        }

        int highestDSNUTILSretCode = cs.getInt(2);
        rc = cs.getInt(3);
        message = cs.getString(4);
        if (rc > 0)
        {
         errorMessage = "RUNSTATP execution failed: " + message;
          throw new DB2RUNSTATPException(rc, errorMessage);
        }
        else
        {
          // Check if the highest SYSPROC.DSNUTILS return code
          // requires an exception to be thrown
          if (highestDSNUTILSretCode > 4)
          {
            errorMessage = "Utility execution failed. Highest DSNUTILS return code: " +
                           highestDSNUTILSretCode;
            throw new DB2RUNSTATPException(rc, errorMessage);
          }
        }

        cs.close();
        System.out.println("DB2Runstats successful.");
      con.commit();
```

Now we take a closer look at the stored procedure itself. In the following paragraph, we discuss the listing. As listed in Chapter 11, “C programming” on page 147 the first element is “Includes and compiler defines.”

You have to include pthread.h and define _OPEN_THREADS to use POSIX threads in your stored procedure. Each thread requires its own user-defined SQLCA to avoid having to serialize its usage. Instead of placing EXEC SQL INCLUDE SQLCA in the global scope, use #include <sqlca.h> and add structure sqlca at the beginning of any routine that uses SQL. See Example 22-6.

Example 22-6 Includes and defines

```
/*********************************************************************/
 /* Includes and compiler defines.                                    */
 /*********************************************************************/
 #define _OPEN_THREADS                /* Required for POSIX threads */
 #ifdef DEBUG                         /* File options for debugging */
   #pragma runopts(plist(os),msgfile(OUT1))
   #define OUT stderr
```

```
    #else
      #pragma runopts(plist(os))
    #endif
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <pthread.h>                 /* Required for POSIX threads */
    #include <ctype.h>                   /* Required for type checking */
    #include <sqlca.h>                   /* Required SQLCA definitions */
    #pragma linkage(dsntiar, OS)         /* SQL message translation    */
    #pragma linkage (dsnrli,OS)          /* RRSAF language interface   */
    #pragma csect (CODE,"RUNSTATP")      /* Names code segment         */
    #pragma csect (STATIC,"RUNSTATS")
```

Next, we define constants and messages. See Example 22-7.

Example 22-7 Constants and messages

```
/*********************************************************************/
 /* Define constants.                                                 */
 /*********************************************************************/
 #define     RETSEV               12  /* Severe error return code   */
 #define     RETOK                 0  /* No error return code       */
 #define     MSGROWLN            121  /* Length of an errmsg line   */
 #define     DATA_DIM             10  /* Number of message lines    */
 #define     BLANK               ' '  /* Buffer padding             */
 #define     LINEFEED           0x25  /* Linefeed character         */
 #define     NULLCHAR           '\0'  /* Null character             */
 #define     TRUE                  1  /* Logical TRUE value         */
 #define     FALSE                 0  /* Logical FALSE value        */
 #define     MAX_OBJECTS          99  /* Maximum number of objects  */
 #define     RRS_PLAN        "?       "/* When a collection name is  */
                                       /* provided instead of a plan */
                                       /* name, a ? has to be put in */
                                       /* the plan name              */
#define     RRS_COLLECTION  "DEVL7083          " /* Collection name */
                                       /* for RRSAF connection       */

 /*********************************************************************/
 /* Define messages.                                                  */
 /*********************************************************************/
 #define ERR_OPEN_RS_CSR       "*** SQL error when opening result set \
cursor..."
 #define ERR_CLR_RS_TBL        "*** SQL error when clearing result \
table..."
 #define ERR_THD_IDENTIFY      "Error in utility thread RRSAF \
IDENTIFY call..."
 #define ERR_THD_SIGNON        "Error in utility thread RRSAF \
SIGNON call..."
 #define ERR_MAX_OBJECTS       "Too many objects in input table..."
 #define ERR_THD_CREATE_THREAD "Error in utility thread RRSAF \
CREATE THREAD call..."
 #define ERR_MALLOC_SYSPRINT   "Unable to allocate memory for \
rows from SYSIBM.SYSPRINT..."
 #define ERR_CALL_UTILS        "*** SQL error when calling \
SYSPROC.DSNUTILS..."
 #define ERR_COUNT_UTILS_ROWS  "*** SQL error when counting rows \
from SYSIBM.SYSPRINT..."
 #define ERR_ASSOC_SYSPRINT    "*** SQL error when associating result \
set locator with SYSPROC.DSNUTILS..."
```

```
 #define ERR_ALLOC_SYSPRINT    "*** SQL error when allocating cursor \
for SYSPROC.DSNUTILS result set locator..."
 #define ERR_FETCH_SYSPRINT    "*** SQL error when fetching from \
SYSIBM.SYSPRINT table..."
 #define ERR_CLOSE_SYSPRINT    "*** SQL error when closing cursor \
from SYSIBM.SYSPRINT..."
 #define ERR_THD_COMMIT        "*** SQL error when committing \
changes in utility thread..."
 #define ERR_COUNT_RSP_TBL      "*** SQL error when counting input \
table rows..."
 #define ERR_MALLOC_PARMS      "Unable to allocate memory for thread \
parameters..."
 #define ERR_CALL_SS           "*** SQL error when calling \
SYSPROC.ADMIN_INFO_SSID..."
 #define ERR_OPEN_TS_IN        "*** SQL error when opening cursor for \
input table..."
 #define ERR_CLOSE_TS_IN       "*** SQL error when closing cursor for \
input table..."
 #define ERR_FETCH_TS_IN       "*** SQL error when fetching from \
input table..."
 #define ERR_THD_COMMIT        "*** SQL error when committing \
changes in utility thread..."
 #define ERR_THD_CREATE        "Error creating a utility thread..."
 #define ERR_THD_JOIN          "Unable to join thread..."
 #define OK_COMP               "Parallel utility execution \
completed successfully..."
 #define ERR_THD_TERMINATE_IDENTIFY "Error in utility thread RRSAF \
TERMINATE IDENTIFY call..."
 #define ERR_THD_TERMINATE_THREAD "Error in utility thread RRSAF \
TERMINATE THREAD call..."
 #define ERR_INSERT_RSP_SYSPRINT "*** SQL error when inserting into \
RSP_SYSPRINT..."
 #define ERR_DSNTIAR           "DSNTIAR could not detail the SQL \
error..."
```

The data types in Example 22-8 are required so that the primary thread can communicate with the secondary threads. For every secondary thread, the primary thread allocates thread parameters, which are a data area that the calling thread initializes with input parameters to tell the secondary thread what to do, and the secondary thread will set output parameters to tell the primary thread what it has done.

Writing to and reading from this data area has to be synchronized. In the THREAD_PARMS structure, all input variables are prefixed with i_ , and all output variables are prefixed with o_ for clarity. The secondary thread, which executes the online utility and passes back the output, has to allocate memory for the output and pass them back using a pointer in the THREAD_PARMS variable. SYSPRINT_LINE is the data type that we defined to hold a single output line. A number of structures including the release information block (RIB) are used for RRSAF calls.

Example 22-8 Data types

```
/*********************************************************************/
 /* Define structures, enums and types.                               */
 /*********************************************************************/
 typedef int BOOL;                    /* Boolean type                 */

 typedef char ERR_MSG[DATA_DIM+1][MSGROWLN]; /* Error message type   */

 typedef struct                       /* DSNUTILS output line         */
```

```
{
  long int seqno;
  char text[255];
  long int ind;
} SYSPRINT_LINE;

typedef struct                        /* DSNUTILS thread parameters */
{
  short int i_thdindex;               /* Thread index needed for    */
                                      /* building utility ID        */
  pthread_t i_thdid;                  /* Thread-id of created thread*/
                                      /* needed for pthread_join    */
  char i_ssid[5];                     /* Subsystem ID to attach to  */
  char i_dbname[9];                   /* Database name              */
  char i_tsname[9];                   /* Tablespace name            */
  BOOL o_error;                       /* Set to true when error     */
  ERR_MSG o_errmsg;                   /* Error message in case of   */
                                      /* runtime or SQL error       */
  long int o_utretcode;               /* DSNUTILS return code       */
  long int o_numsysprintlines;        /* Num DSNUTILS SYSPRINT lines*/
  SYSPRINT_LINE *p_osysprintlines;    /* Ptr tp sysprint lines array*/
} THREAD_PARMS;

typedef struct                        /* Release information block  */
{
  unsigned char filler[17];           /* First 17 bytes not needed  */
  unsigned char ribrel[3];            /* Release identifier e.g. 710*/
                                      /* VERSION,                   */
                                      /* RELEASE,                   */
                                      /* MODIFICATION,              */
} RIB;

typedef struct                        /* RRSAF attach block         */
{
  int somewords[100];
} ATTACH_BLOCK;

typedef struct                        /* EIB                        */
{
  unsigned char eibcode[2];
  unsigned char eibtlen[2];
  unsigned char eibeyec[4];
  unsigned char eibssid[4];
  unsigned char eibgatt[4];
  unsigned char eibgrpn[8];
  unsigned char eibmbrn[8];
  unsigned char eibrsv[16];
} EIB;

typedef char RRS_CORRID[13];          /* RRS signon correlation-ID  */

/********************************************************************/
/* Declare global variables.                                        */
/********************************************************************/
unsigned char rrs_funcs[6][19] = {"IDENTIFY          ",
                                  "SIGNON            ",
                                  "CREATE THREAD     ",
                                  "TERMINATE THREAD  ",
                                  "TERMINATE IDENTIFY",
                                  "TRANSLATE         "};
```

```
/* 22 blanks that can be used for various RRS parms */
unsigned char rrs_parm_blanks[] = "                      ";
```

Before we code the main function, we define the error function as shown in Example 22-9.

Example 22-9 Defining error functions

```
/*********************************************************************/
 /* Define functions.                                                 */
 /*********************************************************************/
 void sql_error(char[], ERR_MSG *, long int *, struct sqlca *);
 void * dsnutils_thread(void *);
 char * trim(char *);
```

sql_error can no longer format a global SQLCA and set a global error message and global return code. Hence, the signature of sql_error now contains pointers to these structures, which are declared in the scope of each function. dsnutils_thread is the function that will be run in secondary threads.

Since all data is passed through a THREAD_PARMS variable, the thread function needs a pointer to that variable, which we pass as an argument.

The main routine declares the sqlca locally to have its own copy. We will use the DB2-supplied stored procedure SYSPROC.ADMIN_INFO_SSID to find out the SSID we need to declare variables for it, as shown in Example 22-10.

Example 22-10 Declaring variables

```
/*********************************************************************/
 /* Main routine of the stored procedure.                             */
 /*********************************************************************/
 main(int argc, char *argv[])
 {
   long int stmtno_executed;          /* OUT param UTILITIES_EX     */
   long int highest_rc;               /* OUT param HIGHEST_RETCODE  */
   long int rc;                       /* OUT param RETCODE          */
   ERR_MSG errmsg;                    /* OUT param MESSAGE          */
                                      /* Local variables            */
   short int thdindex;                /* Thread index               */
   short int locind[4];               /* Indicator variables        */
   short int * pind;                  /* Pointer to indicator vars  */
   char * pcurbyte;                   /* Pointer for copying errmsg */
   int i, j;                          /* Loop control               */
   long int ret_sysprintlines;
   THREAD_PARMS * pthread_parms;      /* Pointer to utility         */
                                      /* thread parameters array    */
   THREAD_PARMS * pcurrthread_parms;
   void * thd_ret;                    /* Thread return pointer      */
   int pthread_rc;                    /* pthread_ function rc       */
   struct sqlca sqlca;                /* SQL communication area     */

   EXEC SQL BEGIN DECLARE SECTION;
   long int h_inputrows;              /* Number of input rows       */
   char h_dbname[9];                  /* Database name              */
   char h_tsname[9];                  /* Tablespace name            */
   long int h_seqno;
   char h_text[255];
   short int i_text;
   /* ADMIN_INFO_SSID host variables                                 */
   char h_ssid[5];                    /* Subsystem ID               */
   long int h_ssrc;                   /* Return code                */
   char h_ssmessage[121];             /* Return message             */
   EXEC SQL END DECLARE SECTION;
```

When you have a large undetermined number of parameters that do not fit into an SQLDA, passing them using one or more global temporary table is a good choice. Two cursors have to be declared: One cursor to return the collected SYSPRINT output lines in a global temporary table, and one cursor to get the input parameters. See Example 22-11 for the cursor declarations.

Example 22-11 Declaring cursors

```
/*****************************************************************/
   /* Declare cursors.                                           */
   /*****************************************************************/
EXEC SQL DECLARE OUT_CSR            /* Result set cursor          */
             CURSOR WITH RETURN WITH HOLD FOR
             SELECT SEQNO, TEXT
               FROM RSP_SYSPRINT
           ORDER BY SEQNO;

   EXEC SQL DECLARE TS_IN CURSOR FOR   /* Input objects cursor      */
             SELECT DBNAME, TSNAME
               FROM RSP_TBL
                FOR FETCH ONLY WITH UR;
```

Next, we initialize the variables and count the number of input lines, as shown in Example 22-12. In our simple example, we create one thread per table space and we limit the number of threads that we create to 99 (MAX_OBJECTS).

Example 22-12 Initializing variables

```
/*****************************************************************/
   /* Initialize variables and OUT parameters.                   */
   /*****************************************************************/
   stmtno_executed = 0;                /* Number of executed         */
                                       /* DSNUTILS calls             */
   highest_rc = RETOK;                 /* Initialize highest rc w/ 0 */
   rc = RETOK;                         /* Initialize rc with 0       */
   thdindex = 0;
   memset(errmsg, NULLCHAR, sizeof(errmsg));  /* Clear errmsg buffer */
   ret_sysprintlines = 0;
   pthread_parms = NULL;               /* Ptr to thread params array */

EXEC SQL SELECT COUNT(*) INTO :h_inputrows
              FROM RSP_TBL;
if (SQLCODE != 0)
     sql_error(ERR_COUNT_RSP_TBL, &errmsg, &rc, &sqlca);
   else
   {
     if (h_inputrows > MAX_OBJECTS)    /* Only allow max 99 parallel */
     {
       strcpy(errmsg[0], ERR_MAX_OBJECTS);
       rc = RETSEV;
     }
   }
```

We need to determine how many threads we are going to start in order to allocate the required THREAD_ PARM variables properly. See Example 22-13.

Example 22-13 Allocating data structures

```
   /*******************************************************************/
   /* Allocate data structures for parallel utility execution.        */
   /*******************************************************************/
   if (rc < RETSEV)
   {
     if ((pthread_parms = (THREAD_PARMS *) /* Allocate params array  */
          malloc(h_inputrows * sizeof(THREAD_PARMS))) == NULL)
     {
         strcpy(errmsg[0], ERR_MALLOC_PARMS);
         rc = RETSEV;
     }
   }
```

Next, we call ADMIN_INFO_SSID to determine the SSID of the subsystem we are connected to with the code shown in Example 22-14. The SSID is a required parameter to make an RRSAF connection in the secondary threads. If we wanted our threads to connect to members of a data sharing system, we could find out this information by issuing a -DISPLAY GROUP command using the DB2-supplied stored procedure SYSPROC.ADMIN_COMMAND_DB2.

Example 22-14 Determining the subsystem ID

```
/*******************************************************************/
   /* Determine the current subsystem ID for RRSAF connection.        */
   /*******************************************************************/
   if (rc < RETSEV)
   {
     EXEC SQL CALL SYSPROC.ADMIN_INFO_SSID(:h_ssid, :h_ssrc, :h_ssmessage);
     if (SQLCODE != 0)
       sql_error(ERR_CALL_SS, &errmsg, &rc, &sqlca);
     else
     {
       if (h_ssrc != RETOK)            /* SSID could not be queried  */
       {
         strcpy(errmsg[0], h_ssmessage);
         rc = RETSEV;
       }
     }
   }
```

Now, we are ready to fetch the table spaces from the input table, initialize the thread parameters, and use pthread_create to create a thread. pthread_create returns a thread_id, which we save in the thread parameters. See Example 22-15. We need that thread ID to later synchronize with the thread.

Example 22-15 Input table spaces and thread IDs

```
/*******************************************************************/
   /* Fetch all input table spaces.                                   */
   /*******************************************************************/
   if (rc < RETSEV)
   {
     EXEC SQL OPEN TS_IN;
     if (SQLCODE != 0)
       sql_error(ERR_OPEN_TS_IN, &errmsg, &rc, &sqlca);
     else
     {
       for (i = 0; i < h_inputrows && rc < RETSEV; i++)
```

```
      {
        EXEC SQL FETCH TS_IN
                 INTO :h_dbname, :h_tsname;
        if (SQLCODE != 0)
          sql_error(ERR_FETCH_TS_IN, &errmsg, &rc, &sqlca);
        else
        {                              /* Start DSNUTILS thread      */
          (pthread_parms + i)->i_thdindex = thdindex;
          strcpy((pthread_parms + i)->i_ssid, h_ssid);
          strcpy((pthread_parms + i)->i_dbname, trim(h_dbname));
          strcpy((pthread_parms + i)->i_tsname, trim(h_tsname));
          if (pthread_create(&((pthread_parms + i)->i_thdid),
                             NULL, dsnutils_thread,
                             (pthread_parms + i)) != 0)
          {
            strcpy(errmsg[0], ERR_THD_CREATE);
            rc = RETSEV;
          }
          else
          {
            stmtno_executed++;
            thdindex++;
          }
        }
      }

      EXEC SQL CLOSE TS_IN;            /* Always close cursors       */
      if (SQLCODE != 0)
        sql_error(ERR_CLOSE_TS_IN, &errmsg, &rc, &sqlca);
    }
  }
```

After all the secondary threads have been created and are running, we have to wait for them to finish. In Example 22-16 we show how we join each thread using pthread_join, and then insert its output lines into the global temporary output table. After we insert the output lines, we free the allocated memory.

Example 22-16 Combining the output

```
for (i = 0; i < thdindex; i++)
  {                                    /* Wait for each thread       */
    pcurrthread_parms = pthread_parms + i;
    pthread_rc = pthread_join(pcurrthread_parms->i_thdid, &thd_ret);
    if (pthread_rc != 0)
    {
      strcpy(errmsg[0], ERR_THD_JOIN);
      rc = RETSEV;
      continue;
    }

    if (pcurrthread_parms->o_error == TRUE)
    {
      memcpy(errmsg, pcurrthread_parms->o_errmsg,
             sizeof(pcurrthread_parms->o_errmsg));
      rc = RETSEV;
    }

    if (pcurrthread_parms->o_utretcode > highest_rc)
      highest_rc = pcurrthread_parms->o_utretcode;
```

```
  /* Insert the sysprint output                        */
  if (pcurrthread_parms->o_numsysprintlines > 0 &&
      pcurrthread_parms->p_osysprintlines != NULL)
  {
    for (j = 0; j < pcurrthread_parms->o_numsysprintlines; j++)
    {
      h_seqno = ret_sysprintlines;
      strcpy(h_text,
             (pcurrthread_parms->p_osysprintlines + j)->text);
      i_text = (pcurrthread_parms->p_osysprintlines + j)->ind;

      EXEC SQL INSERT INTO RSP_SYSPRINT (SEQNO, TEXT)
                          VALUES (:h_seqno, :h_text:i_text);
      if (SQLCODE != 0)
      {
        sql_error(ERR_INSERT_RSP_SYSPRINT, &errmsg, &rc, &sqlca);
        break;
      }
      else
        ret_sysprintlines++;
    }

    free(pcurrthread_parms->p_osysprintlines);
  }
}
```

Finally, we return the results and give the control back to the caller as shown in Example 22-17.

Example 22-17 Returning results and control

```
/*********************************************************************/
/* Return results.                                                   */
/*********************************************************************/
/* Open the cursor to the result set table on the way out */
if (ret_sysprintlines > 0)
{
  EXEC SQL OPEN OUT_CSR;
  if (SQLCODE != 0)
    sql_error(ERR_OPEN_RS_CSR, &errmsg, &rc, &sqlca);
}

/* Set and return OUT parameters */
/* Utilities_ex */
*(long int *) argv[1] = stmtno_executed; /* Number of exec. stmts */
locind[0] = 0;                       /* Tell DB2 to transmit it   */

/* Highest_retcode */
*(long int *)argv[2] = highest_rc;   /* Copy highest_rc to out par*/
locind[1] = 0;                       /* Tell DB2 to transmit it   */

/* Return code */
if (rc == RETOK)
  strcpy(errmsg[0], OK_COMP);
*(int *) argv[3] = rc;               /* Copy rc to out param      */
locind[2] = 0;                       /* Tell DB2 to transmit it   */

/* Return message */
if (errmsg[0][0] == BLANK)           /* If no error message exists*/
  locind[3] = -1;                    /* tell DB2 not to send one  */
```

```
  else                                /* otherwise copy it over and*/
  {                                   /* tell DB2 to transmit it   */
    pcurbyte = argv[4];               /* Set helper pointer and    */
    for (i = 0; i < DATA_DIM + 1; i++) /* parse a row, looking for  */
    {                                 /* the end of its msg text   */
      for (j=0;
          (errmsg[i][j] != NULLCHAR && j < MSGROWLN);
          j++)
        *pcurbyte++ = errmsg[i][j];   /* Copy non-null bytes       */
      if (j>0)
        *pcurbyte++ = LINEFEED;       /* Add linefd to end of row  */
    }

    *pcurbyte = NULLCHAR;             /* Null-terminate the buffer */
    locind[3] = 0;                    /* Tell DB2 to transmit it   */
  }

  /* Return indicator variables */
  pind = (short int *)argv[5];        /* Locate and recast arg     */
  for (j = 0; j < 4; j++)             /* Copy over null-ind array  */
  {
    *pind = locind[j];
    pind++;
  }

  if (pthread_parms != NULL)
    free(pthread_parms);
  /* Return control to caller  */
}
```

We will not list sql_error or trim here. These functions are very similar to the ones in Chapter 11, “C programming” on page 147. You can download the complete source code from the Web as additional material. Download instructions can be found in Appendix B, “Additional material” on page 887.

Next, we look at the function that runs DSNUTILS in a secondary thread. This is shown in Example 22-18. Like the main thread, it declares its own SQLCA SQL parameters to be used when calling DSNUTILS.

Example 22-18 Function that calls DSNUTILS in a secondary thread

```
/*********************************************************************/
/* Thread, which calls DSNUTILS.                                     */
/*********************************************************************/
void *dsnutils_thread(void *arg)
{
  THREAD_PARMS * pthread_parms;       /* Pointer to the thread parms*/
  char index[3];                      /* Thd index string for ut ID */
  RIB * prib;                         /* Local pointer to the RIB   */
  EIB * peib;                         /* Local pointer to the EIB   */
  short int rc;                       /* RRSAF func call return code*/
  long int rli_rc;                    /* RRSAF function return code */
  long int rli_reas;                  /* RRSAF function reason code */
  long int i;
  RRS_CORRID corrid;                  /* RRSAF conn correlation ID  */
  unsigned char planname[9];          /* Plan name for CREATE THREAD*/
  unsigned char collection[19];       /* Coll name for CREATE THREAD*/
  long int dummy_rc;                  /* Dummy rc for sql_error     */
  struct sqlca sqlca;                 /* SQL communication area     */
  EXEC SQL BEGIN DECLARE SECTION;     /* Host variables for util thd*/
```

```
   char h_uid[17];                    /* Host vars for DSNUTILS call*/
   char h_restart[9];
   char h_utstmt[32705];
   long int h_retcode;
   char h_utility[21];
   char h_dsn[55];
   char h_devt[9];
   short int h_space;
   long int h_sysprintrows;           /* Row count SYSIBM.SYSPRINT  */
   long int h_seqno;                  /* SYSPRINT host var seqno    */
   char h_text[255];                  /* SYSPRINT host var text     */
   short int i_text;                  /* SYSPRINT text indicator    */
   volatile SQL TYPE IS RESULT_SET_LOCATOR * sysprint_loc;
   EXEC SQL END DECLARE SECTION;
```

The only parameter that was passed to the thread is a pointer to its thread parameters, which it saves in a local variable as shown in Example 22-19. In our example, it is important for the thread to know its index (or any unique identifier) because it needs to build a unique utility-id.

Example 22-19 Initializing local variables

```
/* Initialize local and thread parameters */
   pthread_parms = (THREAD_PARMS *) arg;  /* Save pointer to params  */
   pthread_parms->o_error = FALSE;
   memset(pthread_parms->o_errmsg,       /* Clear error message     */
          NULLCHAR, sizeof(pthread_parms->o_errmsg));
   pthread_parms->o_utretcode = 0;
   pthread_parms->o_numsysprintlines = 0;
   pthread_parms->p_osysprintlines = NULL;
   sprintf(index, "%02d",                /* Thread index to build ID*/
           pthread_parms->i_thdindex + 1);
```

First, an IDENTIFY has to be issued (as shown in Example 22-20) to establish the task as a user of the DB2 subsystem.

Example 22-20 RRS IDENTIFY

```
/* Issue the RRS IDENTIFY */
   rc = dsnrli((char *) &rrs_funcs[0][0],          /* "IDENTIFY    " */
               (char *) &(pthread_parms->i_ssid[0]), /* Subsystem ID */
               (RIB *) &prib,                      /* RIB pointer    */
               (EIB *) &peib,                      /* EIB pointer    */
               NULL,                               /* Term ecb       */
               NULL,                               /* Startup ecb    */
               (long int *) &rli_rc,               /* Return code    */
               (long int *) &rli_reas);            /* Reason code    */

   if (rc != 0 || rli_rc != 0)            /* If call was not successf*/
   {
     strcpy((pthread_parms->o_errmsg)[0], ERR_THD_IDENTIFY);
     pthread_parms->o_error = TRUE;
   }
```

Next, we do a SIGNON that provides DB2 with a user ID and optionally one or more secondary authorization-ids for the connection. In our case we just use the security environment of the caller.

Example 22-21 RRS SIGNON

```
if (pthread_parms->o_error == FALSE)
   {
     /* The correlation id must be 12 bytes long */
     sprintf(corrid, "RUNSTATP%s  ", index);

     /* Issue the RRS SIGNON */
     rc = dsnrli((unsigned char *) &rrs_funcs[1][0],/* "SIGNON"      */
                 (RRS_CORRID *) corrid,
                 (unsigned char *) rrs_parm_blanks,/* No acctng-token*/
                 (unsigned char *) rrs_parm_blanks,/* Default        */
                 (long int *) &rli_rc,             /* Return code    */
                 (long int *) &rli_reas);          /* Reason code    */

     if (rc != 0 || rli_rc != 0)          /* If call was not successf*/
     {
       /* The SIGNON call was not successful */
       strcpy((pthread_parms->o_errmsg)[0], ERR_THD_SIGNON);
       pthread_parms->o_error = TRUE;
     }
   }
```

The last task for establishing an RRSAF connection is to issue a CREATE THREAD to allocate a plan or package. CREATE THREAD must be issued before any SQL statements can be executed. See Example 22-22.

Example 22-22 RRS CREATE THREAD

```
if (pthread_parms->o_error == FALSE)
   {
     strcpy(planname, RRS_PLAN);
     strcpy(collection, RRS_COLLECTION);

     /* Issue the RRS CREATE THREAD */
     rc = dsnrli((unsigned char *) &rrs_funcs[2][0],/*CREATE THREAD  */
                 (unsigned char *) planname,      /* Plan name       */
                 (unsigned char *) collection,    /* No collection id*/
                 (unsigned char *) rrs_parm_blanks,/* Default reuse p*/
                 (long int *) &rli_rc,            /* Return code     */
                 (long int *) &rli_reas);         /* Reason code     */

     if (rc != 0 || rli_rc != 0)          /* If call was not successf*/
     {
       strcpy((pthread_parms->o_errmsg)[0], ERR_THD_CREATE_THREAD);
       pthread_parms->o_error = TRUE;
     }
   }
```

Now, we are ready to call DSNUTILS, as shown in Example 22-23.

Example 22-23 Calling DSNUTILS

```
if (pthread_parms->o_error == FALSE)
   {
     /* Set up the utility ID first */
     sprintf(h_uid, "RUNSTATP%s", index);
     strcpy(h_restart, "NO");
     sprintf(h_utstmt, "RUNSTATS TABLESPACE %s.%s",
             pthread_parms->i_dbname, pthread_parms->i_tsname);
     strcpy(h_utility, "RUNSTATS TABLESPACE");
     strcpy(h_dsn, "");
     strcpy(h_devt, "");
     h_space = 0;

     /* Call DSNUTILS */
     EXEC SQL CALL SYSPROC.DSNUTILS
                  (:h_uid, :h_restart, :h_utstmt,
                   :h_retcode, :h_utility,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space,
                   :h_dsn, :h_devt, :h_space);
     if (SQLCODE != 466)
     {
       /* An error occured while calling DSNUTILS */
       /* Get error message */
       sql_error(ERR_CALL_UTILS, &(pthread_parms->o_errmsg),
                 &dummy_rc, &sqlca);
       pthread_parms->o_error = TRUE;
     }
     else
       pthread_parms->o_utretcode = h_retcode;
   }
```

If the call to DSNUTILS is successful, we need to retrieve the SYSPRINT lines, allocate memory for them, and pass them back using a field in the thread parameter variable. We have to count the number of SYSPRINT lines to know how many lines we have to allocate dynamically. It is the responsibility of the primary thread to free the memory after reading the SYSPRINT lines and inserting them into the output table. See Example 22-24.

Example 22-24 Counting the SYSPRINT lines

```
if (pthread_parms->o_error == FALSE)
   {
     /* The call to DSNUTILS was successful        */
     /* Now we need to retrieve the result rows    */
     /* and pass them back                         */
     /* Check how many rows are in the result set  */
     /* before retrieving it to allocate the array */
     /* of sysprint_lines                          */
     EXEC SQL SELECT COUNT(*) INTO :h_sysprintrows
                FROM SYSIBM.SYSPRINT;
```

```
  if (SQLCODE != 0)
  {
    /* Get error message */
    sql_error(ERR_COUNT_UTILS_ROWS, &(pthread_parms->o_errmsg),
              &dummy_rc, &sqlca);
    pthread_parms->o_error = TRUE;
  }
}

if (pthread_parms->o_error == FALSE)
{
  /* Allocate sysprint buffer if number of rows */
  /* is greater than 0                          */
  pthread_parms->o_numsysprintlines = h_sysprintrows;
  if (h_sysprintrows > 0)
  {
    if ((pthread_parms->p_osysprintlines = (SYSPRINT_LINE *)
         malloc(h_sysprintrows * sizeof(SYSPRINT_LINE))) == NULL)
    {
      /* Required storage could not be allocated */
      strcpy((pthread_parms->o_errmsg)[0], ERR_MALLOC_SYSPRINT);
      pthread_parms->o_error = TRUE;
    }
  }
}

if (pthread_parms->o_error == FALSE)
{
  /* The storage could be allocated, now we read */
  /* out the lines from SYSIBM.SYSPRINT          */
  /* Associate result set locator                */
  EXEC SQL ASSOCIATE LOCATOR (:sysprint_loc)
               WITH PROCEDURE SYSPROC.DSNUTILS;
  if (SQLCODE != 0)
  {
    /* Get error message */
    sql_error(ERR_ASSOC_SYSPRINT, &(pthread_parms->o_errmsg),
              &dummy_rc, &sqlca);
    pthread_parms->o_error = TRUE;
  }
  else
  {
    /* Try to allocate cursor with result set locator */
    EXEC SQL ALLOCATE SYSPRINT_CSR CURSOR
       FOR RESULT SET :sysprint_loc;

    if (SQLCODE != 0)
    {
      /* Get error message */
      sql_error(ERR_ALLOC_SYSPRINT, &(pthread_parms->o_errmsg),
                &dummy_rc, &sqlca);
      pthread_parms->o_error = TRUE;
    }
  }
}

if (pthread_parms->o_error == FALSE)
{
  /* Fetch all rows */
  for (i = 0; i < h_sysprintrows; i++)
```

```
    {
      memset(h_text, NULLCHAR, sizeof(h_text));

      EXEC SQL FETCH SYSPRINT_CSR
               INTO :h_seqno, :h_text:i_text;

      if (SQLCODE != 0)
      {
        /* Get error message */
        sql_error(ERR_FETCH_SYSPRINT, &(pthread_parms->o_errmsg),
                  &dummy_rc, &sqlca);
        pthread_parms->o_error = TRUE;
        break;
      }
      else
      {
        /* Save output line */
        (pthread_parms->p_osysprintlines + i)->seqno = h_seqno;
        (pthread_parms->p_osysprintlines + i)->ind = i_text;
        strcpy((pthread_parms->p_osysprintlines + i)->text, h_text);
      }
    }
  }

  if (pthread_parms->o_error == FALSE)
  {
    EXEC SQL CLOSE SYSPRINT_CSR;

    if (SQLCODE != 0)
    {
      /* Get error message */
      sql_error(ERR_CLOSE_SYSPRINT, &(pthread_parms->o_errmsg),
                &dummy_rc, &sqlca);
      pthread_parms->o_error = TRUE;
    }
  }
```

Finally, we do an orderly disconnect from the subsystem and leave the utility thread as shown in Example 24-25.

Example 22-25 Disconnecting from the subsystem

```
if (pthread_parms->o_error == TRUE)
    EXEC SQL ROLLBACK;
  else
  {
    EXEC SQL COMMIT;

    if (SQLCODE != 0)
    {
      sql_error(ERR_THD_COMMIT, &(pthread_parms->o_errmsg),
                &dummy_rc, &sqlca);
      pthread_parms->o_error = TRUE;
    }
  }

  /* Issue the RRS TERMINATE THREAD */
  if (!pthread_parms->o_error)
  {
    rc = dsnrli((unsigned char *) &rrs_funcs[3][0],/* TERMINATE THRE*/
```

```
                (long int *) &rli_rc,              /* Return code   */
                (long int *) &rli_reas);           /* Reason code   */
    if (rc != 0 || rli_rc != 0)
    {
      /* The TERMINATE THREAD call was not successful */
      pthread_parms->o_error = TRUE;
      strcpy((pthread_parms->o_errmsg)[0], ERR_THD_TERMINATE_THREAD);
    }
  }

  /* Issue the RRS TERMINATE IDENTIFY */
  if (!pthread_parms->o_error)
  {
    rc = dsnrli((unsigned char *) &rrs_funcs[4][0],/* TERM IDENTIFY */
                (long int *) &rli_rc,              /* Return code   */
                (long int *) &rli_reas);           /* Reason code   */
    if (rc != 0 || rli_rc != 0)
    {
      /* The TERMINATE IDENTIFY call was not successful */
      pthread_parms->o_error = TRUE;
      strcpy((pthread_parms->o_errmsg)[0], ERR_THD_TERMINATE_IDENTIFY);
    }

  }

  /* Leave the utility thread */
  pthread_exit(NULL);
}
```

22.4 Compiling the stored procedure

We compile a multi-threaded stored procedure like any other C language stored procedure, with one exception: Because we explicitly include sqlca.h, we have to include DSN.SDSNC.H in our search path for include files, as shown in Example 22-26.

Example 22-26 Including DSN.SDSNC.H in the search path

```
//RUNSTATP JOB (999,POK),'C P/C/L/B',CLASS=K,MSGCLASS=H,
// NOTIFY=PAOLOR8,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=SC63,L=9999
// JCLLIB ORDER=(DB2V710G.PROCLIB)
//*
//JOBLIB  DD  DSN=DB2V710G.SDSNEXIT,DISP=SHR
//        DD  DSN=DB2G7.SDSNLOAD,DISP=SHR
//        DD  DSN=CEE.SCEERUN,DISP=SHR
//*-------------------------------------------------------------
//* STEP 01: PRE-COMPILE, COMPILE, LINK-EDIT RUNSTATP
//*          STORED PROCEDURE
//*-------------------------------------------------------------
//STEP01   EXEC PROC=DSNHCPP,MEM=RUNSTATP,
//         PARM.PC=('HOST(C)',CCSID(1047)),
//         PARM.COMP='/OPTFILE(DD:COPT)'
//PC.DBRMLIB   DD DSN=SG247083.DEVL.DBRM(RUNSTATP),
//             DISP=SHR
//PC.SYSLIB    DD DSN=SG247083.PROD.SOURCE,
//             DISP=SHR
//PC.SYSIN     DD DSN=SG247083.PROD.SOURCE(RUNSTATP),
//             DISP=SHR
```

```
//COMP.COPT    DD  *
   SEARCH('CEE.SCEEH.H')
   SEARCH('CEE.SCEEH.SYS.H')
   SEARCH('DB2G7.SDSNC.H')
   MARGINS(1,72)
   SOURCE
   LIST
   RENT
   DEF(DEBUG)
/*
//LKED.SYSLMOD DD DSN=SG247083.DEVL.LOAD(RUNSTATP),
//             DISP=SHR
//LKED.SYSIN   DD *
   ORDER CEESTART,RUNSTATP
   INCLUDE SYSLIB(DSNRLI)
   INCLUDE SYSLIB(DSNTIAR)
   ENTRY CEESTART
   MODE AMODE(31),RMODE(ANY)
   NAME RUNSTATP(R)
/*
//*-------------------------------------------------------
//* STEP 02: BIND RUNSTATP STORED PROCEDURE
//*-------------------------------------------------------
//STEP02   EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *
   DSN SYSTEM(DB2G)
   BIND PACKAGE(DEVL7083) -
        MEMBER(RUNSTATP) ACT(REP) ISO(CS) ENCODING(EBCDIC) -
        OWNER(DEVL7083) LIBRARY('SG247083.DEVL.DBRM')
   END
/*
```

22.5 Authorization issues - Best practices

Sometimes it is not desirable to impose special authorization requirements onto the user ID that calls the stored procedure. To illustrate this problem, let us take a look at the following example:

A user with the user ID TEST7083 wants to invoke the stored procedure RUNSTATP in order to collect statistics for the following two table spaces that are part of the DB2 catalog DSNDB06:

```
SYSPKAGE, SYSPLAN
```

Let us further assume that the user ID TEST7083 has almost no DB2-related privileges but nevertheless seeks to employ the Java program that is partially displayed in Example 22-4. This client will be used to establish a JBDC connection to DB2, insert the two table spaces and database names into a GLOBAL TEMPORARY TABLE, and finally to CALL the procedure. The JDBC connection is opened with the user ID TEST7083.

To successfully invoke the procedure RUNSTATP, at least the following two GRANT statements have to be issued:

```
GRANT ALL PRIVILEGES ON DEVL7083.RSP_TBL TO TEST7083
```

This statement allows the user to INSERT the table space and database names into the temporary input table.

```
GRANT EXECUTE ON PROCEDURE DEVL7083.RUNSTATP TO TEST7083
```

This statement allows the user ID TEST7083 to eventually CALL the stored procedure from within the Java program.

An interesting observation now is that the subthread that establishes an RRSAF connection to DB2 in dsnutils_thread(), depends on the SECURITY option defined during the procedure registration. This SECURITY option indicates the nature of the external security environment created for that procedure, at execution time by DB2. Any external activity performed by that procedure is therefore governed by this created security environment. Since SECURITY USER was defined in the sample, the ID used to establish the security environment is the value of the USER special register when the routine is called, that is, TEST7083.

The subthread creation is one of those governed external activities and therefore inherits this security environment. The RRSAF attachment, performed by that subthread, is another governed external activity. Thus the new DB2 connection thread has this security ID (special register USER) as its primary authID, by default. This DB2 connection thread is separate and distinct from the DB2 thread implicitly provided by DB2 for the stored procedure.

Every action that the subthread performs (authority-wise) stems from the external security environment provided to the routine. As an example, SQL statements are executed on the separate DB2 connection thread, which inherited the primary authID from the external security environment, that is, TEST7083. Therefore, the following additional privilege has to be granted to TEST7083*:*

```
GRANT EXECUTE ON PACKAGE DEVL7083.RUNSTATP TO TEST7083
```

This statement allows the primary authID TEST7083 to execute an SQL statement in the dsnutils_thread() function.

We now run the Java program with the following command:

```
java RunstatPDriver DBALIAS TEST7083 TESTPWD
```

and obtain the output shown in Example 22-27.

Example 22-27 Unsuccessful call to DSNUTILS

```
Unsuccessful DSNUTILS CALL
1DSNU000I    009 02:24:07.28 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = RUNSTATP01
 DSNU1044I   009 02:24:07.40 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
0DSNU050I    009 02:24:07.40 DSNUGUTC -  RUNSTATS TABLESPACE DSNDB06.SYSPKAGE
 DSNU060I  ) 009 02:24:07.41 DSNUGMAP - USER TEST7083 NOT AUTHORIZED FOR RUNSTATS UTILITY
ON DATABASE DSNDB06
 DSNU606I  ) 009 02:24:07.41 DSNUSVAL - USER NOT AUTHORIZED TO ACCESS TABLESPACE
DSNDB06.SYSPKAGE
 DSNU012I    009 02:24:07.41 DSNUGBAC - UTILITY EXECUTION TERMINATED, HIGHEST RETURN
CODE=12
1DSNU000I    009 02:24:06.85 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = RUNSTATP02
 DSNU1044I   009 02:24:06.99 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
0DSNU050I    009 02:24:07.07 DSNUGUTC -  RUNSTATS TABLESPACE DSNDB06.SYSPLAN
 DSNU060I  ) 009 02:24:07.11 DSNUGMAP - USER TEST7083 NOT AUTHORIZED FOR RUNSTATS UTILITY
ON DATABASE DSNDB06
 DSNU606I  ) 009 02:24:07.11 DSNUSVAL - USER NOT AUTHORIZED TO ACCESS TABLESPACE
DSNDB06.SYSPLAN
 DSNU012I    009 02:24:07.15 DSNUGBAC - UTILITY EXECUTION TERMINATED, HIGHEST RETURN
CODE=12
```

```
Internal Error DB2RUNSTATPException: Utility execution failed. Highest DSNUTILS return
code: 12
```

The call to DSNUTILS failed and as the result set indicates the authID TEST7083 did not hold the appropriate privileges.

In this scenario, the subthread with its own independent DB2 connection thread made an SQL CALL to the procedure DSNUTILS. The calling characteristics here also govern the operational characteristics of DSNUTILS, which creates another internal dependency on the TEST7083 authID.

To solve this issue, further privileges have to be granted to the authID TEST7083:

```
GRANT STATS ON DATABASE DSNDB06 TO TEST7083
```

Granting these additional privileges might not be desirable, because the user could directly interact with sensitive database objects. Generally it is a good idea to introduce a layer that interacts with the DB2 objects in a consistent way, such as a stored procedure, instead of allowing users to directly access and modify them.

Furthermore it is always good practice to implement a certain division of security, that is, it should be distinguished between the privilege a caller needs to utilize the RUNSTATP procedure, and the privileges needed by this procedure to perform its specific internal tasks.

SECURITY DEFINER

The approach described in this section transforms the above described *runtime* problem into a *configuration* issue.

Instead of using the SECURITY USER option in the signature, the procedure can be defined with SECURITY DEFINER when registered to DB2. This is shown in Example 22-28.

Example 22-28 RUNSTATP definition with SECURITY DEFINER

```
CREATE PROCEDURE DEVL7083.RUNSTATP
  ( OUT UTILITIES_EX INTEGER
  , OUT HIGHEST_RETCODE INTEGER
  , OUT RETCODE INTEGER
  , OUT MESSAGE VARCHAR(1331) CCSID EBCDIC)
  RESULT SETS 1
  EXTERNAL NAME RUNSTATP
  LANGUAGE C
  PARAMETER STYLE GENERAL WITH NULLS
  MODIFIES SQL DATA
  WLM ENVIRONMENT WLMENVR
  STAY RESIDENT NO
  COLLID DEVL7083
  PROGRAM TYPE MAIN
  RUN OPTIONS 'TRAP(OFF),STACK(,,ANY,),POSIX(ON)'
  COMMIT ON RETURN NO
  SECURITY DEFINER
  ASUTIME NO LIMIT;
```

SECURITY DEFINER implies that the external security environment is established using the ID that registered the routine to DB2. In our example the creator of the stored procedure has the ID DEVL7083. This is now the primary authID that the subthreads as well as the separate DB2 connection threads security environment is based on.

For a successful CALL, it has to be ensured that the procedure DEFINER (DEVL7083) has been granted sufficient authorities to call and use DSNUTILS (*configuration-time*).

The nice thing now is, that no special authorization requirements are imposed on the calling user ID (*runtime*), except the privileges needed to insert rows into the temporary table and to CALL the procedure from within the Java program. Issue the following GRANT statement to allow any user ID to successfully call the RUNSTATP procedure.

```
GRANT ALL PRIVILEGES ON DEVL7083.RSP_TBL TO user-ID

GRANT EXECUTE ON PROCEDURE DEVL7083.RUNSTATP TO user-ID
```

As illustrated in Example 22-29, the wrapped CALL to DSNUTILS now completes successfully using the user ID TEST7083.

Example 22-29 Successful call to DSNUTILS

```
1DSNU000I    009 02:25:50.40 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = RUNSTATP01
 DSNU1044I   009 02:25:50.52 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
0DSNU050I    009 02:25:50.52 DSNUGUTC -  RUNSTATS TABLESPACE DSNDB06.SYSPKAGE
 DSNU610I  ) 009 02:25:50.72 DSNUSUTP - SYSTABLEPART CATALOG UPDATE FOR DSNDB06.SYSPKAGE
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.72 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPACKLIST
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.72 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPACKAGE
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.73 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPACKSTMT
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.73 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPACKDEP
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.73 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPACKAUTH
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.73 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPLSYSTEM
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.73 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPKSYSTEM
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.73 DSNUSUTS - SYSTABLESPACE CATALOG UPDATE FOR DSNDB06.SYSPKAGE
SUCCESSFUL
 DSNU620I  ) 009 02:25:50.74 DSNUSEF2 - RUNSTATS CATALOG TIMESTAMP =
2008-01-09-02.25.50.533937
 DSNU010I    009 02:25:50.76 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
1DSNU000I    009 02:25:50.04 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = RUNSTATP02
 DSNU1044I   009 02:25:50.15 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
0DSNU050I    009 02:25:50.15 DSNUGUTC -  RUNSTATS TABLESPACE DSNDB06.SYSPLAN
 DSNU610I  ) 009 02:25:50.24 DSNUSUTP - SYSTABLEPART CATALOG UPDATE FOR DSNDB06.SYSPLAN
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.26 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPLANDEP
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.26 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPLAN
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.27 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSDBRM
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.27 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSSTMT
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.27 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR SYSIBM.SYSPLANAUTH
SUCCESSFUL
 DSNU610I  ) 009 02:25:50.27 DSNUSUTS - SYSTABLESPACE CATALOG UPDATE FOR DSNDB06.SYSPLAN
SUCCESSFUL
 DSNU620I  ) 009 02:25:50.27 DSNUSEF2 - RUNSTATS CATALOG TIMESTAMP =
2008-01-09-02.25.50.202955
```

```
 DSNU010I    009 02:25:50.29 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
DB2Runstats successful.
```

AUTH SIGNON

Although the SECURITY DEFINER approach moved the USER privilege set out of the internal business of the subthreads, it is very static in terms of which ID is used to establish the security environment. A more dynamic approach is described in this section.

Be aware that this approach requires the procedure to run APF-authorized.

RRSAF allows to associate a different primary authID to the established DB2 connection thread. This is done by replacing the SIGNON call with an AUTH SIGNON call. Example 22-30 shows the new AUTH SIGNON call, which is issued after the INDENTIFY and before the CREATE THREAD call.

Example 22-30 RRSAF function calls

```
unsigned char rrs_funcs[6][19] = { "IDENTIFY          ",
                                   "AUTH SIGNON       ",
                                   "CREATE THREAD     ",
                                   "TERMINATE THREAD  ",
                                   "TERMINATE IDENTIFY",
                                   "TRANSLATE         "};
```

We introduce three new parameters for the AUTH SIGNON call, as can be seen in Example 22-31. After the RRSAF connection has been successfully established, a new primary authorization-ID, DEVL7083, is associated with this separate DB2 connection thread. We do not provide a secondary authID because it is not required here.

Example 22-31 AUTH SIGNON call

```
/* Issue the RRS AUTH SIGNON */
     rc = dsnrli((unsigned char *) &rrs_funcs[1][0],/* "AUTH SIGNON" */
                 (RRS_CORRID *) corrid,
                 (unsigned char *) rrs_parm_blanks,/* No acctng-token*/
                 (unsigned char *) rrs_parm_blanks,/* Default        */
                 "DEVL7083 ",                     /* Prim. authid   */
                 0,                               /* ACEE address   */
                 " ",                             /* Sec. authid    */
                 (long int *) &rli_rc,            /* Return code    */
                 (long int *) &rli_reas);         /* Reason code    */
```

In this example the new primary authID is a hard coded string value. However, a variable can be used for a more dynamic solution.

Due to this user ID switch during the RRSAF connect, the CALL to DSNUTILS completes successfully, assuming the new primary authID holds the required privileges.

Restriction: You can basically switch to any valid primary authorization-ID, the only requirement is that your procedure has to run APF authorized.

22.6 Improvements

Creating a thread for every utility is not a good design. In a more sophisticated stored procedure, we would initially start up a number of threads that run utilities in a loop to avoid

the overhead of too many secondary threads. Also, you will run into concurrency issues when too many utilities compete for the same buffer pools and catalog tables, hence, they have to be sorted and utility execution has to be serialized by various criteria.

If you reuse threads, you have to implement event-driven synchronization between the primary scheduler thread and the secondary threads using condition variables, and mutex objects. You can use the corresponding pthread_mutex and pthread_cond functions for that.

Examples of how to use mutex objects and condition variables can be found in *z/OS C/C++ Programming Guide*, SC09-4765-08 and *z/OS C/C++ Run-Time Library Reference*, SA22-7821-09.

22.7 Common design problems using multiple threads

The most common problem that needs to be avoided when a stored procedure uses multiple threads is deadlocks.

As an example of a deadlock that the database manager does not detect, consider a stored procedure that has two threads, both of which access a common data structure. To avoid problems where both contexts change the data structure simultaneously, the data structure is protected by a semaphore. The contexts look like Example 22-32.

Example 22-32 Contexts for semaphore

```
context 1
SELECT * FROM TAB1 FOR UPDATE....
UPDATE TAB1 SET....
get semaphore
access data structure
release semaphore
COMMIT

context 2
get semaphore
access data structure
SELECT * FROM TAB1...
release semaphore
COMMIT
```

Suppose the first context successfully executes the SELECT and UPDATE statements, while the second context gets the semaphore and accesses the data structure. The first context now tries to get the semaphore, but it cannot because the second context is holding the semaphore. The second context now attempts to read a row from table TAB1, but it stops on a database lock held by the first context. The application is now in a state where context 1 cannot finish before context 2 is done, and context 2 is waiting for context 1 to finish. The application is deadlocked, but because the database manager does not know about the semaphore dependency, neither context will be rolled back. The unresolved dependency leaves the application suspended. You can avoid the deadlock that occurred for the previous example in several ways:

- Release all locks held before obtaining the semaphore.

  Change the code for context 1 to perform a commit before it gets the semaphore.
- Do not code SQL statements inside a section protected by semaphores.

  Change the code for context 2 to release the semaphore before doing the SELECT.
- Code all SQL statements within semaphores.

Change the code for context 1 to obtain the semaphore before running the SELECT statement. While this technique will work, it is not highly recommended because the semaphores will serialize access to the database manager, which potentially negates the benefits of using multiple threads.

23

Accessing CICS and IMS

You may have legacy applications that access data that resides in non-DB2 resources, such as IMS databases and VSAM files. Often these applications run under transaction managers such as CICS or IMS. Rather than rewrite entire legacy systems, you would like to be able to access existing applications from newer DB2 applications, or access DB2 data from your existing applications. Stored procedures can help you do just that.

For example, you may have a CICS application that accesses data from a VSAM file. If you are developing DB2 applications that need to access data from that VSAM file, and you do not have the resources to migrate that data from VSAM to DB2 at this time, you can access the VSAM file from a DB2 stored procedure using the external CICS interface or DB2-supplied stored procedure DSNACICS to execute an existing CICS program. You can also access the VSAM file from within your stored procedure, but that would require you to define a DD statement for that file within your application environment. Then, if you alter the stored procedure to execute in a different environment, you need to change the JCL for the old and new environments.

If you have IMS data that you would like to access from a DB2 stored procedure, you can use the ODBA interface to code DLI calls in your stored procedure or use DB2-supplied stored procedure DSNAIMS to execute a program running under the IMS transaction manager.

You can also call DB2 stored procedures from CICS and IMS applications. For example, you may have a CICS application that accesses data from a VSAM file. If you now need to access DB2 data from that application, you can call a DB2 stored procedure to access the DB2 data. The SQL CALLs in CICS and IMS applications do not differ from SQL CALLs in other environments, such as batch, but the program preparation steps differ somewhat. We now discuss how to code SQL calls in CICS and IMS applications, and give an overview of how to prepare those applications.

Your stored procedures that access CICS and IMS can also be debugged just like any other stored procedure. See Chapter 16, “Debugging” on page 313, and Chapter 28, “Tools for debugging DB2 stored procedures” on page 735 for more details.

You need to be familiar with CICS or IMS to understand the topics discussed here. We make no attempt to cover basic CICS and IMS concepts. We focus on the interfaces from DB2

instead. There have been some enhancements in this area, including the introduction of DB2-supplied stored procedures to access non-DB2 data.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions can be found in Appendix B, “Additional material” on page 887.

Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 24 to decide what components are applicable to your environment.

This chapter contains the following:

- Accessing CICS systems from DB2 stored procedures
- Accessing IMS databases from DB2 stored procedures
- Accessing DB2 stored procedures from CICS
- Accessing DB2 stored procedures from IMS

23.1 Accessing CICS systems from DB2 stored procedures

There are two alternatives for accessing CICS systems from a DB2 stored procedure: the external CICS interface (EXCI) and stored procedure DSNACICS. Which alternative you choose is based on the expertise of your development staff. If your developers are experienced CICS programmers, you should be most comfortable using EXCI. If your developers are experienced DB2 programmers with little CICS experience, you should be most comfortable with DSNACICS.

23.1.1 Accessing CICS systems through EXCI

The external CICS interface (EXCI) is an application programming interface that enables a non-CICS program (a client program) running in z/OS to call a program (a server program) running in a CICS region, and to pass and receive data by means of a communications area. The CICS application program is invoked as if linked to by another CICS application program.

EXCI is provided in two forms:

- The EXCI CALL interface and the EXEC CICS interface. The EXCI CALL interface requires you to code a series of commands to allocate and open sessions to a CICS region, issue DPL requests from the non-CICS systems, and to close and deallocate the sessions upon completion of the DPL requests.
- The EXEC CICS interface uses the EXEC CICS LINK PROGRAM command to perform all of the functions of the EXCI CALL command set.

We use the EXEC CICS interface in our case study because it is simpler to code and more frequently used. For more details on each of the interfaces, refer to *CICS Transaction Server for z/OS V3.1 CICS External Interfaces Guide*, SC34-6449-03.

In our case study we make the assumption that a legacy CICS application (COBOL program EMPEXC2C) exists which retrieves the department name for a given department number. The department information is stored in a VSAM file that is in the same format as the sample DEPT table in DB2. Since the same VSAM file is used for both the CICS and IMS case studies, sample JCL for defining the VSAM file is provided in job IMS05.txt in the additional materials and is shown in Example 23-13 on page 480. A new DB2 stored procedure (COBOL program EMPEXC1C) is being developed to return a results set of all employees for a given department number, with the name of the department included for each employee. Stored procedure EMPEXC1C is a version of EMPRSETC, the COBOL results set stored procedure, with an EXCI call to EMPEXC2C replacing the SELECT from the DEPT table.

We executed the following steps to set up the environment and develop our EXCI test case:

- Preparing CICS and WLM environments for using EXCI
- Coding stored procedures to invoke EXCI
- Preparing a stored procedure to use EXCI

Details of each step are discussed in the following sections.

Preparing CICS and WLM for EXCI

Prior to executing a stored procedure that invokes EXCI, you must define the following resources on the resource definition screen in the CICS region where the CICS program will execute: connections, sessions, transaction, program, and any files accessed. You can perform the resource definition (RDO) process using the CEDA transaction. We have included access to a VSAM file. The VSAM file is defined to CICS via the RDO since we access the file from the CICS transaction. Therefore, there is no need to add a DD statement for the VSAM file to the startup JCL for the address space.

We defined all the RDO entries for our test case in group SG247083, which is a copy of the resource group DFH$EXCI supplied with CICS. All of the RDO definitions that are used by EXCI must be defined in the same group. RDO definitions can be maintained by using the CEDA transaction in CICS. Here are the steps we followed to define the resources to CICS. We used a CICS Transaction Server V3.1 system for out testing.

CICS resource definitions

1. Connections definition - Issue the following command to define a connection:

   ```
   CEDA DEF GROUP(SG247083) CON
   ```

   The required fields and the values we chose were as follows. For each field you can use an abbreviation by specifying only those characters shown in capital letters. The values we chose for each field are shown in bold.

   – CONnection - **XCTG**
   – Group - **SG247083** (this is carried over from the CEDA command). Once the first resource has been defined, the group is automatically defined.
   – DEscription - This is optional.
   – ACcessmethod - **IRc** (Abbreviation: IR)
   – PRotocol - **Exci** (Abbreviation: E)
   – Conntype - **Generic** (Abbreviation: G)

2. Sessions definition - Issue the following command to define a session:

   ```
   CEDA DEF GROUP(SG247083) SES
   ```

   The required fields and the values we chose were as follows:

   – Sessions - **XCTGSESS**
   – Connection - **XCTG**, which refers to the connections definition.
   – Protocol - **Exci** (Abbreviation: E)
   – RECEIVECount - **4** (should be some non-zero number)

3. Program definition - You need one entry for the CICS program called by the stored procedure. Issue the following command to define program EMPEXC2C:

   ```
   CEDA DEF GROUP(SG247083) PRO
   ```

   The required fields and the values we chose were as follows:

   – PROGram - **EMPEXC2C**
   – Language - **COBol** (Abbreviation: CO)

4. Transaction definition - You need one entry for the CICS transaction ID that is associated with the CICS program to be executed. The transaction definition refers to CICS program DFHMIRS, which is a stub for the EXCI call. Program DFHMIRS will then execute the CICS program you specify in the CICS LINK statement in your stored procedure. Issue the following command to define transaction DPT1:

   ```
   CEDA DEF GROUP(SG247083) TR
   ```

   The required fields and the values we chose were as follows:

   – TRANSaction - **DPT1**
   – PROGram - **DFHMIRS**
   – PROFile - **DFHCICSA**

   You need to specify a profile with parameter INBFMH=ALL as in DFHCICSA, otherwise the transactions fail with abend code AXFQ.

5. VSAM file definition - Issue the following command to define VSAM file Sg247083.DEPT with a DDname of DEPTNAME:

   ```
   CEDA DEF GROUP(SG247083) FI
   ```

   – File - **DEPTNAME**

– DSNAme - **SG247083.DEPT.CL**, which is the VSAM cluster name

6. Install the group **SG247083**:

```
CEDA I GROUP(SG247083)
```

7. ISC parameter - Verify that SIT parameter ISC is set to YES.
8. RRMS parameter - Verify that SIT parameter RRMS is set to YES. CICS supports MVS resource recovery services (RRS) in applications that use the external CICS interface.

Remember that changes made to program EMPEXC2C only become active in CICS after executing the following command in CICS to pull in the latest version of program EMPEXC2C:

```
CEMT SET PROG(EMPEXC2C) NEWCOPY
```

The results of the CEMT command are shown in Example 23-1.

Example 23-1 CEMT command used to refresh a CICS program

```
SET PROG(EMPEXC2C) NE
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Prog(EMPEXC2C) Leng(0000005392) Cob Pro Ena Pri     Ced     NORMAL
    Res(000) Use(0000000001) Bel Uex Ful Qua Cic
```

WLM definitions

We recommend that you define a separate WLM application environment for your EXCI transactions to minimize the impact that problems with CICS systems can have on your stored procedures. We chose to name the environment DB9AEXCI. The load library that contains the EXCI stub program DFHMIRS needs to be included in the startup JCL for the WLM procedure. The name of the load library typically ends in SDFHEXCI.

Coding a stored procedure to use EXCI

Stored procedure EMPEXC1C includes an EXEC CICS LINK statement to invoke the new transaction defined in the RDO entry as described in the section above. Example 23-2 shows a sample LINK statement.

Example 23-2 Sample EXCI call from stored procedure to CICS

```
EXEC CICS LINK PROGRAM ('EMPEXC2C')
               TRANSID ('DPT1')
               APPLID  ('SCSCPAPB')
               COMMAREA(WS-COMM-AREA)
               LENGTH  (WS-COMM-LEN)
               RETCODE (EXCI-EXEC-RETURN-CODE)
               SYNCONRETURN
END-EXEC.
```

When the LINK statement is executed, CICS loads program DFHMIRS, the EXCI mirror program, which in turn will load the program that is passed in the PROGRAM field of the LINK statement, which is EMPEXC2C in our case. This program then reads the commarea that is passed, uses the DEPTNO field in the commarea, and reads file DEPTNAME to obtain the department name, which is then passed back in the commarea. The fields WS-COMM-AREA and WS-COMM-LEN represent the commarea, and the length of the commarea that is passed from the stored procedure to CICS program EMPEXC2C.

The field EXCI-EXEC-RETURN-CODE contains five diagnostic fields that are passed back to the calling program. An example of the definition of the diagnostic fields in a COBOL program is shown in Example 23-3.

Example 23-3 Diagnostic field definition for stored procedure with EXCI call

```
01 EXCI-EXEC-RETURN-CODE.
  02 EXEC-RESP        PIC 9(8) COMP.
  02 EXEC-RESP2       PIC 9(8) COMP.
  02 EXEC-ABCODE      PIC X(4).
  02 EXEC-MSG-LEN     PIC 9(8) COMP.
  02 EXEC-MSG-PTR     POINTER.
```

The values for each of the diagnostic fields can be found in CICSTS31.CICS.SDFHCOB in member DFHXCRCO.

Preparing a stored procedure to use EXCI

Stored procedures that use EXCI to call CICS programs must include a CICS translation step in the program preparation process. The CICS program needs to be prepared using standard CICS preparation JCL.

Result of running our sample stored procedure

We executed stored procedure EMPEXC1C from IBM Data Studio. The result of the EXCI call to the CICS program EMPEXC2C in the SYSOUT output of the stored procedure address space is listed in Example 23-4.

Example 23-4 Result of EXCI call to EMPEXC2C

```
++ START OF EMPEXC1C STARTING ++
WS-DEPTNO = E11
++ RETURN FROM EXCI:
EXEC-RESP:    00000000
EXEC-RESP2:   00000000
EXEC-ABCODE:
EXEC-MSG-LEN: 00000000
EXEC-MSG-PTR: 0000000000
DEPTNAME:     OPERATIONS
++ SQLCODE AFTER OPEN   = 000000000
++ END OF EMPEXC1C ++
```

23.1.2 Accessing CICS systems through stored procedure DSNACICS

The external CICS interface (EXCI) was introduced as a method to access legacy data in CICS through DB2 stored procedures. EXCI provides the interface customers need to access legacy data, but it does require application developers to understand some CICS syntax (see the LINK statement in “Coding a stored procedure to use EXCI” on page 473), and it does require a CICS translation step in the program preparation JCL for the stored procedure. The CICS transaction invocation stored procedure (DSNACICS) that is available with DB2 for OS/390 Version 6 and later releases can mask much of the CICS interaction, making it easier for developers with minimal CICS expertise to access CICS resources.

DSNACICS is one of the sample stored procedures provided with DB2. DSNACICS gives workstation applications a way to invoke CICS server programs while using TCP/IP as their communication protocol. The workstation applications use TCP/IP and DB2 Connect to connect to a DB2 for z/OS subsystem, and then call DSNACICS to invoke the CICS server programs.

Environment

DSNACICS runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

If you use CICS Transaction Server for OS/390 Version 1 Release 3 or later, you can register your CICS system as a resource manager with recoverable resource management services (RRMS). When you do that, changes to DB2 databases that are made by the program that calls DSNACICS and the CICS server program that DSNACICS invokes are in the same two-phase commit scope. This means that when the calling program performs an SQL COMMIT or ROLLBACK, DB2 and RRS inform CICS about the COMMIT or ROLLBACK.

If the CICS server program that DSNACICS invokes accesses DB2 resources, the server program runs under a separate unit of work from the original unit of work that calls the stored procedure. This means that the CICS server program might deadlock with locks that the client program acquires.

DSNACICS has to be invoked from an APF authorized library.

Authorization

To execute the CALL statement, the owner of the package or plan that contains the CALL statement must have one or more of the following privileges:

- The EXECUTE privilege on stored procedure DSNACICS
- Ownership of the stored procedure
- SYSADM authority

The CICS server program that DSNACICS calls runs under the same user ID as DSNACICS. That user ID depends on the SECURITY parameter that you specify when you define DSNACICS.

The DSNACICS caller also needs authorization from an external security system, such as RACF, to use CICS resources.

Definition

The DDL to create the procedure is located in member DSNTIJSG of the SDSNSAMP library. A sample CREATE PROCEDURE statement is shown in Example 23-5.

Example 23-5 DDL to create sample stored procedure DSNACICS

```
CREATE PROCEDURE SYSPROC.DSNACICS
  ( IN PARM_LEVEL         INTEGER
  , IN PGM_NAME           CHAR(8)
  , IN CICS_APPLID        CHAR(8)
  , IN CICS_LEVEL         INTEGER
  , IN CONNECT_TYPE       CHAR(8)
  , IN NETNAME            CHAR(8)
  , IN MIRROR_TRANS       CHAR(4)
  , INOUT COMMAREA        VARCHAR(32704)
  , IN COMMAREA_TOTAL_LEN INTEGER
  , IN SYNC_OPTS          INTEGER
  , OUT RETURN_CODE       INTEGER
  , OUT MSG_AREA          VARCHAR(500) )
  PARAMETER CCSID EBCDIC
  EXTERNAL NAME DSNACICS
  LANGUAGE ASSEMBLE
  WLM ENVIRONMENT DB9AEXCI
  COLLID SYSPROC
  RUN OPTIONS 'TRAP(OFF)'
  PROGRAM TYPE SUB
  NO SQL
  ASUTIME NO LIMIT
  STAY RESIDENT YES
```

```
  COMMIT ON RETURN NO
  PARAMETER STYLE GENERAL WITH NULLS
  RESULT SETS 0
  SECURITY USER;
```

DSNACICS is provided in executable form as part of the DB2 installation process.

A DB2 stored procedure or application program can issue an SQL CALL to DSNACICS in place of an EXCI call to access CICS. For our test case, we created stored procedure EMPEXC3C, which is a copy of EMPEXC1C (which issues an EXCI call as described in “Coding a stored procedure to use EXCI” on page 473). EMPEXC3C replaces the EXEC CICS LINK statement with an SQL CALL to DSNACICS. The call statement that was used in our test case is shown in Example 23-6. The parameters for the CALL statement are described in detail in Appendix B of *DB2 Version 9.1 for z/OS Administration Guide,* SC18-9840.

Example 23-6 Sample CALL to DSNACICS

```
EXEC SQL
  CALL SYSPROC.DSNACICS
    (:PARM-LEVEL  :IND-PARM-LEVEL,
    :PGM-NAME     :IND-PGM-NAME,
    :CICS-APPLID  :IND-CICS-APPLID,
    :CICS-LEVEL   :IND-CICS-LEVEL,
    :CONNECT-TYPE :IND-CONNECT-TYPE,
    :NETNAME      :IND-NETNAME,
    :MIRROR-TRANS :IND-MIRROR-TRANS,
    :COMM-AREA    :IND-COMM-AREA,
    :COMM-LEN     :IND-COMM-LEN,
    :SYNC-OPTS    :IND-SYNC-OPTS,
    :RET-CODE     :IND-RET-CODE,
    :MSG-AREA     :IND-MSG-AREA)
END-EXEC.
```

We now describe the parameters that are relevant for our test case. These are the same parameters that are supplied on the CICS LINK statement for the EXCI test case:

- PGM-NAME is an input parameter that contains the name of the CICS program to be invoked by the EXCI mirror program DFHMIRS. In our case the PGM-NAME is EMPEXC2C.
- CICS-APPLID is an input parameter that contains the name of the CICS region where the program identified in PGM-NAME will execute. In our case, the CICS-APPLID is SCSCSAR3.
- CONNECT-TYPE is an input parameter that specifies whether the CICS connection is generic or specific. This must match the value in the Conntype parameter of the Connections RDO entry. In our case, the CONNECT-TYPE is GENERIC.
- MIRROR-TRANS is an input parameter that names the CICS transaction ID that invokes program DFHMIRS. In our case the MIRROR-TRANS is DPT1.
- COMM-AREA contains the commarea that will be passed to the CICS program.
- COMM-LEN specifies the length of the commarea to be passed.
- RET-CODE is an output parameter that contains the return code from the DSNACICS call. The return code will either be 0 for successful or 12 for failure. If the return code is 12, the error is described in the MSG-AREA parameter.
- MSG-AREA is an output parameter that contains any error messages from the call.

- The SYNC-OPTS parameter is a functional difference between EXCI LINK and DSNACICS. The option tells CICS whether it should drive SYNCONRETURN (commit) processing at the end of the CICS transaction:
  - If SYNC-OPTS = 1, then the CALLING application controls when the two-phase commit protocol will be driven, and any DB2 updates along with CICS updates will be handled in the same commit scope.
  - If SYNC-OPTS = 2, then CICS will commit at completion of the CICS transactions, and it will not be handled in the same commit scope as any DB2 updates (or other RRS controlled resources).

DSNACICX user exit

When a stored procedure calls DSNACICS, stored procedure DSNACICS always calls user exit DSNACICX first to change some of the parameter values before control is passed to CICS. DSNACICX is a sample exit that is provided in the SDSNSAMP library in both Assembler (member DSNASCIX) and COBOL (member DSNASCIO) formats. You have the option to modify the DSNACICX user exit to define standard values for any of the DSNACICS parameters. You may wish to do this to isolate your application developers from needing to know the meaning of any of the parameters other than the transid, program name, and commarea contents and length.

Preparing a stored procedure to use DSNACICS

There are no special preparation steps for stored procedures that call DSNACICS. That is one of the advantages of using DSNACICS instead of the external CICS interface. You can use the same standard program preparation JCL that you use for your other stored procedures.

Result of running our sample stored procedure

We executed stored procedure EMPEXC3C from IBM Data Studio. The result of the call to DSNACICS in the SYSOUT output of the stored procedure address space is listed in Example 23-7 on page 477.

Example 23-7 Result of calling DSNACICS

```
++ START OF EMPEXC3C STARTING ++
WS-DEPTNO = E11
++ RETURN FROM DSNACICS:
RET-CODE:     000000000
MSG-AREA:
COMM-AREA:    E11OPERATIONS

WS-DEPTNAME:  OPERATIONS
++ SQLCODE AFTER OPEN   = 000000000
++ END OF EMPEXC3C ++
```

23.2 Accessing IMS databases from DB2 stored procedures

There are two methods for accessing IMS databases from a DB2 stored procedure: the IMS Open Database Access interface (ODBA), and stored procedures DSNAIMS and DSNAIMS2. The ODBA interface provides the capability to include IMS database calls within your DB2 stored procedure. The DB2-supplied stored procedures DSNAIMS and DSNAIMS2 provide the capability to access IMS transactions from a DB2 stored procedure. Which alternative you choose will depend on whether you plan to access the IMS data from existing

DB2 programs, or whether you plan to call existing IMS transactions and reuse IMS code from DB2.

23.2.1 Accessing IMS databases through the ODBA interface

The IMS Open Database Access interface (ODBA) provides access to IMS databases from a z/OS application, such as a DB2 stored procedure. Your stored procedure can issue a call using the AERTDLI API and pass a DL/I call function code that specifies the type of database call requested. For example, to read a specific record on an IMS database using a unique key, you would pass the call function code of GU for Get Unique. AERTDLI is not a stored procedure, so the call should be coded according to the normal call convention for the AERTDLI API as documented in the language that you use.

In our case study we make the assumption that a legacy IMS database exists that contains department information. The IMS database is called DEPT, and contains the DEPTNO and DEPTNAME fields comparable to those defined in the DEPT sample DB2 table. We created stored procedure EMPODB1C, which includes uses of ODBA to call the DEPT IMS database to retrieve the department name for a given department number. EMPODB1C then returns a result set of all employees for a given department number with the name of the department included for each employee. Stored procedure EMPODB1C is a version of EMPRSETC, the COBOL results set stored procedure, replacing the SELECT from the DEPT table with an IMS GU call using the AERTDLI API.

The following sections provide information on preparing your IMS and WLM environments for using ODBA, coding application programs to make ODBA calls, and preparing a program to use the interface.

IMS setup for using ODBA and DSNAIMS/DSNAIMS2

The following setup steps were done to prepare the IMS environment for access of the DEPT database by our sample ODBA stored procedure EMPODB1C:

1. Add the DBD and PSB macros to IMS stage 1 gen.
2. Define the DBD source and run the DBDGEN.
3. Define the PSB source and run the PSBGEN.
4. Define the ACBGEN source and run the ACBGEN.
5. Define the VSAM data set and run IDCAMS.
6. Define the source and run the dynamic allocation job for the database.
7. Define and run the DBRC registration for the DEPT database.
8. Load the database with DFSDDLT0.
9. Define and assemble the DFSPRP macro.
10. Perform the IMS Stage 2 gen or online change.
11. Define the execution WLM environment.
12. Define the associated WLM procedure for the WLM environment.

Details of each step are discussed in the following sections.

Add the DBD, PSB and IMS Tran macros to IMS stage 1 gen

Example 23-8 lists the macros we defined in our IMS.

Example 23-8 IMS Stage 1 gen macros

```
*********************************************************************
*  DB2 SP SG24-7083 REDBOOK FOR IMS ODBA AND DSNAIMS EXAMPLES
*********************************************************************
         DATABASE DBD=DEPTDB,ACCESS=UP               HDAM/VSAM
         SPACE 2
         APPLCTN PSB=DEPTPSBL,PGMTYPE=BATCH           LOAD PSB
         SPACE 2
         APPLCTN PSB=DEPTPSB,PGMTYPE=TP,SCHDTYP=PARALLEL
```

Define the DBD source and run the DBDGEN

Example 23-9 provides the DBDGEN source and execution procedure to create our DEPT database.

Example 23-9 IMS DBDGEN source to define the DEPT database

```
//DEPTDB EXEC PROC=DBDGEN,MBR=DEPTDB,SOUT='*'
//C.SYSIN  DD *
DBD    NAME=DEPTDB,ACCESS=HDAM,RMNAME=(DFSHDC40,40,100)
  DATASET DD1=DEPTDB1,DEVICE=3390,SIZE=4096
  SEGM   NAME=DEPT,PARENT=0,BYTES=43
  FIELD  NAME=(DEPTNO,SEQ,U),BYTES=3,START=1,TYPE=C
  FIELD  NAME=DEPTNAME,BYTES=40,START=4,TYPE=C
  DBDGEN
  FINISH
  END
//
```

Define the PSB source and run the PSBGEN

We defined two PSBs for our COBOL application that uses ODBA to access the IMS DEPT database: A load PSB, DEPTPSBL and an application PSB, DEPTPSB. Example 23-10 provides the source for the Load PSB, DEPTPSB.

Example 23-10 IMS PSBGEN source for the load PSB, DEPTPSBL

```
//PSBGEN   EXEC PROC=PSBGEN,MBR=DEPTPSBL,SOUT='*'
//C.SYSIN  DD *
PCB    TYPE=DB,DBDNAME=DEPTDB,PROCOPT=LS,KEYLEN=3
         SENSEG NAME=DEPT,PARENT=0
         PSBGEN LANG=ASSEM,PSBNAME=DEPTPSBL
         END
//
```

Example 23-11 provides the source for the application PSB after initial load is completed. The PCBNAME is required on the PSB for the Application Interface Block (AIB) control block used by the AERTDLI calling Application Programming Interface (API).

Example 23-11 IMS PSBGEN source for the application PSB, DEPTPSB

```
//PSBGEN   EXEC PROC=PSBGEN,MBR=DEPTPSB,SOUT='*'
//C.SYSIN  DD *
PCB    TYPE=DB,DBDNAME=DEPTDB,PCBNAME=DEPTPCB,PROCOPT=A,KEYLEN=3
         SENSEG NAME=DEPT,PARENT=0,PROCOPT=A
         PSBGEN LANG=ASSEM,PSBNAME=DEPTPSB
         END
//
```

Define the ACBGEN source and run the ACBGEN

Executing the source in Example 23-12 defines our ACBs to IMS for our DEPT example.

Example 23-12 ACBGEN for the DEPT DBD and PSBs

```
//ACBGEN   EXEC PROC=ACBGEN,SOUT='*',COMP='POSTCOMP'
//G.SYSIN  DD *
  BUILD DBD=DEPTDB
  BUILD PSB=DEPTPSB
  BUILD PSB=DEPTPSBL
//
```

Define the VSAM data set and run IDCAMS

Executing the source in Example 23-13 defines the VSAM data set for our IMS DEPT database.

Example 23-13 IDCAMS defines for DEPT VSAM data set

```
//ALLOCATE EXEC PGM=IDCAMS,DYNAMNBR=200
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
   DEFINE CLUSTER(                          -
                  NAME(IMS910H.DEPTDB1)     -
                  NONINDEXED                -
                  FREESPACE(10 10)          -
                  RECORDSIZE(2041 2041)     -
                  SHAREOPTIONS(3 3)         -
                  UNIQUE                    -
                  VOLUMES(TOTIM1)           -
                  CYLINDERS(02)             -
                  CONTROLINTERVALSIZE(2048) -
                 )                          -
            DATA(                           -
                  NAME(IMS910H.DEPTDB1.DATA) -
                 )
```

Define the source and run the dynamic allocation job for the database

The JCL and source in Example 23-14 create the dynamic allocation of the DEPT database. Using dynamic allocation is preferred to including a DD statement for the database in the IMS procs.

Example 23-14 Dynamic allocation definition for the DEPT database

```
//STEP01   EXEC PROC=IMSDALOC,SOUT='*'
//ASSEM.SYSIN DD *
  DFSMDA TYPE=INITIAL
  DFSMDA TYPE=DATABASE,DBNAME=DEPTDB
  DFSMDA TYPE=DATASET,DDNAME=DEPTDB1,                                  X
               DSNAME=IMS910H.DEPTDB1,                                 X
               DISP=SHR
  DFSMDA TYPE=FINAL
         END
```

Define and run the DBRC registration for the DEPT database

The JCL and source in Example 23-15 register the DEPTDB database in the DBRC recon data sets.

Example 23-15 DBRC registration for the DEPT database

```
//INITRCON EXEC PROC=DBRC
//D.SYSIN  DD *
  INIT.DB    DBD(DEPTDB) SHARELVL(3) TYPEIMS
  INIT.DBDS  DBD(DEPTDB) DDN(DEPTDB1) -
             DSN(IMS910H.DEPTDB1) -
             ICJCL(ICJCL) OICJCL(OICJCL) RECOVJCL(RECOVJCL) -
             REUSE RECOVPD(0) GENMAX(3)
//*
```

Load the database with DFSDDLT0

We load our DEPT database using the IMS utility, DFSDDLT0. The JCL and the data we loaded is shown in Example 23-16.

Example 23-16 Load JCL and data for DEPT database

```
//DLTO   PROC MBR=DFSDDLTO,PSB=DEPTPSBL,BUF=7,
//            SPIE=0,TEST=0,EXCPVR=0,RST=0,PRLD=,
//            SRCH=0,CKPTID=,MON=N,LOGA=0,FMTO=T,
//            IMSID=,SWAP=,DBRC=N,IRLM=N,IRLMNM=,
//             BKO=N,IOB=,SSM=,APARM=,RGN=2048K,
//            LOCKMAX=,GSGNAME=,TMINAME=
//G      EXEC PGM=DFSRRC00,REGION=&RGN,
//            PARM=(DLI,&MBR,&PSB,&BUF,
//            &SPIE&TEST&EXCPVR&RST,&PRLD,
//            &SRCH,&CKPTID,&MON,&LOGA,&FMTO,
//            &IMSID,&SWAP,&DBRC,&IRLM,&IRLMNM,
//            &BKO,&IOB,&SSM,'&APARM',
//            &LOCKMAX,&GSGNAME,&TMINAME)
//STEPLIB  DD DSN=IMS910H.SDFSRESL,DISP=SHR
//         DD DSN=IMS910H.PGMLIB,DISP=SHR
//IMS      DD DSN=IMS910H.PSBLIB,DISP=(SHR,PASS)
//         DD DSN=IMS910H.DBDLIB,DISP=(SHR,PASS)
//DFSRESLB DD DSN=IMS910H.SDFSRESL,DISP=SHR
//IEFRDER  DD DUMMY
// PEND
//DLTO EXEC DLTO
//DFSVSAMP DD *
VSRBF=2048,20
VSRBF=4096,20
VSRBF=8192,20
//PRINTDD  DD SYSOUT=T
//SYSUDUMP DD DUMMY
//SYSIN DD *
S 1 1 1 1 1    DEPTDB
L        ISRT  DEPT
L        DATA  A00SPIFFY COMPUTER SERVICE DIV.
L        DATA  B01PLANNING
L        DATA  C01INFORMATION CENTER
L        DATA  D01DEVELOPMENT CENTER
L        DATA  D11MANUFACTURING SYSTEMS
L        DATA  D21ADMINISTRATION SYSTEMS
L        DATA  E01SUPPORT SERVICES
L        DATA  E11OPERATIONS
L        DATA  E21SOFTWARE SUPPORT
L        DATA  F22BRANCH OFFICE F2
L        DATA  G22BRANCH OFFICE G2
L        DATA  H22BRANCH OFFICE H2
L        DATA  I22BRANCH OFFICE I2
```

```
L        DATA  J22BRANCH OFFICE J2
```

Define and assemble the DFSPRP macro

IMS ODBA has to be set up. This requires defining and assembling the DFSPRP macro created and assembled. Our system had Fast Path databases and transactions defined, so we needed to include FPBUF, FPBOF, and CNBA buffers. If you do not have FP configured for your IMS system, then these values can be 0. The recommendation for the CSECT name is to use a prefix of DFS followed by IMSID followed by 0. The DFSNAME in the PRP macro is where the output for the assembly of this macro resides. This data set needs to be included in the STEPLIB of the WLM procedure that executes our IMS ODBA stored procedure. See Example 23-17.

Example 23-17 DFSPRP macro that creates the DRA

```
DFSIMSH0 CSECT
         DFSPRP DSECT=NO,                                          X
               FUNCLV=1,          FUNCTION LEVEL                   X
               DDNAME=DFSDB2SP,   DDNAME FOR DRA RESLIB            X
               DSNAME=IMS710P.SDFSRESL,  DSNAME FOR DRA RESLIB     X
               DBCTLID=IMSG,      DBCTL IDENTIFIER                 X
               USERID=,           USER IDENTIFIER                  X
               MINTHRD=1,         MINIMUM NUMBER OF THREADS        X
               MAXTHRD=1,         MAXIMUM NUMBER OF THREADS        X
               TIMER=60,          IDENTIFY TIMER VALUE DEFAULT     X
               FPBUF=5,           NUMBER OF FP BUFFERS PER THREAD  X
               FPBOF=7,           NUMBER OF FP OVERFLOW BUFFERS    X
               SOD=A,             SNAP DATASET OUTPUT CLASS        X
               AGN=,              APPLICATION GROUP NAME           X
               TIMEOUT=60,        DRATERM TIMEOUT VALUE            X
               IDRETRY=0,         IDENTIFY RETRY COUNT             X
               CNBA=5             TOTAL FP NBA BUFFERS FOR CCTL
```

The JCL in Example 23-18 assembled our DFSPRP macro.

Example 23-18 Assembly JCL for the DFSPRP macro

```
//ASSEM EXEC HLASMCL
//C.SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//           DD DSN=IMS910H.SDFSMAC,DISP=SHR
//C.SYSIN    DD DSN=SG247083.ODBA.CNTL(DFSIMSH0),DISP=SHR
//L.SYSLMOD  DD DSN=IMS910H.SDFSRESL(DFSIMSH0),DISP=SHR
```

Perform the IMS Stage 2 gen or online change

We performed an online change to activate our DEPT database. Alternatively, an IMS Stage 2 gen could have been performed. We need to perform an online change for MODBLKS and ACBLIB. The online change JCL in Example 23-19 was used.

Example 23-19 IMS online change input

```
// JCLLIB ORDER=IMS910H.PROCLIB
//*
//* COPY MODBLKS
//*
//MODBLKS  EXEC PROC=OLCUTL,SOUT='*',TYPE=MODBLKS,IN=S,OUT=U
//*
//* COPY ACBLIB
//*
//*ACBLIB   EXEC PROC=OLCUTL,SOUT='*',TYPE=ACB,IN=S,OUT=U
```

```
//
```

Once the above online JCL has been successfully run, we need to activate the current system with this change. From the IMS console, issue the two commands in Example 23-20.

Example 23-20 IMS commands to activate IMS gen changes

```
/MODIFY PREPARE MODBLKS ACBLIB
/MODIFY COMMIT
```

Define the execution WLM environment

We created the WLM environment in Example 23-21 for executing our DB2 COBOL ODBA stored procedure.

Example 23-21 WLM environment for our DB2 COBOL ODBA case study

```
Appl Environment Name . . DB9AODBA
Description . . . . . . . DB9A IMS-ODBA- SG247083 - devl
Subsystem type  . . . . . DB2
Procedure name  . . . . . DB9AODBA
Start parameters  . . . . DB2SSN=&IWMSSNM,APPLENV=DB9AODBA
```

Define the WLM procedure for the WLM environment

The WLM procedure needs to include the STEPLIB data set we assembled our DFSPRP macro into, which in our case was IMS910H.SDFSRESL. The procedure also needs to include the //DFSRESLB DD statement. Additionally, we included data sets for the Distributed Debugger. See Example 23-22.

Example 23-22 WLM procedure for executing our DB2 COBOL stored procedure

```
//STEPLIB  DD    DSN=SG247083.DEVL.LOAD,DISP=SHR
//         DD    DISP=SHR,DSN=DB9A9.SDSNEXIT
//         DD    DISP=SHR,DSN=DB9A9.SDSNLOAD
//         DD    DISP=SHR,DSN=IMS910H.SDFSRESL
//         DD    DISP=SHR,DSN=IMS910H.PGMLIB
//         DD    DISP=SHR,DSN=EQAW.SEQAMOD
//SYSTCPD  DD    DSN=TCP.SC63.TCPPARMS(TCPDATA),DISP=SHR
//CEEDUMP  DD    SYSOUT=*
//DFSRESLB DD  DISP=SHR,DSN=IMS910H.SDFSRESL
```

Coding a stored procedure to use ODBA

COBOL stored procedure EMPODB1C includes COBOL language CALL statements to use the AERTDLI API to read a record from the IMS DEPT database, which is a copy of the DB2 sample DEPT table. A minimum of three calls are required: one to schedule a PSB, one to read the data, and one to deallocate the PSB.

The first call to AERTDLI schedules a PSB and initializes the IMS database environment. The call we issued in stored procedure EMPODB1C, along with the COBOL statements to initialize the parameters of the AIB, is shown in Example 23-23. The parameter APSB contains the value APSB, which is what is required to request allocation of a PSB.

Example 23-23 Sample logic for ODBA call to schedule a PSB

```
INITIALIZE AIB.
SET  AIBRESA1 TO NULLS.
SET  AIBRESA2 TO NULLS.
SET  AIBRESA3 TO NULLS.
```

```
MOVE ZEROES TO AIBRETRN.
MOVE ZEROES TO AIBREASN.
MOVE VAIBID TO AIBID.
MOVE LENGTH OF AIB TO AIBLEN.
MOVE LENGTH OF IOAREA TO AIBOALEN.
MOVE SPACES TO AIBSFUNC.
MOVE APSBNME TO AIBRSNM1.
MOVE TDBCTLID TO AIBRSNM2.
CALL 'AERTDLI' USING APSB, AIB.
```

Once the PSB has been scheduled, your program can issue a call to AERTDLI to read data from the appropriate database. The call we issued, along with the COBOL statements to set the parameters to read department D21 from the DEPT database is shown in Example 23-24. The reserved word SSA-KEY represents the key value being read from the database. In our case it is D21 for department D21. The fields IO-DEPTNO and IO-DEPTNAME make up IOAREA, which represents the record layout for the IMS DEPT database. The value of DPCBNAME is DEPTPCB, which represents the name of the PCB for our call. The GET-UNIQUE parameter represents the type of function call, which has two byte value of GU to read a unique record from the database with the value of D21 in the key.

Example 23-24 Sample logic for ODBA call to read an IMS database record

```
MOVE WS-DEPTNO  TO SSA-KEY.
MOVE SPACES     TO IO-DEPTNO.
MOVE SPACES     TO IO-DEPTNAME.
MOVE DPCBNME    TO AIBRSNM1.
CALL 'AERTDLI' USING GET-UNIQUE, AIB, IOAREA, SSA.
```

A stored procedure that uses ODBA must issue a DPSB PREP call to deallocate a PSB when all IMS work under that PSB is complete. The PREP keyword tells IMS to move in-flight work to an indoubt state. When work is in the indoubt state, IMS does not require activation of syncpoint processing when the DPSB call is executed. IMS commits or backs out the work as part of RRS two-phase commit when the stored procedure caller executes COMMIT or ROLLBACK. The call we issued to deallocate the PSB is shown in Example 23-25. The parameter DPSB contains the value DPSB, which is what is required to request deallocation of a PSB. The parameter SFPREP contains the value PREP, which is the keyword described above.

Example 23-25 Sample logic for ODBA call to deallocate a PSB

```
MOVE APSBNME TO AIBRSNM1.
MOVE SFPREP TO AIBSFUNC.
CALL 'AERTDLI' USING DPSB, AIB.
```

The complete source code for the COBOL stored procedure EMPODB1C can be found in Appendix B, “Additional material” on page 887.

Preparing a stored procedure to use the ODBA interface

Stored procedures that use the ODBA interface to access IMS data require modifications to the link edit step. Example 23-26 shows the overrides to the link edit step that we used to prepare stored procedure EMPODB1C. We include the ODBA module DFSCDLI0, which contains entry point AERTDLI. We also include module DSNRLI, which is the DB2 language interface for RRS. DSNRLI is needed to ensure coordination of two phase commit processing between DB2 and IMS.

Example 23-26 Sample link edit step for stored procedure with ODBA call

```
//LKED.SYSLMOD DD DSN=SG247083.DEVL.LOAD(EMPODB1C),
//             DISP=SHR
//LKED.LOAD DD DSN=IMS91OH.SDFSRESL,
//             DISP=SHR
//LKED.SYSIN   DD *
     INCLUDE SYSLIB(DSNRLI)
     INCLUDE LOAD(DFSCDLI0)
//*-----------------------------------------------
```

Result of running our sample stored procedure

We executed stored procedure EMPODB1C from IBM Data Studio. The result set for employees belonging to department D21 was returned, and the parameter list; see Example 23-27.

Example 23-27 Parameter list EMPODB1C

```
Name            Input           Output
PDEPTNO         D21
PDEPTNAME                       ADMINISTRATION SYSTEMS
PSQLCODE                        0
PSQLSTATE                       00000
PSQLERRMC                       *NULL*
OAIBRETRN                       0
OAIBREASN                       0
```

23.2.2 Accessing IMS databases through stored procedures

DB2 9 for z/OS provides two stored procedures for access to IMS transactions: DSNAIMS and DSNAIMS2. DSNAIMS2 adds IMS multi-segment transactions support.

Using the DSNAIMS stored procedure

While the ODBA interface is valuable if you wish to access IMS databases directly from your DB2 stored procedures, you may also wish to take advantage of existing IMS transactions from your DB2 stored procedures. The IMS transaction invocation stored procedure (DSNAIMS) allows you to access an IMS transaction from a DB2 stored procedure. This stored procedure uses the IMS Open Transaction Manager Access (OTMA) API to connect with IMS and execute the transactions. This functionality has been delivered through the maintenance stream in DB2 for OS/390 and z/OS Version 7 and V8 (respectively PTFs UQ96684 and UQ96685 for APAR PQ77702) and it is included in DB2 9 for z/OS.

DSNAIMS, written in the C language, provides comparable functionality for an IMS transaction environment to what DSNACICS provides for a CICS transaction environment. The DSNAIMS parameters are described in the DB2 documentation, specifically the *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841.

In “SYSPROC.DSNAIMS” on page 556 we show the syntax diagram and list the parameters.

A sample call statement is shown in Example 23-28.

Example 23-28 Sample CALL to DSNAIMS

```
EXEC SQL
         CALL SYSPROC.DSNAIMS(
          :DSNAIMS-FUNCTION, :DSNAIMS-2PC, :XCF-GROUP-NAME,
          :XCF-IMS-NAME, :RACF-USERID, :RACF-GROUPID, :IMS-LTERM,
```

```
          :IMS-MODNAME, :IMS-TRAN-NAME, :IMS-DATA-IN, :IMS-DATA-OUT,
          :OTMA-TPIPE-NAME, :OTMA-DRU-NAME, :OTMA-USER-DATA-IN,
          :OTMA-USER-DATA-OUT, :STATUS-MESSAGE, :RETURN-CODE
                           )
      END-EXEC.
```

DSNAIMS prerequisites

The following functions are required before installing and executing the DSNAIMS stored procedure:

- A WLM-managed stored procedure address space in which to run DSNAIMS.
- DB2 Version 8 requires PTF UQ94696 for APAR PQ89544 and PTF UK03998 for APAR PK04339 if the text includes X'00's in the input variable string.
- When using two-phase commit for DSNAIMS (DSNAIMS_2PC=Y), you need:
  - IMS V7 with UQ78980 and the IMS control region parameter RRS=Y
  - IMS V8 with UQ70789 and the IMS control region parameter RRS=Y
- Other recommended maintenance for DSNAIMS:
  - PK48891 (UK29722) for IMS V9, PK52490 (UK30363) for IMS V10
  - PK30395 (UK18394) for IMS V8, PK30387 (UK15224 and UK18393) for IMS V9
  - PK25672 (UK25116) for IMS V8, PK16294 (UK25115) for IMS V9

DSNAIMS limitations

DSNAIMS cannot handle the following:

- Specifying OTMA commit mode explicitly
- IMS conversational transaction
- IMS multi-segment messages - This restriction is removed in DSNAIMS2; see "Using the DSNAIMS2 stored procedure" on page 489.

DSNAIMS setup

The following are two steps to take to ensure that DSNAIMS is ready for execution:

1. The job DSNTIJIM (provided in the SDSNSAMP data set) can be used to issue the CREATE PROCEDURE statement for DSNAIMS and will grant the execution of the procedure to PUBLIC. This job needs to be customized to fit the parameters of your system.
2. OTMA Callable Interface provides a stand-alone C/I initialization program DFSYSVI0 that must be run after every MVS IPL to initialize the C/I. You will need to add an entry in the MVS program properties table (PPT) for DFSYSVI0. Refer to the *IMS OTMA Guide and Reference* for an explanation of the C/I initialization.

Example 23-29 DSNAIMS format

```
SYSPROC.DSNAIMS(IN    DSNAIMS_FUNCTION CHAR(8),
                IN    DSNAIMS_2PC      CHAR(1),
                IN    XCF_GROUP_NAME   CHAR(8),
                IN    XCF_IMS_NAME     CHAR(16),
                IN    RACF_USERID      CHAR(8),
                IN    RACF_GROUPID     CHAR(8),
                INOUT IMS_LTERM        CHAR(8),
                INOUT IMS_MODNAME      CHAR(8),
                IN    IMS_TRAN_NAME    CHAR(8),
                IN    IMS_DATA_IN      VARCHAR(32000),
                OUT   IMS_DATA_OUT     VARCHAR(32000),
```

```
IN    OTMA_TPIPE_NAME  CHAR(8),
IN    OTMA_DRU_NAME    CHAR(8),
IN    USER_DATA_IN     VARCHAR(1022),
IN    USER_DATA_OUT    VARCHAR(1022),
OUT   STATUS_MESSAGE   VARCHAR(120),
OUT   RETURN_CODE      INT)
```

DSNAIMS examples

For a first example we used IMS Version 9 and IMS sample application DFSSAM02 and transaction PART.

See Example 23-30 for the result when we execute this transaction from a 3270 terminal.

Example 23-30 PART transaction

```
Input:

PART AN96OC1O

Response:

Part...........     AN96OC1O; Desc........... WASHER
 Proc Code......           74; Inv Code.......       2
 Make Dept......        12-OO; Plan Rev Num...
 Make Time......           63; Comm Code......      14
```

We used IBM Data Studio to call DSNAIMS. See Example 23-31 for the input parameters and the result.

Example 23-31 DSNAIMS execution in IBM DATA Studio

```
Name                Input           Output
DSNAIMS_FUNCTION    SENDRECV
DSNAIMS_2PCN
XCF_GROUP_NAME      IMSHEXCF
XCF_IMS_NAME        SCSIMS9H
RACF_USERID
RACF_GROUPID
IMS_LTERM
IMS_MODNAME
IMS_TRAN_NAME
IMS_DATA_IN         PART AN96OC1O
IMS_DATA_OUT                        Part...........     AN96OC1O;
                                    Desc........... WASHER
                                    Proc Code......           74;
                                    Inv Code.......       2
                                    Make Dept......       12-OO;
                                    Plan Rev Num...
                                    Make Time......           63;
                                    Comm Code......      14
OTMA_TPIPE_NAME     ABC
OTMA_DRU_NAME
OTMA_USER_DATA_IN
OTMA_USER_DATA_OUT                  *NULL*
STATUS_MESSAGE                      DSNA315I DSNAIMS FUNCTION SENDRECV HAS COMPLETED
SUCCESSFULLY.
RETURN_CODE                         0
```

Sample parameters for executing a simple IMS command are shown in Example 23-32.

Example 23-32 IMS command

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMSHEXCF", "SCSIMS9H",
                        "", "", "", "", "",
                        "/LOG Hello World.", ims_data_out, "", "", "",
                        user_out, error_message, rc)
```

Sample parameters for executing the IMS IVTNO transaction are shown in Example 23-33.

Example 23-33 IMS transaction

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMSHEXCF", "SCSIMS9H",
                        "", "", "", "", "",
                        "IVTNO     DISPLAY LAST1     ", ims_data_out,
                        "", "", "", user_out, error_message, rc)
```

Sample parameters for Send-only transaction invocation are shown in Example 23-34.

Example 23-34 Send only transaction

```
CALL SYSPROC.DSNAIMS("SEND "N", "IMSHEXCF", "SCSIMS9H",
                        "", "", "", "", "",
                        "IVTNO     DISPLAY LAST1     ", ims_data_out,
                        "DSNAPIPE", "", "", user_out, error_message,
                        rc)
```

Sample parameters for Receive-only transaction invocation are shown in Example 23-32.

Example 23-35 Receive only transaction

```
CALL SYSPROC.DSNAIMS("RECEIVE", "N", "IMSHEXCF", "SCSIMS9H",
                        "", "", "", "", "",
                        "IVTNO     DISPLAY LAST1     ", ims_data_out,
                        "DSNAPIPE", "", "", user_out, error_message,
                        rc)
```

DSNAIMS tips

Since DSNAIMS only connects to one IMS at a time, the following is a list of suggested steps to connect to multiple IMS subsystems simultaneously.

- Make a copy of the supplied job DSNTIJIM and customize it to your environment in accordance with the procedures listed in the document.
- Change the procedure name from SYSPROC.DSNAIMS to another name that will help you remember its target IMS (that is, SYSPROC.DSNAIMSB).
- Change the WLM environment to a different name for DSNAIMSB.
- Make sure to leave the “EXTERNAL NAME” option as “DSNAIMS”.
- Run the new job to create a second instance of the stored procedure.
- Always use the same XCF Group and Member names for each stored procedure instance. This will ensure that every time that stored procedure instance is invoked, a proper connection to the intended target IMS will be created. For example:

  ```
  CALL SYSPROC.DSNAIMS(“SENDRECV”, “N”, “IMS7GRP”, “IMS7TMEM”, …)
  ```

```
CALL SYSPROC.DSNAIMSB(“SENDRECV”, “N”, “IMS8GRP”, “IMS8TMEM”, …)
```

Using the DSNAIMS2 stored procedure

DSNAIMS is only capable of sending a single-segment transaction. DSNAIMS2 has the same functionality as DSNAIMS but with the additional multi-segment input support.

DSNAIMS2 is available in DB2 9 for z/OS. APAR PK26421 (PTF UK26421) and APAR PK32332 (PTF UK18752) also apply.

The same function is made available to DB2 V8 through the maintenance stream: PK07907 is the controlling APAR, currently open.

A new input parameter, OTMA_DATA_INSEG, is defined to obtain information on how many segments are in the message and how to divide the message into segments to construct a valid multi-segment message to send to IMS using the OTMA Callable Interface.

The syntax of DSNAIMS2 is the same as DSNAIMS except for the addition of one new IN parameter, OTMA_DATA_INSEG. The new IN parameter OTMA_DATA_INSEG takes as input the total number of segments and the length of each segment. The data type of OTMA_DATA_INSEG is VARCHAR(512). The format of the input string is the total number of input segments, followed by a list of lengths of all the segments, the values in the string separated by semicolons.

The definition of DSNAIMS2 is shipped in the new sample file DSNTIJI2 (provided in the SDSNSAMP data set). It is listed in Example 23-36.

Example 23-36 DSNAIMS2 DDL

```
CREATE PROCEDURE
SYSPROC.DSNAIMS2(IN DSNAIMS_FUNCTION CHAR(8),
IN DSNAIMS_2PC CHAR(1),
IN XCF_GROUP_NAME CHAR(8),
IN XCF_IMS_NAME CHAR(16),
IN RACF_USERID CHAR(8),
IN RACF_GROUPID CHAR(8),
INOUT IMS_LTERM CHAR(8),
INOUT IMS_MODNAME CHAR(8),
IN IMS_TRAN_NAME CHAR(8),
IN IMS_DATA_IN VARCHAR(32000),
OUT IMS_DATA_OUT VARCHAR(32000),
IN OTMA_TPIPE_NAME CHAR(8),
IN OTMA_DRU_NAME CHAR(8),
IN USER_DATA_IN VARCHAR(1022),
IN USER_DATA_OUT VARCHAR(1022),
OUT STATUS_MESSAGE VARCHAR(120),
IN OTMA_DATA_INSEG VARCHAR(512),
OUT RETURN_CODE INT)
FENCED
EXTERNAL NAME DSNAIMS2
LANGUAGE C
PARAMETER STYLE GENERAL WITH NULLS
PARAMETER CCSID EBCDIC
PARAMETER VARCHAR STRUCTURE
NO SQL
NOT DETERMINISTIC
WLM ENVIRONMENT !WLM!
STAY RESIDENT YES
PROGRAM TYPE SUB
```

```
SECURITY !SECURITY!
COMMIT ON RETURN NO;
```

Sending a multi-segment input transaction

Sample parameters for executing a multi-segment IMS transaction are listed in Example 23-37.

Example 23-37 Multi segment transaction

```
CALL SYSPROC.DSNAIMS2("SEND","N","IMS8GRP","IMS8TMEM",
"IMSCLNM","","","","","","",
"SDK1 1ST SEGMENT FROM CI 2ND SEGMENT FROM CI",
ims_data_out,"","","",user_out, error_message,
      "2;25;20",rc)
```

In this example:

IMS_DATA_IN - Contains "SDK1 1ST SEGMENT FROM CI 2ND SEGMENT FROM CI"

OTMA_DATA_INSEG - contains "2;25;20". This indicates to DSNAIMS2 that IMS_DATA_IN contains 2 segments, and their length. The first segment is 25 bytes long, the second segment is 20 bytes long.

Important: you need to make sure these lengths are correct. If the total length of the segments do not match up to the length of the string in IMS_DATA_IN, IMS rejects the transaction with RC=8 and Reason Code=40.

See the chapter “OTMA Callable Interface” in *IMS Version 9: Open Transaction Manager Access Guide and Reference*, SC18-7829 for a complete list of codes and messages from the C/I APIs.

Sending a single-segment transaction using DSNAIMS2

Sample parameters for executing a single-segment IMS transaction are listed in Example 23-38.

Example 23-38 Single segment transaction

```
CALL SYSPROC.DSNAIMS2("SEND","N","IMS8GRP","IMS8TMEM",
"IMSCLNM","","","","","IVTNO",
"DISPLAY LAST1",ims_data_out,"","","",
user_out, error_message,NULL,rc)
```

To indicate that IMS_DATA_IN contains single-segment data, OTMA_DATA_INSEG should be set to a NULL value, an empty string, the string "0" or "0;".

Enabling a DSNAIMS2 stored procedure

1. DSNAIMS2 has the same prerequisites as DSNAIMS concerning the IMS version (Version 7 and up): enablement of OTMA Callable Interface, a WLM-managed stored procedure address space, and RRSAF enablement.
2. Customize and run job DSNTIJI2 to define DSNAIMS2 to DB2. See the prolog of DSNTIJI2 for instructions. If you prefer to use the name DSNAIMS in your application, customize DSNTIJI2 to define the stored procedure to DB2 as DSNAIMS. However, the EXTERNAL NAME must still be DSNAIMS2.

23.3 Accessing DB2 stored procedures from CICS

CICS Transaction Server for z/OS, referred to as CICS in this chapter, comes with a DB2 attachment facility, which provides CICS applications the ability to access DB2 data while operating in a CICS environment. Each CICS region can be connected to only one DB2 subsystem at a time.

CICS applications that access DB2 objects, including stored procedures, must include DB2 precompiler and bind steps during the program preparation process. The precompiler step builds a DBRM and converts the source code into a format acceptable to the CICS translator step. The bind step reads the DBRM created in the precompiler step, and produces an application plan. See “Preparing a CICS application program that accesses DB2” in *CICS DB2 Guide Version 3 Release 1*, SC34-6457-01 for more details on the program preparation process for CICS programs that access DB2 objects.

Stored procedures are accessed from CICS programs by issuing SQL CALL statements. A CALL statement that references a stored procedure must be bracketed between EXEC SQL and the statement terminator appropriate for the host language. For COBOL programs, the statement terminator is `END-EXEC`**.** Example 23-39 shows what a call to stored procedure EMPRSETC from a COBOL CICS application might look like.

Example 23-39 Sample SQL CALL statement in a CICS program

```
EXEC SQL
     CALL EMPRSETC( :PDEPTNO
                   ,:PDEPTNAME
                   ,:PSQLCODE
                   ,:PSQLSTATE
                   ,:PSQLERRMC
                  )
END-EXEC.
```

The CICS application program must include logic to set the host variables prior to executing the CALL, and must include logic to handle any error conditions returned by the stored procedure. These two logic components are no different than what is required of a batch program written in the same host language.

23.4 Accessing DB2 stored procedures from IMS

IMS Transaction Manager, referred to as IMS TM in this chapter, and IMS batch come with a DB2 attachment facility, which provides IMS applications the ability to access DB2 data while operating in an IMS TM or IMS batch environment. Each IMS TM or batch region can be connected to only one DB2 subsystem at a time.

IMS applications that access DB2 objects, including stored procedures, must include DB2 precompiler and bind steps during the program preparation process. The precompiler step builds a DBRM, and converts the source code into a format acceptable to the CICS translator step. The bind step reads the DBRM created in the precompiler step and produces an application plan. See 1.4.1, “Preparing a CICS application program that accesses DB2”, in the manual *CICS Transaction Server for z/OS Version 2.2 CICS DB2 Guide*, SC34-6014-07 for more details on the program preparation process for CICS programs that access DB2 objects.

Stored procedures are accessed from IMS programs by issuing SQL CALL statements. A CALL statement that references a stored procedure must be bracketed between EXEC SQL,

and the statement terminator appropriate for the host language. For COBOL programs, the statement terminator is END-EXEC**.** Example 23-40 shows what a call to stored procedure EMPRSETC from a COBOL IMS application might look like.

Example 23-40 Sample SQL CALL statement in an IMS program

```
EXEC SQL
     CALL EMPRSETC( :PDEPTNO
                   ,:PDEPTNAME
                   ,:PSQLCODE
                   ,:PSQLSTATE
                   ,:PSQLERRMC
                  )
END-EXEC.
```

The IMS application program must include logic to set the host variables prior to executing the CALL, and must include logic to handle any error conditions returned by the stored procedure. These two logic components are no different than what is required of a batch program written in the same host language.

24

DB2-supplied stored procedures

There are several very useful stored procedures that have been developed to provide server-side database and system administration functions. They are now supplied with the DB2 server and installed using the job DSNTIJSG. Attention has been paid to providing convergence of stored procedure names, parameters, result sets, and return codes, with the intention to simplify the usage and create a consistent picture for the user.

In this chapter we provide an overview of the administrative stored procedures that are now supplied with the DB2 server. The overview is enriched with comprehensive Java samples that are contained in Appendix A, “Samples for using DB2-supplied stored procedures” on page 807 and provide an excellent jump-start for working with them.

All of this enables you to write client applications that perform advanced DB2 and z/OS functions.

This chapter contains the following:

- Overview of the DB2-supplied stored procedures
- Administrative enablement stored procedure - details
- Scheduling administrative stored procedures with the DB2 task scheduler
- Common SQL API - Administration functions common to all IBM data servers
- Using the DB2-supplied stored procedures

24.1 Overview of the DB2-supplied stored procedures

There are stored procedures supplied with the DB2 server and installed using DSNTIJSG to be used by application programs. The trend of providing standardized SQL access to administration functions in the form of procedures and table functions is strategic within the entire DB2 family. Since many application environments use the context of a single connection to drive all access to their databases, it is natural, much easier and seamless to have common SQL interfaces to perform administration functions and ease Java application integration through JDBC or SQLJ.

SQL APIs are already used by most DB vendors. This type of access ensures a base standard infrastructure which satisfies customer requirements and enables IBM to build on other application layers such as Java administration APIs, Web services, and so on.

DB2 for z/OS administration requires DB2 commands, DSN subcommands, and MVS console commands, and the use of JCL. By providing an SQL interface to DB2 commands, DSN subcommands and JCL, DB2 provides a base standard infrastructure to build upon, not only for DB2 tools but also for application vendors, such as SAP®, who increasingly leverage the advantages of Web applications running in application servers like WebSphere Application Server. The DB2-supplied stored procedures are meant to provide that infrastructure.

The intent of this chapter is to show how most of the DB2 administration functions can easily be executed through DB2-supplied stored procedures.

The three categories of DB2-supplied procedures discussed here in detail are:

- Administrative enablement stored procedures

  A set of stored procedures that allow you to execute DB2 commands, DSN subcommands, UNIX commands, manage jobs, manage data sets, execute utilities in parallel, and administer systems.

- DB2 task scheduler

  A set of stored procedures that allow for server-side scheduling of administrative tasks.

- Common SQL API

  Procedures that are implemented with the same signature on all IBM data servers and basically return the operating system and data server configuration in an XML format.

24.1.1 DB2-supplied stored procedures

In this section, we list most of the DB2-supplied stored procedures, grouped by the function they perform on the connected DB2 subsystem or LPAR in which the subsystem runs. We document their new name and original name, and group the procedures by the function they perform on the connected DB2 subsystem or LPAR in which the subsystem runs. We also list new stored procedures that extend the administration capabilities.

Furthermore, information about required APF authorization and program control is provided for every stored procedure. If one of these attributes applies, refer to 24.1.2, “Setting up DB2-supplied stored procedures” on page 502, to figure out the respective configuration requirements.

Some of the stored procedures listed here were previously shipped as part of the DB2 Management Clients Package's z/OS Enablement component (FMID JDB881D and JDB991D). These are now included in the DB2 base (FMID HDB8810 and HDB9910) with modifications and enhancements. They have been given more task oriented names and their

signatures, input tables, and result sets have been altered to make them consistent and descriptive. So refer to the syntax diagrams, input and result table formats documented in 24.2, “Administrative enablement stored procedure - details” on page 521 when porting your application to use these new stored procedures. For your convenience, the Original name of each of these stored procedures is shown in a column next to the Name column in the tables that follow.

Unless otherwise specified, the schema name SYSPROC is assumed for these procedures.

Command execution

Table 24-1 shows the stored procedures that execute DB2 commands, DSN subcommands, and z/OS UNIX commands, by listing name (new and old), the function provided, and whether the stored procedure is APF-authorized and program-controlled.

Table 24-1 Command execution stored procedures.

Name	Original name	Function	APF - authorized	Program - controlled
ADMIN_COMMAND_UNIX	DSNACCUC	Issue a z/OS UNIX command	No	Yes
ADMIN_COMMAND_DSN	DSNACCTS	Issue a DSN subcommand	No	No
ADMIN_COMMAND_DB2	DSNACCMN / DSNACCMD	Issue a DB2 command	No	No
DSNTBIND	none	Submit a BIND PACKAGE command requested from a remote requester environment.	No	No

CALL syntax - For more details on the CALL syntax of every stored procedure, refer to 24.2.1, “Command execution” on page 521.

Sample program - A sample Java program for all command execution stored procedures is in Appendix A, “Samples for using DB2-supplied stored procedures” on page 807.

- AdminDB2Command contains a sample on how to use the stored procedure ADMIN_COMMAND_DB2 to execute multiple DB2 commands, e.g. DISPLAY DDF DETAIL and parse the result set. Refer to A.3, “Issue DB2 commands with AdminDB2Command” on page 819.
- AdminUNIXCommand contains a sample on how to use the stored procedure ADMIN_COMMAND_UNIX to execute a “ls -l“ UNIX command under the provided userid, and display the command output. Refer to A.7, “Issue USS commands with AdminUNIXCommand” on page 852.
- AdminDSNSubcommand contains a sample on how to use the stored procedure ADMIN_COMMAND_DSN to execute the DSN subcommand REBIND PACKAGE. Refer to A.8, “Issue DSN subcommands with AdminDSNSubcommand” on page 855.

Job management

Table 24-2 lists all stored procedures that can be employed for remote job management.

Table 24-2 Job management stored procedures.

Name	Original name	Function	APF - authorized	Program - controlled
ADMIN_JOB_SUBMIT	DSNACCJS	Submit a job	No	Yes
ADMIN_JOB_FETCH	DSNACCJF	Fetch the output of a job	Yes	Yes
ADMIN_JOB_QUERY	DSNACCJQ	Query the status of a job	Yes	Yes
ADMIN_JOB_CANCEL	DSNACCJP	Purge or cancel a job	Yes	Yes

CALL syntax - For more details on the CALL syntax of every stored procedure, refer to 24.2.2, “Job management” on page 530.

Sample program - A sample Java program that executes all job management stored procedures can be found in A.6, “Submit JCL with AdminJob” on page 845.

Data set management

Table 24-3 lists the stored procedures that manage data sets.

Table 24-3 Data set management stored procedure

Name	Original name	Function	APF - authorized	Program - controlled
ADMIN_DS_BROWSE	DSNACCDF	Browse a data set, Generation Data Set (GDS) or library member	Yes	No
ADMIN_DS_WRITE	DSNACCDS	Create, append, or replace a data set, library member, or GDS.	Yes	No
ADMIN_DS_LIST	DSNACCDL	List cataloged data sets, library or Generation Data Group (GDG), members in a library, or GDS in a GDG.	Yes	No
ADMIN_DS_RENAME	DSNACCDR	Rename a data set, library or library member.	Yes	No
ADMIN_DS_DELETE	DSNACCDD	Delete a data set, library, GDS, or library member.	Yes	No
ADMIN_DS_SEARCH	DSNACCDE	Check the existence of a data set, library, GDG, GDS, or library member.	Yes	No

CALL syntax - Detailed information on how to issue a CALL on a data set management stored procedure can be found in 24.2.3, “Data set management” on page 536.

Sample program - A sample Java program that executes all data set management stored procedures can be found in A.5, “Manage data sets with AdminDataSet” on page 838.

System administration

Table 24-4 lists the procedures that can be employed for system administration.

Table 24-4 System administration stored procedure

Name	Original name	Function	APF - authorized	Program - controlled
ADMIN_INFO_HOST	DSNACCGH	Get the fully qualified TCP/IP host names for non-data sharing subsystems or members of a data sharing group	No	No
ADMIN_INFO_SSID	DSNACCSS	Get the DB2 subsystem identifier	No	No
DSNACICS	none	Invoke CICS server programs	No	No
DSNLEUSR	none	Stores encrypted credentials in the SYSIBM.USERNAMES table	No	No
DSNAIMS	none	Invoke IMS transactions	No	No
DSNAIMS2	none	Invoke IMS transactions	No	No
DSNWZP	none	Displays the current settings of system parameters	No	No
DSNWSPM	none	Formats IFCID 148 records.	No	No
WLM_REFRESH	none	Refreshes a WLM environment	No	No
DSNTPSMP	none	Prepares an external SQL procedure for execution.	No	No

CALL syntax - Refer to 24.2.4, “System administration” on page 548 for more information on the CALL syntax for the procedures ADMIN_INFO_HOST, ADMIN_INFO_SSID, DSNACICS, DSNLEUSR, DSNAIMS and DSNAIMS2. Further examples of DSNACICS, DSNAIMS and DSNAIMS2 are in Chapter 23, “Accessing CICS and IMS” on page 469.

The other procedures contained in the table are only listed to provide a complete picture of all system administration server programs. For more detailed information on these, refer to the DB2 Version 9.1 for z/OS *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840.

Sample program - A sample Java program that executes ADMIN_INFO_HOST and ADMIN_INFO_SSID can be found in A.1, “Display DB2 system information with AdminSystemInformation” on page 808.

Utility execution

Table 24-5 lists the stored procedures that support utility execution.

Table 24-5 Utility execution stored procedure.

Name	Original name	Function	APF - authorized	Program - controlled
ADMIN_UTL_SCHEDULE	DSNACCMO	Execute multiple utilities in parallel	No	No

Name	Original name	Function	APF - authorized	Program - controlled
ADMIN_UTL_SORT	DSNACCST	Sort multiple objects for utility execution	No	No
DSNUTILS	none	Execute DB2 online utilities	Yes	No
DSNUTILU	none	Executes DB2 online utilities and accepts control statements in Unicode UTF-8 characters	Yes	No
DSNACCOX	DSNACCOR	Make recommendations for object maintenance	No	No

CALL syntax - More detailed information on the CALL syntax for the two stored procedures ADMIN_UTL_SCHEDULE and ADMIN_UTL_SORT, as well as some more best practices on these two utility execution stored procedures can be found in 24.2.5, “Utility execution” on page 560.

The other stored procedures in this table are only listed to provide a complete picture; more details can be found in *DB2 Version 9.1 for z/OS Utility Guide an Reference,* SC18-9855.

Sample program - A sample Java program that executes ADMIN_UTL_SCHEDULE and DSNACCOX can be found in A.4, “Automate RUNSTATS with AdminUtilityExecution” on page 827.

Scheduling tasks

Table 24-6 lists the stored procedures that allow the scheduling of either stored procedures or JCL tasks. This is the SQL interface for the DB2 task scheduler. These procedures are new and were not part of the DB2 UDB Control Center.

Table 24-6 Stored procedures for scheduling administrative tasks.

Name	Original name	Function	APF - authorized	Program - controlled
ADMIN_TASK_ADD* *This procedure was temporarily named ADMIN_TASK_SCHEDULE	none	Schedule a stored procedure or JCL task	No	No
ADMIN_TASK_REMOVE	none	Remove a scheduled task	No	No
ADMIN_TASK_LIST (UDF)	none	List all scheduled tasks	No	No
ADMIN_TASK_STATUS (UDF)	none	List the last execution status of all scheduled tasks	No	No

CALL syntax - More detailed information on the CALL syntax for the two stored procedures ADMIN_TASK_ADD, ADMIN_TASK_REMOVE and the table functions ADMIN_TASK_LIST, ADMIN_TASK_STATUS can be found in 24.3, “Scheduling administrative stored procedures with the DB2 task scheduler” on page 569. This section also provides a checklist for distinct scheduling scenarios.

For further details, refer to *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840 and *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

Sample programs - Sample Java programs can be found in A.9, "Task Scheduler Sample Use cases" on page 858.

Common SQL API

The Common SQL API is a set of three stored procedures, shown in Table 24-7, that return XML documents containing data server system or configuration related information. All three procedures feature the same signature and are common to all IBM data servers.

Table 24-7 Common SQL API stored procedures

Name	Original name	Function	APF - authorized	Program - controlled
GET_CONFIG	none	Collect data server configuration information	Yes	No
GET_MESSAGE	none	Retrieve the Short Message Text associated to a certain SQLCODE	No	No
GET_SYSTEM_INFO	none	Collect data server system information	Yes	No

CALL syntax - For more information on the concept and how to work with the Common SQL API, refer to 24.4, "Common SQL API - Administration functions common to all IBM data servers" on page 589. Additional details can be found in *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840 and *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

Sample program - Sample Java programs can be found in A.10, "Invoking the Common SQL API stored procedures" on page 870.

XML schema processing

The set of stored procedures that is provided to you for registration and removal of XML schema repositories is a collection of five stored procedures, shown in Table 24-8.

Table 24-8 XML schema repository stored procedures

Name	Original name	Function	APF - authorized	Program - controlled
XSR_REGISTER	none	First procedure in XML schema registration process	No	No
XSR_ADDSCHEMADOC	none	Add every XML schema other than the primary XML schema document	No	No
XSR_COMPLETE	none	Final procedure in XML schema registration process	No	No
XSR_REMOVE	none	Remove all components of an XML schema	No	No

Name	Original name	Function	APF - authorized	Program - controlled
XDBDECOMPXML	none	Extracts values from serialized XML data and populates relational tables with the values	No	No

CALL syntax - For more information on the concept and how to work with the DB2 provided stored procedures for XML schema registration and removal, refer to *DB2 Version 9.1 for z/OS XML Guide*, SC18-9858.

Unified Debugger

The set of stored procedures that are used by the Unified Debugger that is built into Data Studio is listed in Table 24-9 for your reference.

Table 24-9 Unified Debugger stored procedures

Name	Original name	APF - authorized	Program - controlled
DBG_ENDSESSIONMANAGER	none	No	No
DBG_INITIALIZECLIENT	none	No	No
DBG_LOOKUPSESSIONMANAGER	none	No	No
DBG_PINGSESSIONMANAGER	none	No	No
DBG_RECVCLIENTREPORTS	none	No	No
DBG_SENDCLIENTCOMMANDS	none	No	No
DBG_SENDCLIENTREQUESTS	none	No	No
DBG_TERMINATECLIENT	none	No	No
DBG_RUNSESSIONMANAGER	none	Yes	No
DB2DEBUG.DEBUGGERLEVEL	none	No	No
DB2DEBUG.CREATE_SESSION	none	No	No
DB2DEBUG.DESTROY_SESSION	none	No	No
DB2DEBUG.QUERY_SESSION	none	No	No
DB2DEBUG.LIST_SESSION	none	No	No
DB2DEBUG.PUT_COMMAND	none	No	No
DB2DEBUG.GET_REPORT	none	No	No

For more information on the concept and how to work with the DB2-provided stored procedures for the Unified Debugger, refer to Chapter 28, “Tools for debugging DB2 stored procedures” on page 735.

ODBC / JDBC metadata

Before you can use certain functions of the IBM DB2 Driver for JDBC and SQLJ on a DB2 for z/OS subsystem, you need to install the set of stored procedures shown in Table 24-10.

These procedures and the associated database objects are installed with the help of the post-install job DSNTIJMS.

Table 24-10 ODBC/JDBC metadata stored procedures

Name	Original name	APF - authorized	Program - controlled
SYSIBM.SQLCOLPRIVILEGES	none	No	No
SYSIBM.SQLCOLUMNS	none	No	No
SYSIBM.SQLFOREIGNKEYS	none	No	No
SYSIBM.SQLPRIMARYKEYS	none	No	No
SYSIBM.SQLPROCEDURECOLS	none	No	No
SYSIBM.SQLPROCEDURES	none	No	No
SYSIBM.SQLSPECIALCOLUMNS	none	No	No
SYSIBM.SQLSTATISTICS	none	No	No
SYSIBM.SQLTABLEPRIVILEGES	none	No	No
SYSIBM.SQLTABLES	none	No	No
SYSIBM.SQLGETTYPEINFO	none	No	No
SYSIBM.SQLUDTS	none	No	No
SYSIBM.SQLCAMESSAGE	none	No	No

For more information about enabling the DB2-supplied stored procedures and defining the tables used by the IBM DB2 Driver for JDBC and SQLJ, refer to *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java*, SC18-9842.

Java procedure processing routines

One way to organize the classes for a Java routine is to collect those classes into a JAR file. If you do this, you need to install the JAR file into the DB2 catalog. Table 24-11 lists the DB2-provided built-in stored procedures that are used for JAR file management.

Table 24-11 Java procedure processing routines

Name	Original name	Function	APF - authorized	Program - controlled
SQLJ.INSTALL_JAR	none	Installs a JAR file into the local DB2 catalog	No	No
SQLJ.REPLACE_JAR	none	Replaces an existing JAR file in the local DB2 catalog	No	No
SQLJ.REMOVE_JAR	none	Deletes a JAR file from the local DB2 catalog	No	No
SQLJ.DB2_INSTALL_JAR	none	Installs a JAR file into the local DB2 catalog or a remote DB2 catalog	No	No

Name	Original name	Function	APF - authorized	Program - controlled
SQLJ.DB2_REPLACE_JAR	none	Replaces an existing JAR file in the local DB2 catalog or a remote DB2 catalog.	No	No
SQLJ.DB2_REMOVE_JAR	none	Deletes a JAR file from the local DB2 catalog or a remote DB2 catalog	No	No
SQLJ. DB2_UPDATEJARINFO	none	Inserts class, class source, and associated options for a previously installed JAR file in a local or remote catalog.	No	No
SQLJ.ALTER_JAVA_PATH	none	Modifies the class resolution path of an previously installed JAR file to a specified value	No	No

CALL syntax - For more information on how to work with the DB2 provided stored procedures for Java procedure processing, refer to *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java,* SC18-9842.

24.1.2 Setting up DB2-supplied stored procedures

The complexity to set up the DB2-supplied stored procedures has been decreased dramatically. Now there are only four steps necessary to enable them. The four boxes in Figure 24-1 depict these four steps.

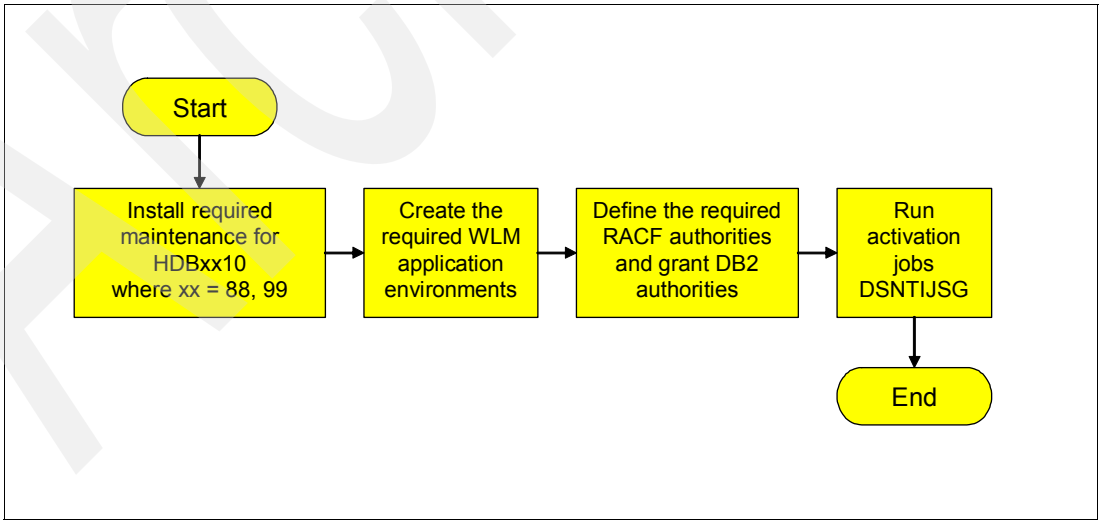

Figure 24-1 Steps for enablement

Step 1: Install the required maintenance for HDBxx10

Table 24-12 lists all required PTF/ and PAR numbers to enable the new Administrative enablement, the task scheduler, and the common SQL API procedures that are discussed later in this chapter in detail.

Table 24-12 PTF numbers for stored procedures

PTF number	DB2 version	Stored procedures contained
UK32046	V8	**Administrative enablement stored procedures** - ADMIN_COMMAND_DB2 - ADMIN_INFO_SSID **Task scheduler stored procedures / UDFs** - ADMIN_TASK_ADD - ADMIN_TASK_REMOVE - ADMIN_TASK_LIST - ADMIN_TASK_STATUS
UK32047 (API updated in this PTF)	V9	**Task scheduler stored procedures / UDFs** - ADMIN_TASK_ADD - ADMIN_TASK_REMOVE - ADMIN_TASK_LIST - ADMIN_TASK_STATUS
UK32059	V8	**Administrative enablement stored procedures** - ADMIN_COMMAND_DSN - ADMIN_COMMAND_UNIX - ADMIN_JOB_xxxxxx - ADMIN_DS_xxxxxx - ADMIN_INFO_HOST - ADMIN_UTL_xxxxxx
UK32060	V9	**Administrative enablement stored procedures** - ADMIN_COMMAND_DSN - ADMIN_COMMAND_UNIX - ADMIN_JOB_xxxxxx - ADMIN_DS_xxxxxx - ADMIN_INFO_HOST - ADMIN_UTL_xxxxxx
UK32061	V8	**Common SQL API stored procedure** - GET_CONFIG - GET_MESSAGE - GET_SYSTEM_INFO
UK33845	V9	**Common SQL API stored procedure** - GET_CONFIG - GET_MESSAGE - GET_SYSTEM_INFO
UK32795	V9	**DSNACCOX**

Note: The PTF numbers listed in Table 24-12 reflect the most recent version of the respective procedures at the time the book was published. Check for other possible PTFs that might contain updates in the future.

Step 2: Create the required WLM application environment

All DB2-provided stored procedures run in a WLM application environment. Table 24-13 lists considerations for creating a minimal number of WLM application environments for the DB2-provided stored procedures based on similar characteristics of the stored procedures. The intent is to minimize the number of address spaces. If you are activating only a few types of DB2-supplied procedures or you have specific performance requirements, you might want to have more granularity in the WLM environment allocation.

In Table 24-13, if the procedure name is in bold then this procedure requires APF authorization. If the procedure name is suffixed with (*) then special DD statements are required in the WLM JCL or additional post-install jobs have to be submitted. In this case the column “Recommended WLM application environment name” contains further information. The column with DSNTIJSG lists the environment names created by default and added here for cross reference.

Table 24-13 WLM environment definitions for DB2 stored procedures

Stored procedure name	NUMTCB	Recommended WLM application environment name	DSNTIJSG application environment name
DSNUTILS(*) **DSNUTILU(*)** DSNWZP DSNWSPM **GET_SYSTEM_INFO**	Required: 1	DSNWLM_UTILS (* SYSIN and SYSPRINT must be allocated to work files)	WLMENV1

Stored procedure name	NUMTCB	Recommended WLM application environment name	DSNTIJSG application environment name
WLM_REFRESH(*) DSNACCOX ADMIN_INFO_SSID ADMIN_INFO_HOST ADMIN_COMMAND_DB2 ADMIN_UTL_SCHEDULE ADMIN_UTL_SORT **ADMIN_DS_BROWSE** **ADMIN_DS_WRITE** **ADMIN_DS_RENAME** **ADMIN_DS_DELETE** **ADMIN_DS_SEARCH** **ADMIN_DS_LIST** **DSNAIMS** **DSNAIMS2** DSNLEUSR **GET_CONFIG** GET_MESSAGE ADMIN_TASK_ADD ADMIN_TASK_REMOVE *ADMIN_TASK_LIST (UDF)* *ADMIN_TASK_STATUS (UDF)* SYSIBM.SQLCOLPRIVILEGES(**) SYSIBM.SQLCOLUMNS(**) SYSIBM.SQLFOREIGNKEYS(**) SYSIBM.SQLPRIMARYKEYS(**) SYSIBM.SQLPROCEDURECOLS(**) SYSIBM.SQLPROCEDURES(**) SYSIBM.SQLSPECIALCOLUMNS(**) SYSIBM.SQLSTATISTICS(**) SYSIBM.SQLTABLEPRIVILEGES(**) SYSIBM.SQLTABLES(**) SYSIBM.SQLGETTYPEINFO(**) SYSIBM.SQLUDTS(**) SYSIBM.SQLCAMESSAGE(**) SQLJ.INSTALL_JAR SQLJ.REPLACE_JAR SQLJ.REMOVE_JAR SQLJ.DB2_INSTALL_JAR SQLJ.DB2_REPLACE_JAR SQLJ.DB2_REMOVE_JAR SQLJ.DB2_UPDATEJARINFO SQLJ.ALTER_JAVA_PATH **DBG_RUNSESSIONMANAGER (***)**	Recommended: 40 - 60	DSNWLM_GENERAL *(* Requires RACF special resource profile of the form ssid.WLM_REFRESH.wlm env-name)* *(** Run post-install job DSNTIJMS for object creation)* *(*** PSMDEBUG DD required for tracing)*	WLMENV, WLMENV5, WLMENV6, WLMENV7 !WLM! in DSNTIJM !WLM! in DSNTIJI2
ADMIN_COMMAND_DSN DSNTBIND(*) DSNTPSMP(*)	Required: 1	DSNWLM_REXX *(* See sample address space proc in prefix SDSNSAMP.DSN8WLMP)*	REXX_WLMENV
DSNACICS(*)	Recommended: 25 - 40	DSNWLM_CICS *(* See DSNCICS address space proc in prefix.SDSNSAMP(DSNTIJMV)*	WLMENV2

Stored procedure name	NUMTCB	Recommended WLM application environment name	DSNTIJSG application environment name
XSR_COMPLETE	Recommended: 5 - 8	DSNWLM_JAVA	WLMENVJ
XSR_REGISTER XSR_ADDSCHEMADOC XSR_REMOVE XDBDECOMPXML XDBDECOMPXML100MB	Recommended: 40 - 60	DSNWLM_XML	WLMENV3
ADMIN_COMMAND_UNIX ADMIN_JOB_SUBMIT **ADMIN_JOB_FETCH** **ADMIN_JOB_QUERY** **ADMIN_JOB_CANCEL**	Recommended: 40 - 60	DSNWLM_PROGRAM_CONTROL *(RACF program controlled)*	WLMENV_RACFPC
DB2DEBUG.DEBUGGERLEVEL DB2DEBUG.CREATE_SESSION DB2DEBUG.DESTROY_SESSION DB2DEBUG.QUERY_SESSION DB2DEBUG.LIST_SESSION DB2DEBUG.PUT_COMMAND DB2DEBUG.GET_REPORT DBG_INITIALIZECLIENT DBG_TERMINATECLIENT DBG_SENDCLIENTREQUESTS DBG_SENDCLIENTCOMMANDS DBG_RECVCLIENTREPORTS DBG_ENDSESSIONMANAGER DBG_PINGSESSIONMANAGER DBG_LOOKUPSESSIONMANAGER	Recommended: 5 - 20	DSNWLM_DEBUGGER *(* For all listed procedures: Run post-install job DSNTIJSD for object creation)* *(PSMDEBUG DD required for tracing)* *(Frequent refresh when debug diagnostics are activated)*	!WLMENV! in DSNTIJSD
SYSPROC.OSC_EXECUTE_TASK **SYSPROC.EXPLAIN_SQL**	Recommended: 1	DSNWLM_JAVA_LARGEMEM	WLMENVJU in DSNTIJOS
DB2MQ.MQSEND DB2MQ.MQRECEIVE DB2MQ.MQRECEIVECLOB DB2MQ.MQREAD DB2MQ.MQREADCLOB DB2MQ.MQRECEIVEALL DB2MQ.MQRECEIVEALLCLOB DB2MQ.MQREADALL DB2MQ.MQREADALLCLOB	Recommended: 5 - 20	DSNWLM_MQSERIES	WLMENV8
DB2XML.SOAPHTTPCC DB2XML.SOAPHTTPNC DB2XML.SOAPHTTPNV DB2XML.SOAPHTTPVV	Recommended: 5 - 20	DSNWLM_WEB_SERVICES	WLMENV8

As a best practice recommendation, we tried to keep the number of different WLM application environments as low as possible. For the above listed DB2-supplied stored procedures we therefore recommend to create at least eleven different WLM application environments per DB2 subsystem. The following listing contains some more information about the WLM application environments:

- **DSNWLM_UTILS** - It is required that online utility execution stored procedures DSNUTILS and DSNUTILU run in a WLM application environment where NUMTCB = 1. Both procedures use data sets that are allocated in the JCL procedure for the WLM application environment. If more than one instance of DSNUTILS were allowed to run in the same address space, the instances would overwrite each other's data sets, leading to indeterministic behavior. Be aware that SYSIN and SYSPRINT must be allocated to work files, in the WLM JCL.

  Example 24-1 contains a sample startup procedure for the DSNWLM_UTILS WLM application environment.

Example 24-1 Sample startup procedure for DSNWLM_UTILS

```
//*****************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* DB2 UTILITIES STORED PROCEDURES AND OTHER ROUTINES WITH SIMILAR
//* RUNTIME REQUIREMENTS.
//*    APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*             SUPPORTED BY THIS JCL PROCEDURE.
//*    DB2SSN : THE DB2 SUBSYSTEM NAME.
//*    RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//*    NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*             FOR THIS WLM ENVIRONMENT, ALWAYS SPECIFY A NUMTCB OF 1.
//*
//* NOTES:
//* (1) ALL LIBRARIES IN THE STEPLIB CONCATENATION NEED TO BE APF-
//*     AUTHORIZED
//* (2) THE SYSIN AND SYSPRINT DD STATEMENTS ARE REQUIRED AND NEED TO
//*     BE ALLOCATED TO WORK FILES
//* (3) THE UTPRINT, RNPRIN01, AND DSSPRINT DD STATEMENTS ARE ALSO
//*     REQUIRED
//*
//*****************************************************************
//DSNWLMU PROC APPLENV=DSNWLM_UTILS,
//            DB2SSN=DSN,RGN=OK,NUMTCB=1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//            PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB   DD DISP=SHR,DSN=prefix.SCEERUN
//          DD DISP=SHR,DSN=prefix.SDSNEXIT
//          DD DISP=SHR,DSN=prefix.SDSNLOAD
//UTPRINT   DD SYSOUT=*
//RNPRIN01  DD SYSOUT=*
//DSSPRINT  DD SYSOUT=*
//SYSIN     DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSPRINT  DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
```

  Note: Be aware of the following special required DD statements:

  - SYSIN - Allocates a work file for temporarily storing utility input statements.
  - SYSPRINT - Allocates a work file for temporarily storing utility output messages.
  - RNPRIN01 - Allocates a data set for messages from DFSORT™. Required only if you plan to invoke RUNSTATS and collect distribution statistics.
  - UTPRINT - Allocates a data set for messages from DFSORT.
  - DSSPRINT - Allocates a data set for messages when making concurrent copies.

- **DSNWLM_GENERAL** - This is for most DB2-supplied stored procedures. Notice that some procedures require APF authorization.

Note: ADMIN_UTL_SCHEDULE is a complex stored procedure that creates up to 99 parallel threads to execute DB2 online utilities. Therefore, this WLM application environment has to be created with at least NUMTCB > 1.

Example 24-2 shows a sample startup procedure for the DSNWLM_GENERAL WLM application environment.

Example 24-2 Sample startup procedure for DSNWLM_GENERAL

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* MOST DB2-SUPPLIED STORED PROCEDURES AND USER-DEFINED FUNCTIONS.
//*    APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*             SUPPORTED BY THIS JCL PROCEDURE.
//*    DB2SSN : THE DB2 SUBSYSTEM NAME.
//*    RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//*    NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*             FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 40 - 60.
//*
//* NOTES:
//*    ALL LIBRARIES IN THE STEPLIB CONCATENATION NEED TO BE APF-
//*    AUTHORIZED.
//*
//*********************************************************************
//DSNWLMG PROC APPLENV=DSNWLM_GENERAL,
//            DB2SSN=DSN,RGN=0K,NUMTCB=40
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//            PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=prefix.SCEERUN
//        DD DISP=SHR,DSN=prefix.SDSNEXIT
//        DD DISP=SHR,DSN=prefix.SDSNLOAD
```

- **DSNWLM_REXX** - ADMIN_COMMAND_DSN, DSNTBIND, and DSNTPSMP are REXX stored procedures, which must run in a WLM application environment where NUMTCB = 1.

Example 24-3 contains a sample startup procedure for the DSNWLM_REXX WLM application environment.

Example 24-3 Sample startup procedure for DSNWLM_REXX

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* DB2-SUPPLIED REXX STORED PROCEDURES.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, ALWAYS SPECIFY A NUMTCB OF 1.
//*
//* NOTES:
//* (1) AT LEAST ONE LIBRARY IN THE STEPLIB CONCATENATION NEEDS TO BE
//*     NON-APF AUTHORIZED.
//* (2) FOR DSNTPSMP, NUMTCB=1 IS REQUIRED. SPECIFY NO OTHER VALUE.
//*     THIS ASSURES CONCURRENT EXECUTIONS OF DSNTPSMP WILL RUN IN
//*     THEIR OWN ADDRESS SPACE, WHICH IS NEEDED FOR PROPER DATASET
//*     OPERATION FROM WITHIN A REXX/TSO DB2 STORED PROCEDURE.
//*
//*********************************************************************
//DSNWLMR PROC APPLENV=DSNWLM_REXX,
```

```
//        DB2SSN=DSN,RGN=OK,NUMTCB=1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//             DYNAMNBR=5,                      <== Allow for Dyn Allocs
//             PARM='&DB2SSN,1,&APPLENV'        <== Use 1, not NUMTCB
//*
//NUMTCB@1 SET NUMTCB=                          <== Null NUMTCB symbol
//*
//* Include SDSNEXIT to use Secondary Authids (DSN3@ATH DSN3@SGN exits)
//STEPLIB  DD DISP=SHR,DSN=prefix.RUNLIB.LOAD
//         DD DISP=SHR,DSN=CBC!!.SCCNCMP        <== C Compiler
//         DD DISP=SHR,DSN=prefix.SCEERUN       <== LE runtime
//         DD DISP=SHR,DSN=prefix.SDSNEXIT
//         DD DISP=SHR,DSN=prefix.SDSNLOAD
//SYSEXEC  DD DISP=SHR,                         <== Location of DSNTPSMP
//         DSN=DSN!!0.SDSNCLST
//SYSTSPRT DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSABEND DD DUMMY
//DSNTRACE DD SYSOUT=*
//*
//**** Data sets required by the SQL Procedures Processor
//SQLDBRM DD DISP=SHR,                          <== DBRM Library
//        DSN=DSN!!0.DBRMLIB.DATA
//SQLCSRC DD DISP=SHR,                          <== Generated C Source
//        DSN=DSN!!0.SRCLIB.DATA
//SQLLMOD DD DISP=SHR,                          <== Application Loadlib
//        DSN=DSN!!0.RUNLIB.LOAD
//SQLLIBC DD DISP=SHR,                          <== C header files
//        DSN=CEE!!.SCEEH.H
//        DD DISP=SHR,
//        DSN=CEE!!.SCEEH.SYS.H
//        DD DISP=SHR, <== Debug header file
//        DSN=DSN!!0.SDSNC.H
//SQLLIBL DD DISP=SHR, <== Linkedit includes
//        DSN=CEE!!.SCEELKED
//        DD DISP=SHR,
//        DSN=DSN!!0.SDSNLOAD
//SYSMSGS DD DISP=SHR, <== Prelinker msg file
//        DSN=CEE!!.SCEEMSGP(EDCPMSGE)
//*
//****         DSNTPSMP Configuration File - CFGTPSMP (optional)
//*            A site provided sequential dataset or member, used to
//*            define customized operation of DSNTPSMP in this APPLENV.
//*CFGTPSMP DD DISP=SHR,DSN=
//*
//**** Workfiles required by the SQL Procedures Processor
//SQLSRC   DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//         DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLPRINT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//         DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLTERM DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//        DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLOUT  DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//        DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLCPRT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//        DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLUT1  DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//        DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLUT2  DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
```

```
//        DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLCIN  DD UNIT=SYSALLDA,SPACE=(32000,(20,20))
//SQLLIN  DD UNIT=SYSALLDA,SPACE=(3200,(30,30)),
//        DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SQLDUMMY DD DUMMY
//SYSMOD  DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//        DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
```

Note: Be aware of the following special required DD statements:

- SYSEXEC - Allocates the prefix.SDSNCLST library. This data set is where the DB2-supplied REXX routines reside.
- SYSTSPRT - Allocates an output data destination for messages from the TSO/E command processor.
- SQLDBRM - Allocates the data set for DBRMs created by running the DSNTPSMP stored procedure, and for DBRMs for packages bound by running the DSNTPSMP and DSNTBIND stored procedures.
- SQLCSRC - Allocates the data set for storing generated C code created by running DSNTPSMP.
- SQLLMOD - Allocates the data set for storing load modules created by running DSNTPSMP.
- SQLLIBC - Allocates C header files used by DSNTPSMP for compiling SQL procedures.
- SQLLIBL - Allocates include files used by DSNTPSMP for link editing external modules for SQL procedures.
- SYSMSGS - Allocates the message file for the IBM Language Environment pre-link editor.
- CFGTPSMP (optional) - Allocates a site-provided sequential data set or member, used to define customized operation of DSNTPSMP in this application environment.
- SQLSRC - Allocates a work file used by DSNTPSMP. This work file needs to have a fixed block record format, a logical record length of 80, and a block size of 23440.
- SQLPRINT - Allocates a work file used by DSNTPSMP. This work file needs to have a variable block record format, a logical record length of 137, and a block size of 23476.
- SQLTERM - Allocates a work file used by DSNTPSMP. This work file needs to have a variable block record format, a logical record length of 137, and a block size of 23476.
- SQLOUT - Allocates a work file used by DSNTPSMP. This work file needs to have a variable block record format, a logical record length of 137, and a block size of 23476.
- SQLCPRT - Allocates a work file used by DSNTPSMP. This work file needs to have a variable block record format, a logical record length of 137, and a block size of 23476.
- SQLUT1 - Allocates a work file used by DSNTPSMP. This work file needs to have a fixed block record format, a logical record length of 80, and a block size of 23440.
- SQLUT2 - Allocates a work file used by DSNTPSMP. This work file needs to have a fixed block record format, a logical record length of 80, and a block size of 23440.
- SQLCIN - Allocates a work file used by DSNTPSMP. Do not specify a DCB for allocating this work file.
- SQLLIN - Allocates a work file used by DSNTPSMP. This work file needs to have a fixed block record format, a logical record length of 80, and a block size of 3200.
- SYSMOD - Allocates a work file used by DSNTPSMP. This work file needs to have a fixed block record format, a logical record length of 80, and a block size of 23440.

– SQLDUMMY - Required but needs to be allocated to DUMMY.

- **DSNWLM_CICS** - Only for DSNACICS. If CICS 4.1 is to be called by DSNACICS, NUMTCB must not exceed 25. If a newer version is to be called, the recommended NUMTCB value is 40 and must not exceed 100.

  Example 24-4 contains a sample startup procedure for the DSNWLM_CICS WLM application environment.

Example 24-4 Sample startup procedure for DSNWLM_CICS

```
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* THE DSNACICS STORED PROCEDURE.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 25 - 100.
//*          THE RECOMMENDED VALUE IS 40.
//*
//* NOTES:
//* (1) ALL LIBRARIES IN THE STEPLIB CONCATENATION NEED TO BE APF-
//*     AUTHORIZED
//* (2) THE STEPLIB DD MUST ALLOCATE THE CICS SDFHEXCI LIBRARY
//* (3) UNCOMMENT THE DSNDUMP DD STATEMENT BELOW TO CAUSE THE DSNACICS
//*     STORED PROCEDURE TO TAKE AN SVC WHENEVER IT IS GOING TO
//*     GENERATE AN ERROR MESSAGE.
//*
//*********************************************************************
//DSNWLMC PROC APPLENV=DSNWLM_DSNACICS,
//            DB2SSN=DSN,RGN=OK,NUMTCB=40
//*
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//            PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB   DD DISP=SHR,DSN=prefix.SDFHEXCI
//         DD DISP=SHR,DSN=prefix.SCEERUN
//         DD DISP=SHR,DSN=prefix.SDSNEXIT
//         DD DISP=SHR,DSN=prefix.SDSNLOAD
//*DSNDUMP  DD SYSOUT=A
```

Note: Be aware of the following special required DD statement:

– DSNDUMP - In normal use, this DD is commented out. Uncomment it to cause DSNACICS to take an SVC dump whenever it is going to generate an error message. When used, allocate this DD to output class for system dumps.

- **DSNWLM_JAVA -** For DB2-supplied Java stored procedures.

  Example 24-5 shows a sample startup procedure for the DSNWLM_JAVA WLM application environment.

Example 24-5 Sample startup procedure for DSNWLM_JAVA

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* DB2-SUPPLIED JAVA STORED PROCEDURES AND USER-DEFINED FUNCTIONS.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 5 - 8.
```

```
//*
//* NOTES:
//* (1) ALL LIBRARIES IN THE STEPLIB CONCATENATION NEED TO BE APF-
//*     AUTHORIZED
//* (2) THE JAVAENV DD STATEMENT IS REQUIRED AND NEEDS TO BE ALLOCATED
//*     TO THE IBM LANGUAGE ENVIRONMENT RUN-TIME OPTIONS DATA SET FOR
//*     JAVA STORED PROCEDURES AND UDFS THAT USE THIS WLM ENVIRONMENT.
//*     THIS DATA SET IS CREATED BY DB2 INSTALLATION JOB DSNTIJMV.
//*     FOR FURTHER INFORMATION ABOUT THE JAVAENV DATA SET, SEE THE
//*     DISCUSSION OF RUN-TIME ENVIRONMENTS FOR JAVA ROUTINES IN THE
//*     DB2 FOR Z/OS APPLICATION PROGRAMMING GUIDE AND REFERENCE FOR
//*     JAVA.
//* (3) THE JSPDEBUG DD STATEMENT IS ALSO REQUIRED.
//*
//*********************************************************************
//DSNWLMJ PROC APPLENV=DSNWLM_JAVA,
//            DB2SSN=DSN,RGN=0K,NUMTCB=5
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//            PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB   DD DISP=SHR,DSN=prefix.SCEERUN
//          DD DISP=SHR,DSN=prefix.SDSNEXIT
//          DD DISP=SHR,DSN=prefix.SDSNLOAD
//JAVAENV   DD DISP=SHR,DSN=prefix.DSNWLMJ.JAVAENV
//JSPDEBUG  DD SYSOUT=*
```

Note: Be aware of the following special required DD statements:

- JAVAENV - Allocates a data set that contains Language Environment runtime options for Java stored procedures. The presence of this DD statement indicates to DB2 that the WLM environment is for Java routines. For an interpreted Java routine, this data set must contain the environment variable JAVA_HOME, which indicates to DB2 that the WLM environment is for interpreted Java routines. JAVA_HOME also specifies the highest-level directory in the set of directories that contain the Java SDK.
- JSPDEBUG - Allocates a data set in which Language Environment puts runtime diagnostics.

- **DSNWLM_XML** - For XML validate stored procedures (high memory requirements).

Example 24-6 contains a sample startup procedure for the DSNWLM_XML WLM application environment.

Example 24-6 Sample startup procedure for DSNWLM_XML

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* MOST DB2-SUPPLIED STORED PROCEDURES AND USER-DEFINED FUNCTIONS FOR
//* XML PROCESSSING.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 40 - 60.
//*
//* NOTES:
//* ALL LIBRARIES IN THE STEPLIB CONCATENATION NEED TO BE APF-
//* AUTHORIZED.
//*
//*********************************************************************
//DSNWLMX PROC APPLENV=DSNWLM_GENERAL,
//            DB2SSN=DSN,RGN=0K,NUMTCB=40
```

```
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//            PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB   DD DISP=SHR,DSN=prefix.SCEERUN
//          DD DISP=SHR,DSN=prefix.SDSNEXIT
//          DD DISP=SHR,DSN=prefix.SDSNLOAD
```

- **DSNWLM_PROGRAM_CONTROL** - For DB2-supplied stored procedures that must be defined to RACF program control.

  Example 24-7 contains a sample startup procedure for the DSNWLM_PROGRAM_CONTROL WLM application environment.

Example 24-7 Sample startup procedure for DSNWLM_PROGRAM_CONTROL

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* DB2-SUPPLIED STORED PROCEDURES AND USER-DEFINED FUNCTIONS THAT
//* ARE REGISTERED TO RACF PROGRAM CONTROL OR EQUIVALENT.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 40 - 60.
//*
//* NOTES:
//* (1) ALL LIBRARIES IN THE STEPLIB CONCATENATION NEED TO BE APF-
//*     AUTHORIZED.
//* (2) THIS WLM ENVIRONMENT IS FOR EXCLUSIVE USE OF STORED PROCEDURES
//*     AND UDFS THAT ARE DEFINED TO PROGRAM CONTROL.
//*
//*********************************************************************
//DSNWLMP PROC APPLENV=DSNWLM_PROGRAM_CONTROL,
//            DB2SSN=DSN,RGN=0K,NUMTCB=40
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//            PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB     DD DISP=SHR,DSN=prefix.SCEERUN
//            DD DISP=SHR,DSN=prefix.SDSNEXIT
//            DD DISP=SHR,DSN=prefix.SDSNLOAD
```

- **DSNWLM_DEBUGGER** - For debugging SQL stored procedures. There are a couple of special requirements for this WLM environments. Chapter 15, “Native SQL procedures” on page 253 contains a detailed description of these requirements.

  Example 24-8 shows a sample startup procedure for the DSNWLM_DEBUGGER WLM application environment.

Example 24-8 Sample startup procedure for DSNWLM_DEBUGGER

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING THE
//* STORED PROCEDURES FOR THE UNIFIED DEBUGGER.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 5 - 20.
//*
//* NOTES:
//* THERE IS NO SPECIAL APF REQUIREMENT FOR THIS WLM ENVIRONMENT.
```

```
//*
//*********************************************************************
//DSNWLMD PROC APPLENV=DSNWLM_DEBUGGER,
//        DB2SSN=DSN,RGN=OK,NUMTCB=5
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=prefix.SCEERUN
//        DD DISP=SHR,DSN=prefix.SDSNEXIT
//        DD DISP=SHR,DSN=prefix.SDSNLOAD
//SYSIN   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
```

Note: The environment is frequently refreshed when debug diagnostics are activated.

- **DSNWLM_JAVA_LARGEMEM** - Similar to DSNWLM_JAVA, but specifies additional DDs and a maximum TCB setting of 1 to account for high memory usage by the DB2-supplied Java routines assigned to run in it.

  Example 24-9 contains a sample startup procedure for the DSNWLM_JAVA_LARGEMEM WLM application environment.

Example 24-9 Sample startup procedure for DSNWLM_JAVA_LARGEMEM

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* DB2-SUPPLIED JAVA STORED PROCEDURES AND USER-DEFINED FUNCTIONS FOR
//* THE OPTIMIZATION SERVICE CENTER THAT USE LARGE AMOUNTS OF MEMORY.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, ALWAYS SPECIFY A NUMTCB OF 1.
//*
//* NOTES:
//* (1) ALL LIBRARIES IN THE STEPLIB CONCATENATION NEED TO BE APF-
//*     AUTHORIZED
//* (2) THE JAVAENV DD STATEMENT IS REQUIRED AND NEEDS TO BE ALLOCATED
//*     TO THE IBM LANGUAGE ENVIRONMENT RUNTIME OPTIONS DATA SET FOR
//*     JAVA STORED PROCEDURES AND UDFS THAT USE THIS WLM ENVIRONMENT.
//*     THIS DATA SET IS CREATED BY DB2 INSTALLATION JOB DSNTIJMV.
//*     FOR FURTHER INFORMATION ABOUT THE JAVAENV DATA SET, SEE THE
//*     DISCUSSION OF RUN-TIME ENVIRONMENTS FOR JAVA ROUTINES IN THE
//*     DB2 FOR Z/OS APPLICATION PROGRAMMING GUIDE AND REFERENCE FOR
//*     JAVA.
//* (3) THE JSPDEBUG DD STATEMENT IS ALSO REQUIRED.
//* (4) THE JAVAOUT AND JAVAERR DD STATEMENTS ARE ALSO REQUIRED AND
//*     NEED TO BE ALLOCATED TO A VALID HFS PATH UNDER AN EXISTING
//*     DIRECTORY STRUCTURE.
//*
//*********************************************************************
//DSNWLMO  PROC APPLENV=DSNWLM_JAVA_LARGEMEM,
//         DB2SSN=DSN,RGN=OK,NUMTCB=1
//IEFPROC  EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//         PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD DISP=SHR,DSN=prefix.SCEERUN
//         DD DISP=SHR,DSN=prefix.SDSNEXIT
//         DD DISP=SHR,DSN=prefix.SDSNLOAD
//JAVAENV  DD DISP=SHR,DSN=prefix.DSNWLMO.JAVAENV
//JSPDEBUG DD SYSOUT=*
//JAVAOUT  DD PATH='/V1R7/USR/db2a1O/JAVAOUT.TXT',
```

```
//            PATHOPTS=(ORDWR,OCREAT,OAPPEND),
//            PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
//JAVAERR  DD PATH='/V1R7/USR/db2a10/JAVAERR.TXT',
//         PATHOPTS=(ORDWR,OCREAT,OAPPEND),
//         PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
```

Note: Be aware of the following special required DD statements:

- JAVAENV - Allocates a data set that contains Language Environment runtime options for Java stored procedures. The presence of this DD statement indicates to DB2 that the WLM environment is for Java routines. For an interpreted Java routine, this data set must contain the environment variable JAVA_HOME. This environment variable indicates to DB2 that the WLM environment is for interpreted Java routines. JAVA_HOME also specifies the highest-level directory in the set of directories that containing the Java SDK.
- JSPDEBUG - Allocates a data set in which Language Environment puts runtime diagnostics.
- JAVAOUT - Allocates an HFS file for diagnostic output from OSC Java routines.
- JAVAERR - Allocates an HFS file for error output from OSC Java routines.

- **DSNWLM_MQSERIES** - For running DB2 MQSeries functions.

Example 24-10 contains a sample startup procedure for the DSNWLM_MQSERIES WLM application environment.

Example 24-10 Sample startup procedure for DSNWLM_MQSERIES

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* THE MQSERIES FUNCTIONS.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 10 OR
//*          MORE.
//*
//* NOTES:
//* THERE IS NO SPECIAL APF REQUIREMENT FOR THIS WLM ENVIRONMENT.
//*
//* UNCOMMENT THE DD STATEMENTS UNDER STEPLIB FOR THE MQSERIES
//* SCSQLOAD, SCSQAUTH, AND SCSQANLE RUNTIME LIBRARIES IF THEY
//* ARE NOT INCLUDED IN YOUR SYSTEM LINKLIST. YOU ALSO NEED TO
//* EDIT THE DATA SET NAME PREFIXES FOR THESE LIBRARIES.
//*
//*********************************************************************
//DSNWLMM PROC APPLENV=DSNWLM_MQSERIES,
//        DB2SSN=DSN,RGN=0K,NUMTCB=10
//*
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=prefix.SCEERUN
//        DD DISP=SHR,DSN=prefix.SDSNEXIT
//        DD DISP=SHR,DSN=prefix.SDSNLOAD
//* Uncomment the following if not included in the LINKLIST
//*       DD DISP=SHR,DSN=prefix.SCSQLOAD
//*       DD DISP=SHR,DSN=prefix.SCSQAUTH
//*       DD DISP=SHR,DSN=prefix.SCSQANLE
```

- **DSNWLM_WEB_SERVICES -** For running DB2 Web Services and SOAP functions.

  Example 24-11 contains a sample startup procedure for the DSNWLM_WEB_SERIES WLM application environment.

Example 24-11 Sample startup procedure for DSNWLM_WEB_SERIES

```
//*********************************************************************
//* JCL FOR STARTING THE WLM-ESTABLISHED ADDRESS SPACE FOR RUNNING
//* THE DB2 WEB SERVICES/SOAP FUNCTIONS.
//* APPLENV: THE MVS WLM APPLICATION ENVIRONMENT
//*          SUPPORTED BY THIS JCL PROCEDURE.
//* DB2SSN : THE DB2 SUBSYSTEM NAME.
//* RGN    : THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//* NUMTCB : THE NUMBER OF TCBS USED TO PROCESS END USER REQUESTS.
//*          FOR THIS WLM ENVIRONMENT, SPECIFY A NUMTCB OF 10 OR
//*          MORE.
//*
//* NOTES:
//* THERE IS NO SPECIAL APF REQUIREMENT FOR THIS WLM ENVIRONMENT.
//*
//* THE WSERROR DD STATEMENT IS REQUIRED AND NEEDS TO BE ALLOCATED
//* TO A VALID HFS PATH UNDER AN EXISTING DIRECTORY STRUCTURE.
//*
//*********************************************************************
//DSNWLMW PROC APPLENV=DSNWLM_WEB_SERVICES,
//        DB2SSN=DSN,RGN=0K,NUMTCB=10
//*
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=prefix.SCEERUN
//        DD DISP=SHR,DSN=prefix.SDSNEXIT
//        DD DISP=SHR,DSN=prefix.SDSNLOAD
//WSERROR DD PATH='/tmp/wsc.err',
//           PATHOPTS=(ORDWR,OCREAT,OAPPEND),
//           PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
```

  Note: Be aware of the following special required DD statement:

  – WSERROR - Allocates an HFS file for diagnostic output from the DB2 Web Services functions.

To minimize the overhead required to start and manage WLM application environment address spaces, we recommend that you create the WLM application environments DSNWLM_GENERAL / DSNWLM_XML / DSNWLM_PROGRAM_CONTROL / DSNWLM_CICS with NUMTCB = 40 - 60, and assign the respective listed stored procedures to them. However, in a very CPU-constrained environment, the recommendation is to use a lower number, such as 5 - 20.

Step 3: Define the required RACF authorities and grant DB2 authorities

To execute the CALL statement, the owner of the package or plan that contains the CALL statement must have one or more of the following privileges on each package that the stored procedure uses:

- The EXECUTE privilege on the respective package
- Ownership of the package
- PACKADM authority for the package collection
- SYSADM authority

In addition to the required authorities to execute a stored procedure, the following RACF authorities and DB2 authorities need to be granted.

Authorities for command execution stored procedures

ADMIN_COMMAND_UNIX - This procedure employs the __login() function to switch users. Therefore the load module for this procedure DSNADMCU has to run RACF program-controlled, the executing WLM address space has to be defined accordingly. The new user ID must have the appropriate authority to execute the USS command.

ADMIN_COMMAND_DSN - To execute the DSN subcommand, you must use a privilege set that includes the authorization to execute the respective subcommand, as described in *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844.

ADMIN_COMMAND_DB2 - This procedure can be used to issue any DB2 command. Hence, it will require the corresponding system privilege, such as the DISPLAY system privilege to issue a -DISPLAY BUFFERPOOL. Privileges required for other DB2 commands can be found in *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844.

Authorities for job management stored procedures

ADMIN_JOB_FETCH, ADMIN_JOB_SUBMIT, ADMIN_JOB_CANCEL, and ADMIN_JOB_QUERY use the __login() function to switch users. This requires that all these procedures are defined to RACF program control. The executing WLM address space has to be defined accordingly.

Authorities for data set management stored procedures

The data set manipulation stored procedures require the access authorities that are in place for the data set through RACF data set profiles. In addition to this, all these stored procedures need to run APF-authorized.

Authorities for system administration stored procedures

ADMIN_INFO_SSID - No additional authorization required.

ADMIN_INFO_HOST - The ADMIN_INFO_HOST stored procedure internally calls the ADMIN_COMMAND_DB2 stored procedure to execute the following DB2 commands:

- -DISPLAY DDF
- -DISPLAY GROUP

The owner of the package or plan that contains the CALL ADMIN_INFO_HOST statement must also have the authorization required to execute the stored procedure ADMIN_COMMAND_DB2 and the specified DB2 commands. To determine the privilege or authority required to issue a DB2 command, see *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844.

DSNWZP / DSNWSPM - The primary authorization ID of the caller must have MONITOR1 and TRACE privileges to run this routine.

WLM_REFRESH - The user ID that calls WLM_REFRESH requires READ access to a general resource profile named !DSN!.WLM_REFRESH.!WLMENV! of class DSNR in order to refresh a WLM application environment. See the sample job DSNTEJ6W for details.

DSNACICS - The CICS server program that DSNACICS calls runs under the same user ID as DSNACICS. That user ID depends on the SECURITY parameter that you specify when you define DSNACICS. The DSNACICS caller also needs authorization from an external security system, such as RACF, to use CICS resources. See Part 2 of *DB2 Version 9.1 for z/OS Installation Guide,* GC18-9846.

DSNLEUSR - The owner of the package or plan that contains the CALL statement must also have INSERT authority on SYSIBM.USERNAMES.

DSNAIMS and DSNAIMS2 - Ensure that OTMA C/I is initialized. See *IMS Version 9: Open Transaction Manager Access Guide and Reference*, SC18-7829 for an explanation of the C/I initialization.

Authorities for utility execution stored procedures

ADMIN_UTL_SCHEDULE - The ADMIN_UTL_SCHEDULE stored procedure internally calls the following stored procedures:

- ADMIN_COMMAND_DB2, to execute the DB2 DISPLAY UTILITY command
- ADMIN_INFO_SSID, to obtain the subsystem ID of the connected DB2 subsystem
- ADMIN_UTL_SORT, to sort objects into parallel execution units
- DSNUTILU, to run the requested utilities

The owner of the package or plan that contains the CALL ADMIN_UTL_SCHEDULE statement must also have the authorization to execute these stored procedures and run the requested utilities. To determine the privilege or authority required to call DSNUTILU, see *DB2 Version 9.1 for z/OS Utility Guide an Reference,* SC18-9855.

ADMIN_UTL_SORT - The owner of the package or plan that contains the CALL statement must also have SELECT authority on the following catalog tables:

- SYSIBM.SYSTABLEPART
- SYSIBM.SYSINDEXPART
- SYSIBM.SYSINDEXES
- SYSIBM.SYSTABLES

DSNACCOX - The user ID that calls DSNACCOX must have SELECT authority on the real-time statistics tables and the DISPLAY system privilege.

DSNUTILS or DSNUTILU - The user ID that runs a utility through DSNUTILS or DSNUTILU must have authorization to run the specified utility.

Authorities for task scheduler stored procedures

ADMIN_TAKS_ADD / ADMIN_TASK_REMOVE - Both procedures internally call the stored procedure SYSPROC.DSNWZP. Be aware that the MONITOR1 and TRACE privileges required for a successful DSNWZP CALL are inherited to the calling userID of the task scheduler procedures.

Users with SYSOPR, SYSCTRL, or SYSADM authority can remove any task. Other users who have EXECUTE privileges on this stored procedure can remove tasks that they added. Attempting to remove a task that was added by a different user returns an error in the output parameters return-code and message. Refer to “Removing a scheduled task” on page 575 for further details.

Authorities for Common SQL API stored procedures

GET_CONFIG or **GET_MESSAGE** or **GET_SYSTEM_INFO** - There is no special authorization required to successfully invoke all three stored procedures. A calling user-ID only requires execution privileges to be granted.

APF-authorized

Some stored procedures need to run APF-authorized. In this case, ensure that all data sets in the STEPLIB DD concatenation of the WLM JCL are added to the APF authorization list. If only one of these data sets is not APF-authorized, all other data sets in the same STEPLIB concatenation are treated non-APF-authorized as well.

Refer to 24.1.2, “Setting up DB2-supplied stored procedures” on page 502 to obtain the APF authorization requirement for the respective DB2-supplied stored procedure.

We did not split DSNWLM_GENERAL into two WLM application environments separating the APF from non-APF-authorized running stored procedures, but rather used only one. This is because we wanted to create as few WLM application environments as possible for the best practice recommendations.

Program controlled

Several stored procedures described in this book execute the __login() function to switch users. If the BPX.DAEMON facility class is active and the BPX.DAEMON.HFSCTL facility class is not defined, the modules that implement these stored procedures must be defined to RACF program control. Otherwise, the following error will be returned:

```
EDC5139I Operation not permitted
```

Once these modules are defined to RACF program control, they must be loaded into a WLM-established stored procedure address space that only loads controlled programs. This is the reason why we specified a different WLM application environment for these stored procedures, instead of using DSNWLM_GENERAL.

You can define programs from traditional libraries to program control or define the BPX.DAEMON.HFSCTL profile in the facility class so that programs that are loaded from MVS libraries are not checked for program control. If the BPX.DAEMON.HFSCTL FACILITY class profile has been set up, you can choose to run the stored procedure that requires RACF program control (for example ADMIN_COMMAND_UNIX, ADMIN_JOB_SUBMIT) in the WLM application environment where the other administration enablement stored procedures are running (for example DSNWLM_GENERAL). Otherwise, they must run in a different WLM environment, such as DSNWLM_PROGRAM_CONTROL.

To define programs from traditional libraries to program control, you need to:

1. Activate the RACF program control (both access control to load modules and program access to data sets):

   ```
   SETROPTS WHEN(PROGRAM)
   ```

2. Define one of the following profiles:

   a. For a particular program, define a discrete RACF PROGRAM class profile:

      ```
      RDEFINE PROGRAM membername ADDMEM( 'datasetname' / volser/NOPADCHK) UACC(READ)
      ```

      The following members of SDSNLOAD must be program-controlled:

      - SDSNLOAD(DSNX9WLM)
      - SDSNLOAD(DSNX9SPA)
      - SDSNLOAD(DSNARRS)
      - SDSNLOAD(DSN3ID00)
      - SDSNLOAD(DSNX9WLS)
      - SDSNLOAD(DSNADMJS)
      - SDSNLOAD(DSNADMJP)
      - SDSNLOAD(DSNADMJQ)
      - SDSNLOAD(DSNADMJF)
      - SDSNLOAD(DSNADMCU)

   b. For all members in a data set:

      ```
      RDEFINE PROGRAM * ADDMEM( 'datasetname' / volser / NOPADCHK) UACC(READ)
      ```

3. Refresh the in-storage copy of the PROGRAM profile:

   ```
   SETROPTS WHEN(PROGRAM) REFRESH
   ```

To set up the BPX.DAEMON.HFSCTL FACILITY class, you need to:

1. Define the resource profile:

```
RDEFINE FACILITY BPX.DAEMON.HFSCTL UACC(NONE)
```

2. Give READ access to users:

```
PERMIT BPX.DAEMON.HFSCTL CLASS(FACILITY) ID(uuuuuu) ACCESS(READ) SETROPTS
RACLIST(FACILITY) REFRESH
```

In a production environment, it is recommended to define the required SDSNLOAD members to program control as an alternative to: RDEFINE FACILITY BPX.DAEMON.HFSCTL UACC (READ). The following listing shows a sample JCL that defines the required SDSNLOAD members to RACF program control.

Example 24-12 RACF program control JCL

```
//RACFJOB  JOB (ACCOUNT),'NAME',MSGCLASS=H,MSGLEVEL=(1,1),CLASS=A,
// NOTIFY=&SYSUID
//STEP01   EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
   SETROPTS WHEN(PROGRAM)
   RDEFINE PROGRAM DSNX9WLM ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNX9SPA ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNARRS  ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSN3IDOO ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNX9WLS ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNADMJS ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNADMJP ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNADMJQ ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNADMJF ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNADMCU ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   SETROPTS WHEN(PROGRAM) REFRESH
```

For detailed samples of RACF program control, refer to the installation job DSNTIJRA.

For more information on BPX.DAEMON and setting up program control, refer to *z/OS UNIX System Services Planning*, GA22-7800-03.

Refer to 24.1.2, “Setting up DB2-supplied stored procedures” on page 502 for the program-controlled requirement for the respective DB2-supplied stored procedure.

Step 4: Run the activation job DSNTIJSG

Edit and run the activation job DSNTIJSG following the instructions in its header. Also refer to the closing text in the respective PTF for additional information.

Step 5 (optional): Set up DRDA host connectivity

In order to connect from your client workstation to a DB2 subsystem, you have to start the Distributed Data Facility (DDF) at the host, and catalog the subsystem on your client. In order to catalog the DB2 for z/OS subsystem, you can enter the following commands from a DB2 command window:

```
db2 catalog tcpip node <node> remote <IP address> server <port>
db2 catalog dcs database <dbname> as <location>
db2 catalog database <dbname> as <db alias> at node <node> authentication DCS
```

You can find out the location, IP address, and TCP port number by issuing the DB2 command -DISPLAY DDF from an MVS console. In our tests we used the following commands to catalog the test system:

```
db2 catalog tcpip node TCPOOO1 remote wtsc63.itso.ibm.com server 12345
db2 catalog dcs database DB9A as DB9A
db2 catalog database DB9A as DB9A at node TCPOOO1 authentication DCS
```

Verify that your DB2 subsystem is cataloged correctly by connecting to it. Enter the following command from a DB2 command window:

```
db2 connect to <db alias> user <user> using <password>
```

You should see a message like the following with the version, release, and modification number of your DB2 subsystem:

```
Database Connection Information

 Database server        = DB2 OS/390 9.1.5
 SQL authorization ID   = PAOLOR3
 Local database alias   = DB9A
```

It is important that you now bind the applications and utilities on your DB2 subsystem. While still connected and in the DB2 command window, change to the SQLLIB\bnd directory and issue the following commands:

```
db2 bind @db2ubind.lst blocking all sqlerror continue grant public
db2 bind @db2cli.lst blocking all sqlerror continue grant public
```

You only have to bind the applications and utilities the first time you use a new client against a DB2 subsystem. Disconnect after you have successfully bound the utilities. For more information, refer to *DB2 Connect Version 9 User's Guide,* SC10-4229-00.

24.2 Administrative enablement stored procedure - details

In this section we describe in detail the new administration enablement stored procedures that are part of DB2 for z/OS Version 8 and 9. It contains the syntax diagrams, a description of the procedure options and the possible result sets. The corresponding Java samples that employ these procedures are in Appendix A, "Samples for using DB2-supplied stored procedures" on page 807.

The stored procedures are grouped according to the following functions:

- Command execution
- Job management
- Data set management
- System administration
- Utility execution

24.2.1 Command execution

The following procedures belong to this group:

- SYSPROC.ADMIN_COMMAND_UNIX
- SYSPROC.ADMIN_COMMAND_DSN
- SYSPROC.ADMIN_COMMAND_DB2

SYSPROC.ADMIN_COMMAND_UNIX

This stored procedure executes a z/OS UNIX System Services command and returns the output.

Load module name: DSNADMCU

Package name: DSNADMCU

Figure 24-2 illustrates the ADMIN_COMMAND_UNIX CALL syntax.

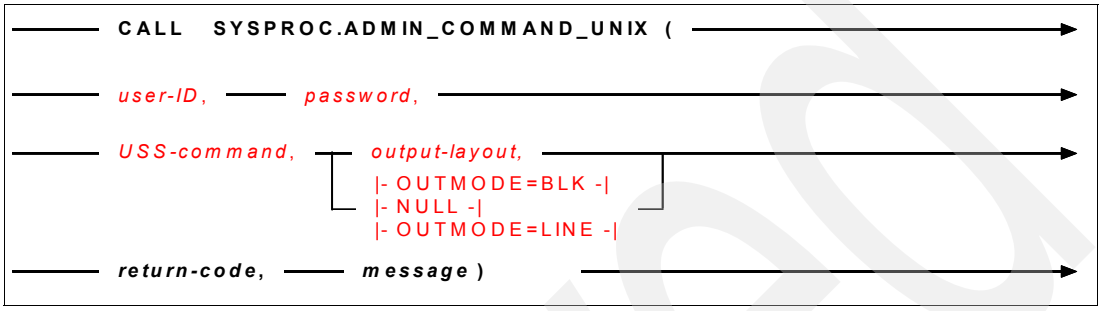

Figure 24-2 CALL ADMIN_COMMAND_UNIX stored procedure

ADMIN_COMMAND_UNIX option descriptions

- ***user-ID:*** Specifies the user ID under which the z/OS UNIX System Services command is issued.

  This is an input parameter of type VARCHAR(128) and cannot be null.

- ***password:*** Specifies the password associated with the input parameter `user-ID`.

  The value of the password is passed to the stored procedure as part of payload, and is not encrypted. It is not stored in dynamic cache when parameter markers are used.

  Recommendation: Have the application that invokes this stored procedure pass an encrypted password called a passticket.

  This is an input parameter of type VARCHAR(24) and cannot be null.

- ***USS-command:*** Specifies the z/OS UNIX System Services command to be executed.

  This is an input parameter of type VARCHAR(32704) and cannot be null.

- ***output-layout:*** Specifies how the output from the z/OS UNIX System Services command is returned. The output from the z/OS UNIX System Services command is a multi-line message. Possible values are:
  - OUTMODE=LINE - Each line is returned as a row in the result set.
  - OUTMODE=BLK - The lines are blocked into 32677 blocks and each block is returned as a row in the result set.

  If a null or empty string is provided, then the default option OUTMODE=BLK is used.

  This is an input parameter of type VARCHAR(1024).

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:
  - 0 - The call completed successfully.
  - 12 - The call did not complete successfully. The message output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- **message**: Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

This is an output parameter of type VARCHAR(1331).

ADMIN_COMMAND_UNIX output

This stored procedure returns the following output parameters:

- return-code
- message

In addition to the preceding output, the stored procedure returns one result set that contains the z/OS UNIX System Services command output messages.

Table 24-14 shows the format of the result set returned in the created global temporary table SYSIBM.USS_CMD_OUTPUT.

Table 24-14 Result set row for ADMIN_COMMAND_UNIX stored procedure

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
TEXT	VARCHAR(32677)	A block of text or a line from the output messages of a z/OS UNIX System Services command

SYSPROC.ADMIN_COMMAND_DSN

The SYSPROC.ADMIN_COMMAND_DSN stored procedure executes a BIND, REBIND, or FREE DSN subcommand and returns the output from the DSN subcommand execution.

Load module name: DSNADMCS

Package name: no package, this is a REXX procedure

Figure 24-3 illustrates the ADMIN_COMMAND_DSN CALL syntax.

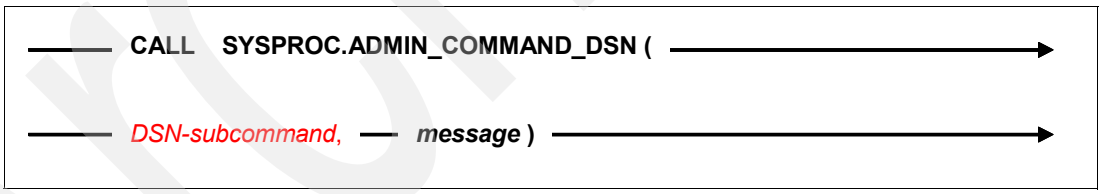

Figure 24-3 CALL ADMIN_COMMAND_DSN stored procedure

ADMIN_COMMAND_DSN option descriptions

- ***DSN-subcommand:*** Specifies the DSN subcommand to be executed. If the DSN subcommand passed to the stored procedure is not BIND, REBIND, or FREE, an error message is returned. The DSN subcommand is performed using the authorization ID of the user who invoked the stored procedure.

  This is an input parameter of type VARCHAR(32704) and cannot be null.

- ***message:*** Contains messages if an error occurs during stored procedure execution.

  Note: A blank message does not mean that the DSN subcommand completed successfully. The calling application must read the result set to determine if the DSN subcommand was successful or not.

  This is an output parameter of type VARCHAR(1331).

ADMIN_COMMAND_DSN output

This stored procedure returns an error message in the output parameter ***message*** in case an error occurs.

The stored procedure returns one result set that contains the DSN subcommand output messages. Table 24-15 shows the format of the result set returned in the created global temporary table SYSIBM.DSN_SUBCMD_OUTPUT.

Table 24-15 Result set row for ADMIIN_COMMAND_DSN stored procedure

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
TEXT	VARCHAR(255)	DSN subcommand output message line

SYSPROC.ADMIN_COMMAND_DB2

The SYSPROC.ADMIN_COMMAND_DB2 stored procedure executes one or more DB2 commands on a connected DB2 subsystem or on a DB2 data sharing group member and returns the command output messages.

Load module name: DSNADMCD

Package name: DSNADMCD

Figure 24-4 illustrates the ADMIN_COMMAND_DB2 CALL syntax.

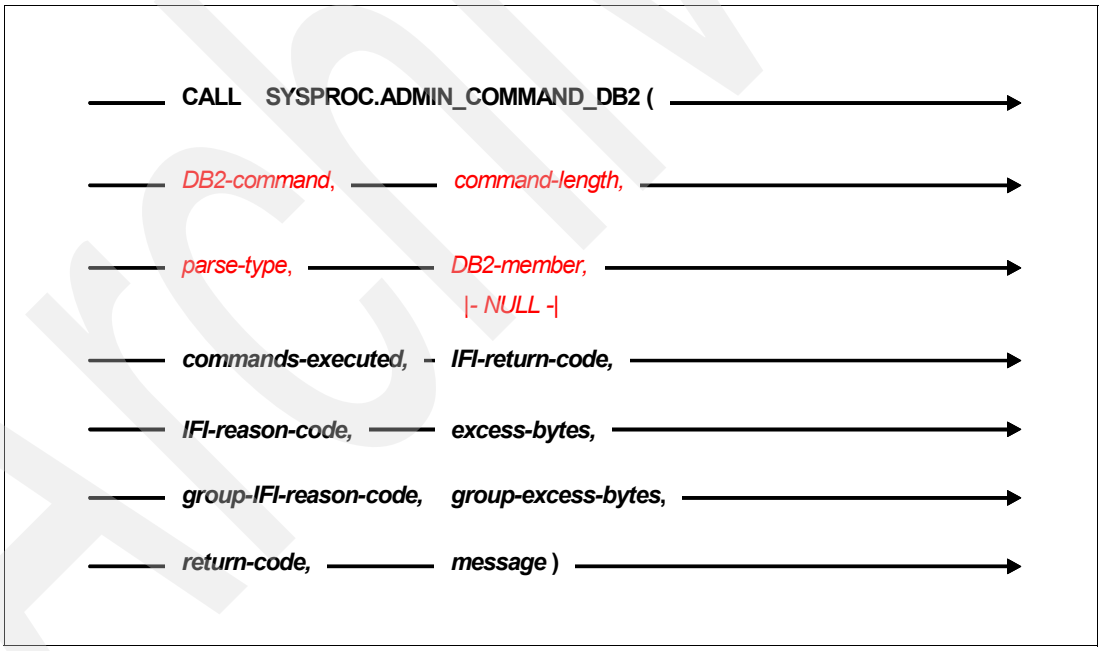

Figure 24-4 CALL ADMIN_COMMAND_DB2 stored procedure

ADMIN_COMMAND_DB2 option descriptions:

- ***DB2-command:*** Specifies any DB2 command such as -DISPLAY THREAD(*), or multiple DB2 commands. With multiple DB2 commands, use \0 to delimit the commands. The DB2 command is executed using the authorization ID of the user who invoked the stored procedure.

  This is an input parameter of type VARCHAR(32704) and cannot be null.

- ***command-length:*** Specifies the length of the DB2 command or commands. When multiple DB2 commands are specified in DB2-command, command-length is the sum of all of those commands, including the \0 command delimiters.

  This is an input parameter of type INTEGER and cannot be null.

- ***parse-type:*** Identifies the type of output message parsing requested.

  If you specify a parse type, ADMIN_COMMAND_DB2 parses the command output messages and provides the formatted result in a specific global temporary table. Possible values are:

  BP - parse “-DISPLAY BUFFERPOOL” command output messages

  DB - parse “-DISPLAY DATABASE” command output messages and return database information.

  TS - parse “-DISPLAY DATABASE(...) SPACENAM(...)” command output messages and return table spaces information.

  IX - parse “-DISPLAY DATABASE(...) SPACENAM(...)” command output messages and return index spaces information.

  THD - parse “-DISPLAY THREAD” command output messages.

  UT - parse “-DISPLAY UTILITY” command output messages.

  GRP - parse “-DISPLAY GROUP” command output messages.

  DDF - parse “-DISPLAY DDF” command output messages.

  Any other value - Do not parse any command output messages.

  Refer to “ADMIN_COMMAND_DB2 output” on page 526 for detailed information about the respective result set structures which are determined by *parse-type*.

  This is an input parameter of the type VARCHAR(3) and cannot be null.

- ***DB2-member:*** Specifies the name of a single data sharing group member on which an IFI request is to be executed.

  This is an input parameter of type VARCHAR(8).

- ***commands-executed:*** Provides the number of commands that were executed.

  This is an output parameter of type INTEGER.

- ***IFI-return-code:*** Provides the IFI return code.

  This is an output parameter of the type INTEGER.

- ***IFI-reason-code:*** Provides the IFI reason code.

  This is an output parameter of the type INTEGER.

- ***excess-bytes:*** Indicates the number of bytes that did not fit in the return area.

  This is an output parameter of the type INTEGER.

- ***group-IFI-reason-code:*** Provides the reason code for the situation in which an IFI call requests data from members of a data sharing group and not all the data is returned from group members.

  This is an output parameter of the type INTEGER.

- ***group-excess-bytes:*** Indicates the total length of data that was returned from other data sharing group members and did not fit in the return area.

  This is an output parameter of the type INTEGER.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The stored procedure did not encounter an SQL error during processing. Check the *IFI-return-code* output parameter to determine whether the DB2 command issued using the instrumentation facility interface (IFI) was successful or not.

  12 - The stored procedure encountered an SQL error during processing. The message output parameter contains messages describing the SQL error.

  This is an output parameter of the type INTEGER.

- ***message:*** Contains messages describing the SQL error encountered by the stored procedure. If no SQL error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of the type VARCHAR(1331).

ADMIN_COMMAND_DB2 output

This stored procedure returns the following output parameters:

- commands-executed
- IFI-return-code
- IFI-reason-code
- excess-bytes
- group-IFI-reason-code
- group-excess-bytes
- return-code
- message

In addition to the preceding output, the stored procedure returns two result sets.

The first result set is returned in the created global temporary table SYSIBM.DB2_CMD_OUTPUT and contains the DB2 command output messages that were not parsed.

Table 24-16 shows the format of the first result set.

Table 24-16 Result set row for SYSIBM.DB2_CMD_OUTPUT

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
TEXT	CHAR(80)	DB2 command output message line

The format of the second result set varies, depending on the DB2 command issued and the *parse-type* value.

Table 24-17 shows the format of the second result set returned in the created global temporary table SYSIBM.BUFFERPOOL_STATUS when ***parse-type*** = 'BP'.

Table 24-17 Result set row for SYSIBM.BUFFERPOOL_STATUS

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1to n*
BPNAME	CHAR(6)	Buffer pool name

Column name	Data type	Contents
VPSIZE	INTEGER	Buffer pool size
VPSEQT	INTEGER	Sequential steal threshold for the buffer pool
VPPSEQT	INTEGER	Parallel sequential threshold for the buffer pool
VPXPSEQT	INTEGER	Assisting parallel sequential threshold for the buffer pool
DWQT	INTEGER	Deferred write threshold for the buffer pool
PCT_VDWQT	INTEGER	Vertical deferred write threshold for the buffer pool (as percentage of virtual buffer pool size)
ABS_VDWQT	INTEGER	Vertical deferred write threshold for the buffer pool (as absolute number of buffers)
PGSTEAL	CHAR(4)	Page-stealing algorithm that DB2 uses for the buffer pool
ID	INTEGER	Buffer pool internal identifier
USE_COUNT	INTEGER	Number of open table spaces or index spaces that reference this buffer pool
PGFIX	CHAR(3)	Specifies whether the buffer pool should be fixed in real storage when it is used

Table 24-18 shows the format of the second result set returned in the created global temporary table SYSIBM.DB2_THREAD_STATUS when ***parse-type*** = 'THD'.

Table 24-18 Result set row for SYSIBM.DB2_THREAD_STATUS

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1to n*
TYPE	INTEGER	Thread type: **0** - Unknown **1** - Active **2** - Inactive **3** - Indoubt **4** - Postponed
NAME	CHAR(8)	Connection name used to establish the thread
STATUS	CHAR(11)	Status of the conversation or socket
ACTIVE	CHAR(1)	Indicates whether a thread is active or not. An asterisk means that the thread is active within DB2.
REQ	CHAR(5)	Current number of DB2 requests on the thread
ID	CHAR(12)	Recovery correlation ID associated with the thread
AUTHID	CHAR(8)	Authorization ID associated with the thread
PLAN	CHAR(8)	Plan name associated with the thread
ASID	CHAR(4)	Address space identifier
TOKEN	CHAR(6)	Unique thread identifier
COORDINATOR	CHAR(46)	Name of the two-phase commit coordinator

Column name	Data type	Contents
RESET	CHAR(5)	Indicates whether or not the thread needs to be reset to purge info from the indoubt thread report
URID	CHAR(12)	Unit of recovery identifier
LUWID	CHAR(35)	Logical unit of work ID of the thread
WORKSTATION	CHAR(18)	Client workstation name
USERID	CHAR(16)	Client user ID
APPLICATION	CHAR(32)	Client application name
ACCOUNTING	CHAR(247)	Client accounting information
LOCATION	VARCHAR(4050)	Location name of the remote system
DETAIL	VARCHAR(4050)	Additional thread information

Table 24-19 shows the format of the second result set returned in the created global temporary table SYSIBM.UTILITY_JOB_STATUS when *parse-type* = 'UT'

Table 24-19 Result set row for SYSIBM.UTILITY_JOB_STATUS

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
CSECT	CHAR(8)	Name of the command program CSECT that issued the message
USER	CHAR(8)	User ID of the person running the utility
MEMBER	CHAR(8)	Utility job is running on this member
UTILID	CHAR(16)	Utility job identifier
STATEMENT	INTEGER	Utility statement number
UTILITY	CHAR(20)	Utility name
PHASE	CHAR(20)	Utility restart from the beginning of this phase
COUNT	INTEGER	Number of pages or records processed in a utility phase
STATUS	CHAR(18)	Utility status
DETAIL	VARCHAR(4050)	Additional utility information
NUM_OBJ	INTEGER	Total number of objects in the list of objects the utility is processing
LAST_OBJ	INTEGER	Last object that started

Table 24-20 shows the format of the second result set returned in the created global temporary table SYSIBM.DB_STATUS when *parse-type* = 'DB' or 'TS' or 'IX'.

Table 24-20 Result set row for SYSIBM.DB_STATUS

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1to n*
DBNAME	CHAR(8)	Name of the database
SPACENAME	CHAR(8)	Name of the table space or index
TYPE	CHAR(2)	Status type: **DB** - Database **TS** - Tablespace **IX** - Index
PART	SMALLINT	Individual partition or range of partition
STATUS	CHAR(18)	Status of the database, table space or index

Table 24-21 shows the format of the second result set returned in the created global temporary table SYSIBM.DATA_SHARING_GROUP when *parse-type* = 'GRP'

Table 24-21 Result set row for SYSIBM.DATA_SHARING_GROUP

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1to n*
DB2_MEMBER	CHAR(8)	Name of the DB2 group member
ID	INTEGER	ID of the DB2 group member
SUBSYS	CHAR(4)	Subsystem name of the DB2 group member
CMDPREF	CHAR(8)	Command prefix for the DB2 group member
STATUS	CHAR(8)	Status of the DB2 group member
DB2_LVL	CHAR(3)	DB2 version, release and modification level
SYSTEM_NAME	CHAR(8)	Name of the z/OS system where the member is running, or was last running in cases when the member status is QUIESCED or FAILED
IRLM_SUBSYS	CHAR(4)	Name of the IRLM subsystem to which the DB2 member is connected
IRLMPROC	CHAR(8)	Procedure name of the connected IRLM

Table 24-22 shows the format of the second result set returned in the created global temporary table SYSIBM.DDF_CONFIG when *parse_type* = 'DDF':

Table 24-22 Result set row for SYSIBM.DDF_CONFIG

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
STATUS	CHAR(6)	Operational status of DDF
LOCATION	CHAR(18)	Location name of DDF
LUNAME	CHAR(17)	Fully qualified LUNAME of DDF
GENERICLU	CHAR(17)	Fully qualified generic LUNAME of DDF
IPV4ADDR	CHAR(17)	IPV4 address of DDF

Column name	Data type	Contents
IPV6ADDR	CHAR(39)	IPV6 address of DDF
TCPPORT	INTEGER	SQL listener port used by DDF
RESPORT	INTEGER	Resync listener port used by DDF
SQL_DOMAIN	CHAR(45)	SQL domain used by DDF
RSYNC_DOMAIN	CHAR(45)	Resync domain used by DDF

24.2.2 Job management

The following procedures belong to this group:

- SYSPROC.ADMIN_JOB_SUBMIT
- SYSPROC.ADMIN_JOB_FETCH
- SYSPROC.ADMIN_JOB_QUERY
- SYSPROC.ADMIN_JOB_CANCEL

SYSPROC.ADMIN_JOB_SUBMIT

The SYSPROC.ADMIN_JOB_SUBMIT stored procedure submits a job to a JES2 or JES3 system.

Load module name: DSNADMJS

Package name: DSNADMJS

Figure 24-5 illustrates the ADMIN_JOB_SUBMIT CALL syntax

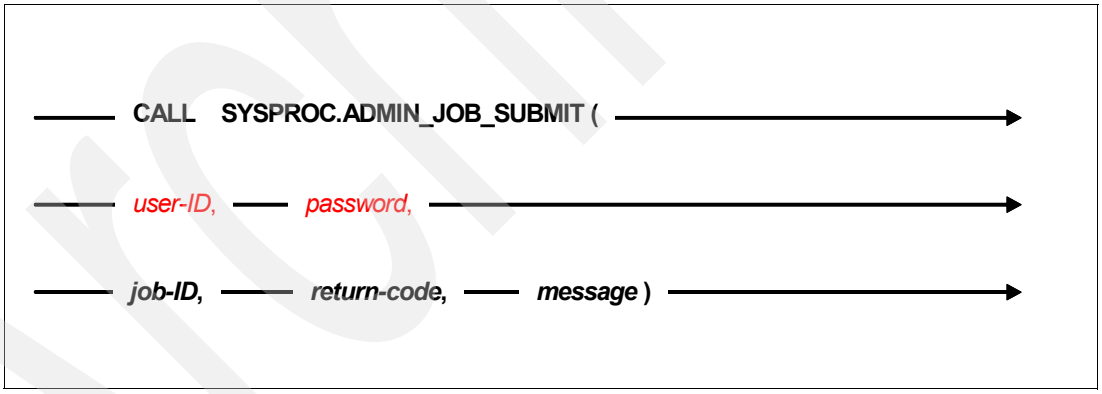

Figure 24-5 CALL ADMIN_JOB_SUBMIT stored procedure

ADMIN_JOB_SUBMIT option descriptions:

- ***user-ID:*** Specifies the user ID under which the job is submitted.

  This is an input parameter of type VARCHAR(128) and cannot be null.

- ***password:*** Specifies the password associated with the input parameter user-ID.

  The value of *password* is passed to the stored procedure as part of payload, and is not encrypted. It is not stored in dynamic cache when parameter markers are used.

  Recommendation: Have the application that invokes this stored procedure pass an encrypted password called a passticket.

  This is an input parameter of type VARCHAR(24) and cannot be null.

- ***job-ID:*** Identifies JES2 or JES3 job ID of the submitted job.

  This is an output parameter of type CHAR(8).

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:
  - 0 - The call completed successfully.
  - 12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

Additional ADMIN_JOB_SUBMIT input

In addition to the input parameters, the stored procedure submits the job's JCL from the created global temporary table SYSIBM.JOB_JCL for execution.

Table 24-23 shows the format of the created global temporary table SYSIBM.JOB_JCL.

Table 24-23 Row for input table SYSIBM.JOB_JCL

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, from 1to n
STMT	VARCHAR(80)	A JCL statement

ADMIN_JOB_SUBMIT output

This stored procedure returns the following output parameters:

- job-ID
- return-code
- message

SYSPROC.ADMIN_JOB_FETCH

The SYSPROC.ADMIN_JOB_FETCH stored procedure retrieves SYSOUT from JES spool and returns the SYSOUT.

Load module name: DSNADMJF

Package name: DSNADMJF

Figure 24-6 illustrates the ADMIN_JOB_FETCH CALL syntax.

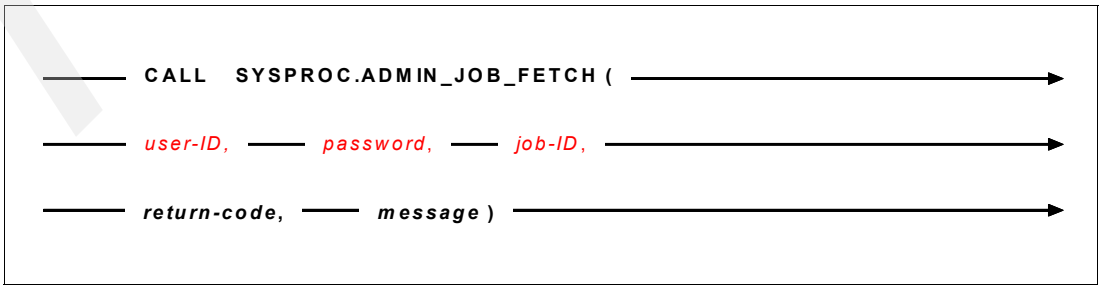

Figure 24-6 CALL ADMIN_JOB_FETCH stored procedure

ADMIN_JOB_FETCH option descriptions

- ***user-ID***: Specifies the user ID under which SYSOUT is retrieved.

  This is an input parameter of type VARCHAR(128) and cannot be null.

- ***password***: Specifies the password associated with the input parameter user-ID.

  The value of *password* is passed to the stored procedure as part of payload, and is not encrypted. It is not stored in dynamic cache when parameter markers are used.

  Recommendation: Have the application that invokes this stored procedure pass an encrypted password called a passticket.

  This is an input parameter of type VARCHAR(24) and cannot be null.

- job-ID: Specifies the JES2 or JES3 job ID whose SYSOUT data sets are to be retrieved.

  This is an input parameter of type CHAR(8) and cannot be null.

- ***return-code***: Provides the return code from the stored procedure. Possible values are:

  – 0 - The call completed successfully.
  – 12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message***: Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

ADMIN_JOB_FETCH output

This stored procedure returns the following output parameters:

- return-code
- message

In addition to the preceding output, the stored procedure returns one result set that contains the data from the JES-managed SYSOUT data set that belong to the job ID specified in the input parameter job-ID.

Table 24-24 shows the format of the result set returned in the created global temporary table SYSIBM.JES_SYSOUT.

Table 24-24 Result set row for SYSIBM.JES_SYSOUT

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
TEXT	VARCHAR(4096)	A record in the SYSOUT data set

SYSPROC.ADMIN_JOB_QUERY

The SYSPROC.ADMIN_JOB_QUERY stored procedure displays the status and completion information of a job.

Load module name: DSNADMJQ

Package name: None

Figure 24-7 illustrates the ADMIN_JOB_QUERY CALL syntax.

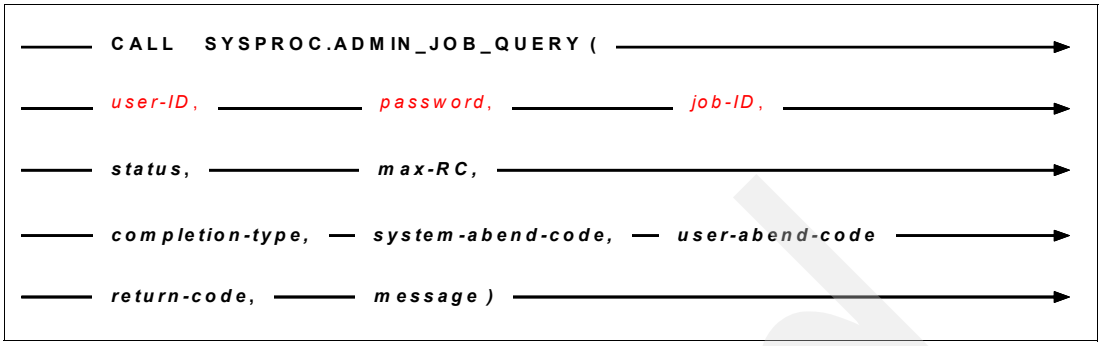

Figure 24-7 CALL ADMIN_JOB_QUERY stored procedure

ADMIN_JOB_QUERY option descriptions

- ***user-ID:*** Specifies the user ID under which the job is queried.

  This is an input parameter of type VARCHAR(128) and cannot be null.

- ***password:*** Specifies the password associated with the input parameter *user-ID*.

  The value of *password* is passed to the stored procedure as part of payload, and is not encrypted. It is not stored in dynamic cache when parameter markers are used.

  Recommendation: Have the application that invokes this stored procedure pass an encrypted password called a *passticket*.

  This is an input parameter of type VARCHAR(24) and cannot be null.

- ***job-ID:*** Specifies the job ID of the job being queried. Acceptable formats are:
  - Jnnnnnnn
  - JOBnnnnn

  where n is a digit between 0 and 9. For example: JOB01035

  Both Jnnnnnnn and JOBnnnnn must be exactly 8 characters in length.

  This is an input parameter of type CHAR(8), and cannot be null.

- ***status:*** Identifies the current status of the job. Possible values are:

  1 - Job received, but not yet run (INPUT)

  2 - Job running (ACTIVE)

  3 - Job finished and has output to be printed or retrieved (OUTPUT)

  4 - Job not found

  5 - Job in an unknown phase

  This is an output parameter of type INTEGER.

- ***max-RC:*** Provides the job completion code.

  This parameter is always null if querying in a JES3 z/OS Version 1.7 or earlier system. For JES3, this feature is only supported for z/OS Version 1.8 or higher.

  This is an output parameter of type INTEGER.

- ***completion-type:*** Identifies the job's completion type. Possible values are:

  0 - No completion information is available

  1 - Job ended normally

  2 - Job ended by completion code

  3 - Job had a JCL error

4 - Job was canceled

5 - Job terminated abnormally

6 - Converter terminated abnormally while processing the job

7 - Job failed security checks

8 - Job failed in end-of-memory

This parameter is always null if querying in a JES3 z/OS Version 1.7 or earlier system. For JES3, this feature is only supported for z/OS Version 1.8 or higher.

The ***completion-type*** information is the last six bits in the field STTRMXRC of the IAZSSST mapping macro. This information is returned via SSI 80. For additional information, see the discussion of the SSST macro in *z/OS MVS Data Areas*, SY28-1164.

This is an output parameter of type INTEGER.

- ***system-abend-code:*** Returns the system abend code if an abnormal termination occurs.

  This parameter is always null if querying in a JES3 z/OS Version 1.7 or earlier system. For JES3, this feature is only supported for z/OS Version 1.8 or higher.

  This is an output parameter of type INTEGER.

- ***user-abend-code:*** Returns the user abend code if an abnormal termination occurs.

  This parameter is always null if querying in a JES3 z/OS Version 1.7 or earlier system. For JES3, this feature is only supported for z/OS Version 1.8 or higher

  This is an output parameter of type INTEGER.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The call completed successfully

  4 - The job was not found, or the job status is unknown.

  12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  This is an output parameter of type VARCHAR(1331).

ADMIN_JOB_QUERY output

This stored procedure returns the following output parameters:

- status
- max-RC
- completion-type
- system-abend-code
- user-abend-code
- return-code
- message

SYSPROC.ADMIN_JOB_CANCEL

The SYSPROC.ADMIN_JOB_CANCEL stored procedure purges or cancels a job.

Load module name: DSNADMJP

Package name: None

Figure 24-8 illustrates the ADMIN_JOB_CANCEL CALL syntax.

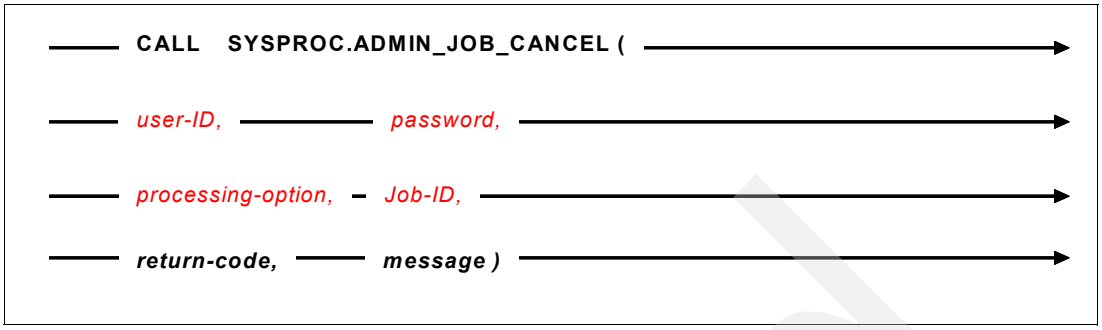

Figure 24-8 CALL ADMIN_JOB_CANCEL stored procedure

ADMIN_JOB_CANCEL option descriptions

- ***user-ID:*** Specifies the user ID under which the job is canceled or purged.

  This is an input parameter of type VARCHAR(128) and cannot be null.

- ***password:*** Specifies the password associated with the input parameter *user-ID*.

  The value of *password* is passed to the stored procedure as part of payload, and is not encrypted. It is not stored in dynamic cache when parameter markers are used.

  Recommendation: Have the application that invokes this stored procedure pass an encrypted password called a *passticket*.

  This is an input parameter of type VARCHAR(24) and cannot be null.

- ***processing-option:*** Identifies the type of command to invoke. Possible values are:
  - 1 - Cancel a job
  - 2 - Purge a job

  This is an input parameter of type INTEGER and cannot be null.

- ***job-ID:*** Specifies the job ID of the job to be canceled or purged. Acceptable formats are
  - Jnnnnnnnn
  - JOBnnnnn

  where *n* is a digit between 0 and 9. For example: JOB01035

  Both Jnnnnnnn and JOBnnnnn must be exactly 8 characters in length.

  This is an input parameter of type CHAR(8) and cannot be null.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:
  - 0 - The call completed successfully.
  - 12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  This is an output parameter of type VARCHAR(1331).

  The first messages in this area are generated by the stored procedure. Messages that are generated by z/OS might follow the first messages.

ADMIN_JOB_CANCEL output

This stored procedure returns the following output parameters:

- return-code
- message

24.2.3 Data set management

The following procedures belong to this group:

- SYSPROC.ADMIN_DS_BROWSE
- SYSPROC.ADMIN_DS_WRITE
- SYSPROC.ADMIN_DS_LIST
- SYSPROC.ADMIN_DS_RENAME
- SYSPROC.ADMIN_DS_DELETE
- SYSPROC.ADMIN_DS_SEARCH

SYSPROC.ADMIN_DS_BROWSE

The SYSPROC.ADMIN_DS_BROWSE stored procedure returns either text or binary records from a physical sequential (PS) data set, generation data set, or partitioned data set (PDS) or partitioned data set extended (PDSE) member. This stored procedure supports only data sets with LRECL=80 and RECFM=FB.

Data set action is performed under the security context of the authorization ID of the user who invoked the stored procedure.

Load module name: DSNADMDB

Package name: DSNADMDB

Figure 24-9 illustrates the ADMIN_DS_BROWSE CALL syntax.

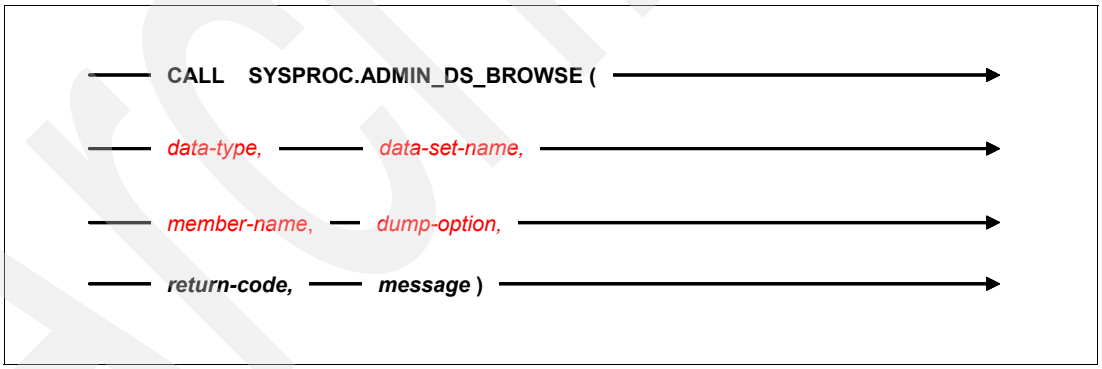

Figure 24-9 CALL ADMIN_DS_BROWSE stored procedure

ADMIN_DS_BROWSE option descriptions

- ***data-type:*** Specifies the type of data to be browsed. Possible values are:

  1 - Text data

  2 - Binary data

  This is an input parameter of type INTEGER and cannot be null.

- ***data-set-name:*** Specifies the name of the data set, or of the library that contains the member to be browsed. Possible values are:

  PS data set name: If reading from a PS data set, the *data-set-name* contains the name of the PS data set.

PDS or PDSE name: If reading from a member that belongs to this PDS or PDSE, the *data-set-name* contains the name of the PDS or PDSE.

GDS name: If reading from a generation data set, the *data-set-name* contains the name of the generation data set, such as USERGDG.FILE.G0001V00.

This is an input parameter of type CHAR(44) and cannot be null.

- ***member-name:*** Specifies the name of the PDS or PDSE member, if reading from a PDS or PDSE member. Otherwise, a blank character.

  This is an input parameter of type CHAR(8) and cannot be null.

- ***dump-option:*** Specifies whether to use the DB2 standard dump facility to dump the information necessary for problem diagnosis when an SQL error occurred or when a call to the IBM routine IEFDB476 to get messages about an unsuccessful SVC 99 call failed. Possible values are:

  Y - Generate a dump

  N - Do not generate a dump

  This is an input parameter of type CHAR(1) and cannot be NULL.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:
  - 0 - The call completed successfully.
  - 12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 or z/OS might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

ADMIN_DS_BROWSE output

This stored procedure returns the following output parameters:

- return-code
- message

In addition to the preceding output, the stored procedure returns one result set that contains the text or binary records read.

Table 24-25 shows the format of the result set returned in the created global temporary table SYSIBM.TEXT_REC_OUTPUT containing text records read.

Table 24-25 Result set row for SYSIBM.TEXT_REC_OUTPUT

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
TEXT_REC	VARCHAR(80)	Record read (text format)

Table 24-26 shows the format of the result set returned in the created global temporary table SYSIBM.BIN_REC_OUTPUT containing binary records read.

Table 24-26 Result set row for SYSIBM.BIN_REC_OUTPUT

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
BINARY_REC	VARCHAR(80) FOR BIT DATA	Record read (binary format)

SYSPROC.ADMIN_DS_WRITE

The SYSPROC.ADMIN_DS_WRITE stored procedure writes either text or binary records passed in a global temporary table to either a physical sequential (PS) data set, partitioned data set (PDS) or partitioned data set extended (PDSE) member, or generation data set (GDS). It can either append or replace an existing PS data set, PDS or PDSE member, or GDS. It can create a new PS data set, PDS or PDSE data set or member, or a new GDS for an existing generation data group (GDG) as needed. This stored procedure supports only data sets with LRECL=80 and RECFM=FB.

Data set action is performed under the security context of the authorization ID of the user who invoked the stored procedure.

Load module name: DSNADMDW

Package name: DSNADMDW

Figure 24-10 illustrates the ADMIN_DS_WRITE CALL syntax.

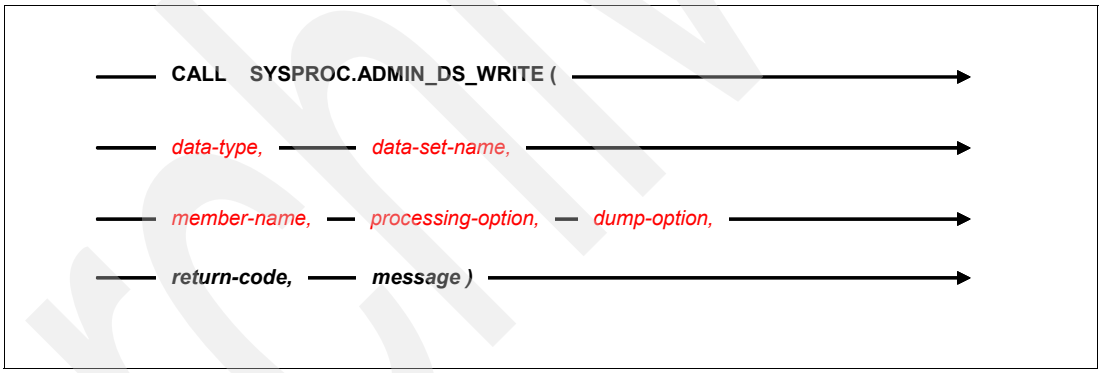

Figure 24-10 CALL ADMIN_DS_WRITE stored procedure

ADMIN_DS_WRITE option descriptions

- ***data-type:*** Specifies the type of data to be saved. Possible values are:

  1 - Text data

  2 - Binary data

  This is an input parameter of type INTEGER and cannot be null.

- ***data-set-name:*** Specifies the name of the data set, GDG that contains the GDS, or library that contains the member, to be written to. Possible values are:

  PS data set name: Name of the PS data set, if writing to a PS data set.

  GDG name: Name of the GDG, if writing to a GDS within this GDG.

  PDS or PDSE name: Name of the PDS or PDSE, if writing to a member that belongs to this library.

  This is an input parameter of type CHAR(44) and cannot be null.

- ***member-name***: Specifies the relative generation number of the GDS, if writing to a GDS, or the name of the PDS or PDSE member, if writing to a PDS or PDSE member. Otherwise, a blank character. Possible values are:

  GDS relative generation number: Relative generation number of a GDS, if writing to a GDS. For example: -1, 0, +1

  PDS or PDSE member name: Name of the PDS or PDSE member, if writing to a library member.

  blank: In all other cases, blank.

  This is an input parameter of type CHAR(8) and cannot be null.

- ***processing-option:*** Specifies the type of operation. Possible values are:

  R: Replace

  A: Append

  NM: New member

  ND: New PS, PDS, PDSE, or GDS data set

  This is an input parameter of type CHAR(2) and cannot be null.

- ***dump-option:*** Specifies whether to use the DB2 standard dump facility to dump the information necessary for problem diagnosis when an SQL error has occurred or when a call to the IBM routine IEFDB476 to get messages about an unsuccessful SVC 99 call failed. Possible values are:

  Y - Generate a dump

  N - Do not generate a dump

  This is an input parameter of type CHAR(1) and cannot be null.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The call completed successfully

  12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 or z/OS might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

Additional ADMIN_DS_WRITE input:

In addition to the input parameters, the stored procedure reads records to be written to a file from a created global temporary table. If the data to be written is text data, then the stored procedure reads records from SYSIBM.TEXT_REC_INPUT. If the data is binary data, then the stored procedure reads records from the created global temporary table SYSIBM.BIN_REC_INPUT.

Table 24-27 shows the format of the created global temporary table SYSIBM.TEXT_REC_INPUT containing text records to be saved.

Table 24-27 Input table SYSIBM.TEXT_REC_INPUT row format

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
TEXT_REC	CHAR(80)	Text record to be saved

Table 24-28 shows the format of the created global temporary table SYSIBM.BIN_REC_INPUT containing binary records to be saved:

Table 24-28 SYSIBM.BIN_REC_INPUT result set format

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, from 1 to n
BINARY_REC	VARCHAR(80) FOR BIT DATA	Binary record to be saved

ADMIN_DS_WRITE output

This stored procedure returns the following output parameters:

- return-code
- message

SYSPROC.ADMIN_DS_LIST

The SYSPROC.ADMIN_DS_LIST stored procedure returns a list of data set names, generation data group (GDG), partitioned data set (PDS), or partitioned data set extended (PDSE) members, or generation data sets of a GDG.

Data set action is performed under the security context of the authorization ID of the user who invoked the stored procedure.

Load module name: DSNADMDL

Package name: DSNADMDL

Figure 24-11 illustrates the ADMIN_DS_LIST CALL syntax.

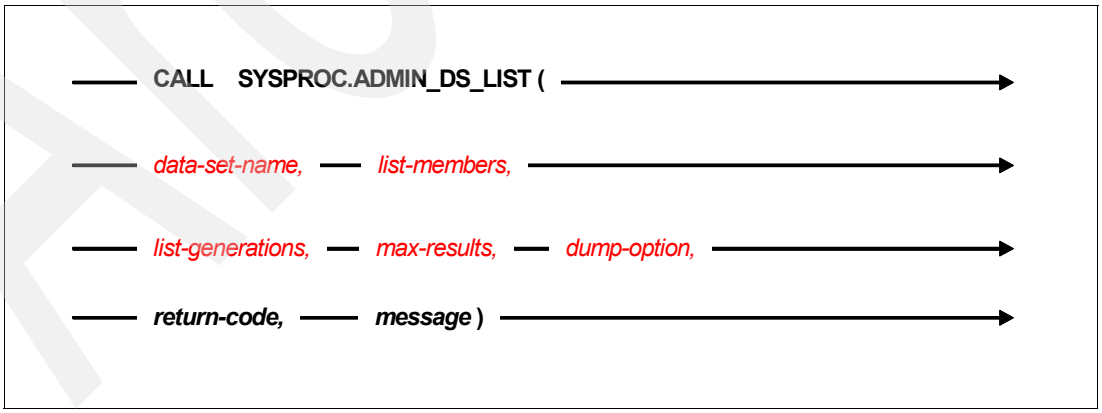

Figure 24-11 CALL ADMIN_DS_LIST stored procedure

ADMIN_DS_LIST option descriptions

- ***data-set-name:*** Specifies the data set name. You can use masking characters. For example: USER.*

  If no masking characters are used, only one data set will be listed.

This is an input parameter of type CHAR(44) and cannot be null.

- ***list-members:*** Specifies whether to list PDS or PDSE members. Possible values are:

  Y - List members. Only set to Y when *data-set-name* is a fully qualified PDS or PDSE.

  N - Do not list members.

  This is an input parameter of type CHAR(1) and cannot be null.

- ***list-generations:*** Specifies whether to list generation data sets. Possible values are:

  Y - List generation data sets. Only set to Y when data-set-name is a fully qualified GDG.

  N - Do not list generation data sets.

  This is an input parameter of type CHAR(1) and cannot be null.

- ***max-results:*** Specifies the maximum number of result set rows. This option is applicable only when both *list-members* and *list-generations* are 'N'.

  This is an input parameter of type INTEGER and cannot be null.

- ***dump-option:*** Specifies whether to use the DB2 standard dump facility to dump the information necessary for problem diagnosis when any of the following errors occurred:

  – SQL error

  – A call to the IBM routine IEFDB476 to get messages about an unsuccessful SVC 99 call failed

  – Load Catalog Search Interface module error

  Possible Values are:

  Y - Generate a dump.

  N - Do not generate a dump.

  This is an input parameter of type CHAR(1) and cannot be null.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The call completed successfully

  12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 or z/OS might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

ADMIN_DS_LIST output

This stored procedure returns the following output parameters:

- return-code
- message

In addition to the preceding output, the stored procedure returns one result set that contains the list of data sets, GDGs, PDS or PDSE members, or generation data sets that were requested. Table 24-29 shows the format of the result set returned in the created global temporary table SYSIBM.DSLIST.

Table 24-29 Result set row for SYSIBM.DSLIST

Column name	Data type	Contents
DSNAME	VARCHAR(44)	Data set name, if *list-members* is 'N' and *list-generations* is 'N'. Member name, if *list-members* is 'Y' Absolute generation number (of the form G0000V00) from a generation data set name, if *list-generations* is 'Y'
CREATE_YEAR	INTEGER	Year data set was created. Not applicable for member and VSAM cluster.
CREATE_DAY	INTEGER	The day of the year that the data set was created, as an integer in the range of 1 to 366 where 1 represents January 1). Not applicable for member and VSAM cluster.
TYPE	INTEGER	Type of data set Possible values are: - **0** - Unknown type of data set - **1** - PDS data set - **2** - PDSE data set - **3 -** Member of PDS or PDSE - **4 -** Physical sequential data set - **5 -** Generation data group - **6 -** Generation data set - **8 -** VSAM cluster - **9 -** VSAM data component -**10 -** VSAM index component
VOLUME	CHAR(6)	Volume where data set resides. Not applicable for member and VSAM cluster.
PRIMARY_EXTENT	INTEGER	Size of first extent (not applicable for member and VSAM cluster)
SECONDARY_EXTENT	INTEGER	Size of secondary extent. Not applicable for member and VSAM cluster.
MEASUREMENT_UNIT	CHAR(9)	Unit of measurement for first extent and secondary extent. Possible values are: - *BLOCKS* - *BYTES* - *CYLINDERS* - *KB* - *MB* - *TRACKS* Not applicable for member and VSAM cluster
EXTENTS_IN_USE	INTEGER	Current allocated extents. Not applicable for member and VSAM cluster.
DASD_USAGE	CHAR(8) FOR BIT DATA	Disk usage. For VSAM data and VSAM index only.

Column name	Data type	Contents
HARBA	CHAR(6) FOR BIT DATA	High allocated RBA. For VSAM data and VSAM index only.
HURBA	CHAR(6) FOR BIT DATA	High used RBA. For VSAM data and VSAM index only.

When a data set spans more than one volume, one row is returned for each volume that contains a piece of the data set. The VOLUME, EXTENTS_IN_USE, DASD_USAGE, HARBA, and HURBA columns will reflect information for the specified volume.

SYSPROC.ADMIN_DS_RENAME

The SYSPROC.ADMIN_DS_RENAME stored procedure renames a physical sequential (PS) data set, a partitioned data set (PDS) or partitioned data set extended (PDSE), or a member of a PDS or PDSE.

Data set action is performed under the security context of the authorization ID of the user who invoked the stored procedure.

Load module name: DSNADMDR

Package name: None

Figure 24-12 illustrates the ADMIN_DS_RENAME CALL syntax.

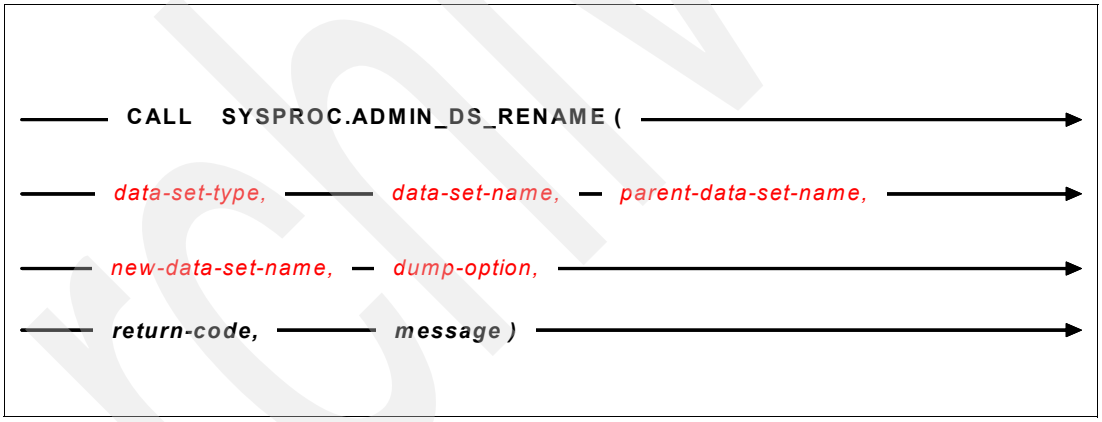

Figure 24-12 CALL ADMIN_DS_RENAME stored procedure

ADMIN_DS_RENAME option description

- ***data-set-type:*** Specifies the type of data set to rename. Possible values are:

  1 - Partitioned data set (PDS)

  2 - Partitioned data set extended (PDSE)

  3 - Member of a PDS or PDSE

  4 - Physical sequential data set (PS)

  This is an input parameter of the type INTEGER and cannot be null.

- ***data-set-name:*** Specifies the data set or member to be renamed. Possible values are:

  PS, PDS, or PDSE name: If *data-set-type* is 1, 2, or 4, the *data-set-name* contains the name of the PS, PDS, or PDSE to be renamed.

  PDS or PDSE member name: If *data-set-type* is 3, the *data-set-name* contains the name of the PDS or PDSE member to be renamed.

This is an input parameter of type CHAR(44) and cannot be null.

- ***parent-data-set-name:*** Specifies the name of the PDS or PDSE, if renaming a PDS or PDSE member. Otherwise, a blank character. Possible values are:

  blank: If *data-set-type* is 1, 2, or 4, the *parent-data-set-name* is left blank.

  PDS or PDSE name: If *data-set-type* is 3, the *parent-data-set-name* contains the name of the PDS or PDSE whose member is to be renamed.

  This is an input parameter of type CHAR(44) and cannot be null.

- ***new-data-set-name:*** Specifies the new data set or member name. Possible values are:

  new data set name: If *data-set-type* is 1, 2, or 4, the *new-data-set-name* contains the new data set name.

  new member name: If *data-set-type* is 3, the *new-data-set-name* contains the new member name.

  This is an input parameter of type CHAR(44) and cannot be null.

- ***dump-option:*** Specifies whether to use the DB2 standard dump facility to dump the information necessary for problem diagnosis when any of the following errors occurred:
  - A call to the IBM routine IEFDB476 to get messages about an unsuccessful SVC 99 call failed
  - Load IDCAMS program error

  Possible values are:

  Y - Generate a dump

  N - Do not generate a dump

  This is an input parameter of type CHAR(1) and cannot be null.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The data set, PDS member, or PDSE member was renamed successfully

  12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages based on *return-code* and *data-set-type* combinations.

  Table 24-30 shows the messages that will be returned in a message based on *return-code* and *data-set-type* combinations.

Table 24-30 ADMIN_DS_RENAME expected messages

return-code	***data-set-type***	***message***
0	1, 2. or 4	Contains IDCAMS messages.
0	3	No message is returned.
Not 0	not applicable	Contains messages describing the error encountered by the stored procedure. The first messages are generated by the stored procedure and messages that are generated by z/OS might follow these first messages. The first messages can also be generated by z/OS.

This is an output parameter of type VARCHAR(1331).

ADMIN_DS_RENAME output

This stored procedure returns the following output parameters:

- return-code
- message

SYSPROC.ADMIN_DS_DELETE

The SYSPROC.ADMIN_DS_DELETE stored procedure deletes a physical sequential (PS) data set, a partitioned data set (PDS) or partitioned data set extended (PDSE), a generation data set (GDS), or a member of a PDS or PDSE.

Data set action is performed under the security context of the authorization ID of the user who invoked the stored procedure.

Load module name: DSNADMDD

Package name: None

Figure 24-13 illustrates the ADMIN_DS_DELETE CALL syntax.

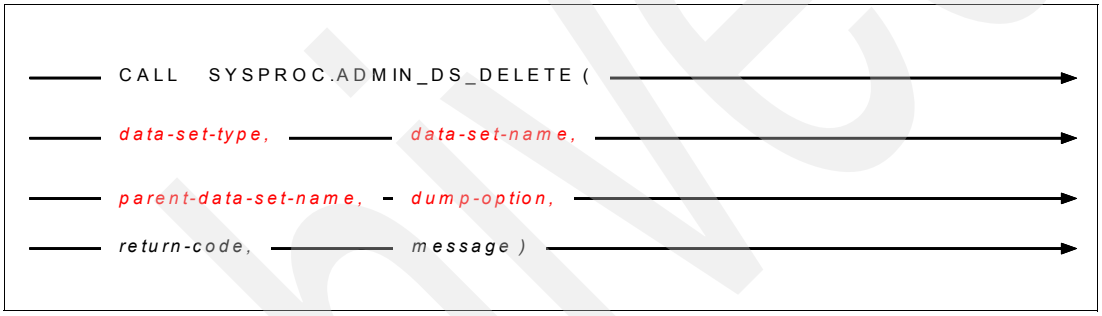

Figure 24-13 CALL ADMIN_DS_DELETE stored procedure

ADMIN_DS_DELETE option descriptions

- ***data-set-type:*** Specifies the type of data set to delete. Possible values are:

  1 - Partitioned data set (PDS)

  2 - Partitioned data set extended (PDSE)

  3 - Member of a PDS or PDSE

  4 - Physical sequential data set (PS)

  6 - Generation data set (GDS)

  This is an input parameter of type INTEGER and cannot be null.

- ***data-set-name:*** Specifies the name of the data set, library member, or GDS absolute generation number to be deleted. Possible values are:

  PS, PDS, or PDSE name: If *data-set-type* is 1, 2, or 4, the *data-set-name* contains the name of the PS, PDS, or PDSE to be deleted.

  PDS or PDSE member name: If *data-set-type* is 3, the *data-set-name* contains the name of the PDS or PDSE member to be deleted.

  absolute generation number: If *data-set-type* is 6, the *data-set-name* contains the absolute generation number of the GDS to be deleted, such as G0001V00.

  This is an input parameter of type CHAR(44) and cannot be null.

- ***parent-data-set-name*:** Specifies the name of the library that contains the member to be deleted, or of the GDG that contains the GDS to be delete. Otherwise blank. Possible values are:

  blank: If *data-set-type* is 1, 2, or 4, the *parent-data-set-name* is left blank.

  PDS or PDSE name: If *data-set-type* is 3, the *parent-data-set-name* contains the name of the PDS or PDSE whose member is to be deleted.

  GDG name: If *data-set-type* is 6, the *parent-data-set-name* contains the name of the GDG that the GDS to be deleted belongs to.

  This is an input parameter of type CHAR(44) and cannot be null.

- ***dump-option:*** Specifies whether to use the DB2 standard dump facility to dump the information necessary for problem diagnosis when a call to the IBM routine IEFDB476 to get messages about an unsuccessful SVC 99 call failed

  Possible values are:

  Y - Generate a dump

  N - Do not generate a dump

  This is an input parameter of type CHAR(1) and cannot be null.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - Data set, PDS member, PDSE member, or GDS was deleted successfully.

  12 - The call did not complete successfully. The message output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by z/OS might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

ADMIN_DS_DELETE output

This stored procedure returns the following output parameters:

- return-code
- message

SYSPROC.ADMIN_DS_SEARCH

The SYSPROC.ADMIN_DS_SEARCH stored procedure determines if a physical sequential (PS) data set, partitioned data set (PDS), partitioned data set extended (PDSE), generation data group (GDG), generation data set (GDS) is cataloged, or if a library member of a cataloged PDS or PDSE exists.

Data set action is performed under the security context of the authorization ID of the user who invoked the stored procedure.

Load module name: DSNADMDE

Package name: None

Figure 24-14 illustrates the ADMIN_DS_SEARCH CALL syntax.

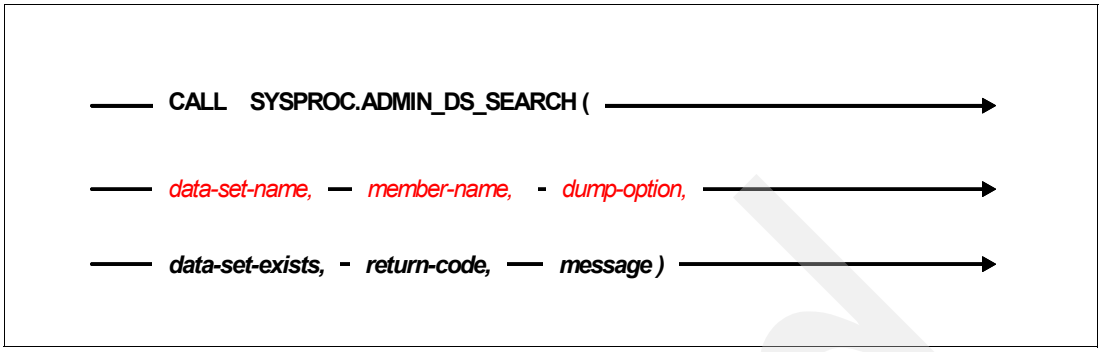

Figure 24-14 CALL ADMIN_DS_SEARCH stored procedure

ADMIN_DS_SEARCH option descriptions

- ***data-set-name:*** Specifies the name of a PS data set, PDS, PDSE, GDG, or GDS

  This is an input parameter of type CHAR(44) and cannot be null.

- ***member-name:*** Specifies the name of a PDS or PDSE member. Set this parameter to a blank character if you only want to check the existence of the PDS or PDSE.

  This is an input parameter for type CHAR(8) and cannot be null.

- ***dump-option:*** Specifies whether to use the DB2 standard dump facility to dump the information necessary for problem diagnosis when any of the following errors occurred:
  - A call to the IBM routine IEFDB476 to get messages about an unsuccessful SVC 99 call failed
  - Load IDCAMS program error

  Possible values are:

  Y - Generate a dump

  N - Do not generate a dump

  This is an input parameter of type CHAR(1) and cannot be null.

- ***data-set-exists:*** Indicates whether a data set or library member exists or not. Possible values are:

  -1 - Call did not complete successfully. Unable to determine if a data set or member exists.

  0 - Data set or member was found

  1 - Data set not found

  2 - PDS or PDSE member not found

  This is an output parameter of type INTEGER

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The call completed successfully

  12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains IDCAMS messages if *return-code* is 0. Otherwise, contains messages describing the error encountered by the stored procedure. The first messages are generated by the stored procedure and messages that are generated by z/OS might follow these first messages.

This is an output parameter of type VARCHAR(1331).

ADMIN_DS_SEARCH output

This stored procedure returns the following output parameters:

- data-set-exists
- return-code
- message

General GDG and GDS considerations

The following shows a GDG and a GDS:

```
USER01.GDG
USER01.GDG.G0001V00
```

ADMIN_DS_LIST will only list the GDG. The GDSs will only be listed by ADMIN_DS_LIST when the *data-set-name* is USER01.GDG and *list-generations* ='Y'. The DSNAME returned then is G0001V00.

GDGs cannot be renamed.

If you want to check with ADMIN_DS_SEARCH if a generation exists, check with a *data-set-name* of USER01.GDG.G0001V00.

If you want to delete a GDS with ADMIN_DS_DELETE, specify G0001V00 as the *data-set-name* and USER01.GDG as the *parent-data-set-name*.

24.2.4 System administration

The following procedures belong to this group:

- SYSPROC.ADMIN_INFO_HOST
- SYSPROC.ADMIN_INFO_SSID
- SYSPROC.DSNACICS
- SYSPROC.DSNLEUSR
- SYSPROC.DSNAIMS
- SYSPROC.DSNAIMS2
- SYSPROC.ADMIN_UTL_SCHEDULE
- SYSPROC.ADMIN_UTL_SORT

SYSPROC.ADMIN_INFO_HOST

The SYSPROC.ADMIN_INFO_HOST stored procedure returns the host name of a connected DB2 subsystem or the host name of every member of a data sharing group.

Load module name: DSNADMIH

Package name: DSNADMIH

Figure 24-15 illustrates the ADMIN_INFO_HOST CALL syntax.

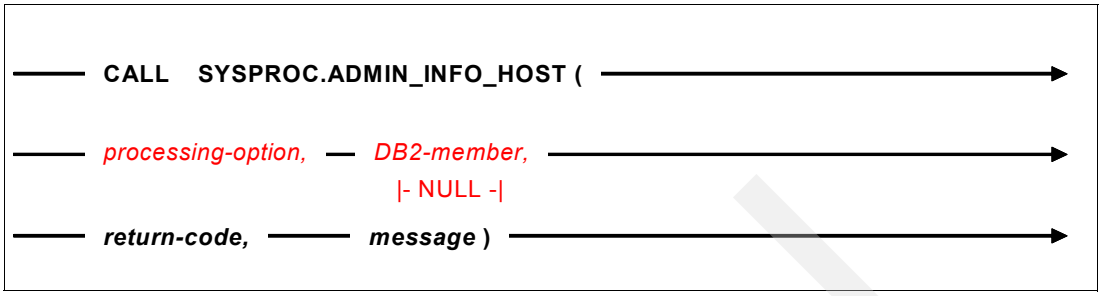

Figure 24-15 CALL ADMIN_INFO_HOST stored procedure

ADMIN_INFO_HOST option descriptions

- ***processing-option:*** Specifies the processing option. Possible values are:

  1 - Return the host name of the connected DB2 subsystem or the host name of a specified DB2 data sharing group member.

  For a data sharing group member, you must specify *DB2-member*.

  2 - Return the host name of every DB2 member of the same data sharing group.

  This is an input parameter of type INTEGER, and cannot be not null.

- ***DB2-member:*** Specifies the DB2 data sharing group member name.

  This parameter must be null if *processing-option* is 2.

  This is an input parameter of type CHAR(8).

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The call completed successfully.

  4 - Unable to list the host name of the connected DB2 subsystem or of every DB2 member of the same data sharing group due to one of the following reasons:

  - The IPADDR field returned when the -DISPLAY DDF command is executed on the connected DB2 subsystem or DB2 member contains the value -NONE.

  - One of the DB2 members is down.

  12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

ADMIN_INFO_HOST output

This stored procedure returns the following output parameters:

- return-code
- message

In addition to the preceding output, the stored procedure returns one result set that contains the host names.

Table 24-31 shows the format of the result set returned in the created global temporary table SYSIBM.SYSTEM_HOSTNAME.

Table 24-31 Result set row SYSIBM.SYSTEM_HOSTNAME

Column name	Data type	Contents
ROWNUM	INTEGER	Sequence number of the table row, *from 1 to n*
DB2_MEMBER	CHAR(8)	DB2 data sharing group member name
HOSTNAME	VARCHAR(255)	Host name of the connected DB2 subsystem if the *processing-option* input parameter is 1 and the *DB2-member* input parameter is null. Otherwise, the host name of the DB2 member specified in the DB2_MEMBER column.

SYSPROC.ADMIN_INFO_SSID

The SYSPROC.ADMIN_INFO_SSID stored procedure returns the name of the connected DB2 subsystem.

Load module name: DSNADMIS

Package name: None

Figure 24-16 illustrates the ADMIN_INFO_SSID CALL syntax.

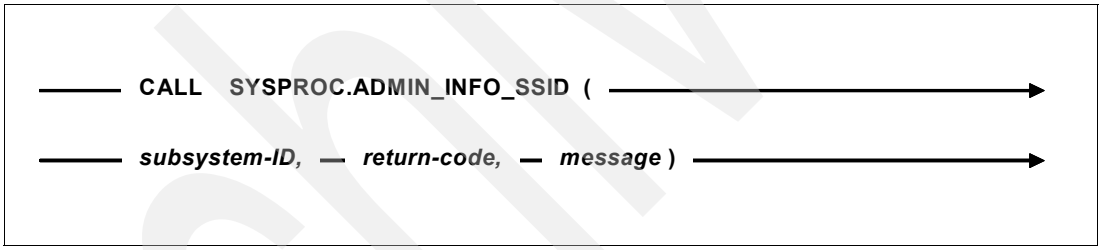

Figure 24-16 CALL ADMIN_INFO_SSID stored procedure

ADMIN_INFO_SSID option descriptions

- ***subsystem-ID***: Identifies the subsystem ID of the connected DB2 subsystem.

  This is an output parameter of type VARCHAR(4).

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The call completed successfully.

  12 - The call did not complete successfully. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  This is an output parameter of the type VARCHAR(1331).

ADMIN_INFO_SSID output

This stored procedure returns the following output parameters:

- subsystem-ID
- return-code
- message

SYSPROC.DSNACICS

The CICS transaction invocation stored procedure (DSNACICS) invokes CICS server programs.

Load module name: DSNACICS

Package name: None

DSNACICS gives workstation applications a way to invoke CICS server programs while using TCP/IP as their communication protocol. The workstation applications use TCP/IP and DB2 Connect to connect to a DB2 for z/OS subsystem, and then call DSNACICS to invoke the CICS server programs.

The DSNACICS input parameters require knowledge of various CICS resource definitions with which the workstation programmer might not be familiar. For this reason, DSNACICS invokes the DSNACICX user exit routine. The system programmer can write a version of DSNACICX that checks and overrides the parameters that the DSNACICS caller passes. If no user version of DSNACICX is provided, DSNACICS invokes the default version of DSNACICX, which does not modify any parameters.

Environmental considerations

DSNACICS runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

If you use CICS Transaction Server for OS/390 Version 1 Release 3 or later, you can register your CICS system as a resource manager with recoverable resource management services (RRMS). When you do that, changes to DB2 databases that are made by the program that calls DSNACICS and the CICS server program that DSNACICS invokes are in the same two-phase commit scope. This means that when the calling program performs an SQL COMMIT or ROLLBACK, DB2 and RRS inform CICS about the COMMIT or ROLLBACK.

If the CICS server program that DSNACICS invokes accesses DB2 resources, the server program runs under a separate unit of work from the original unit of work that calls the stored procedure. This means that the CICS server program might deadlock with locks that the client program acquires.

Syntax

The following syntax diagram shows the SQL CALL statement for invoking this stored procedure.

Because the linkage convention for DSNACICS is GENERAL WITH NULLS, if you pass parameters in host variables, you need to include a null indicator with every host variable. Null indicators for input host variables must be initialized before you execute the CALL statement.

The syntax diagram in Figure 24-17 shows the SQL CALL statement for DSNACICS.

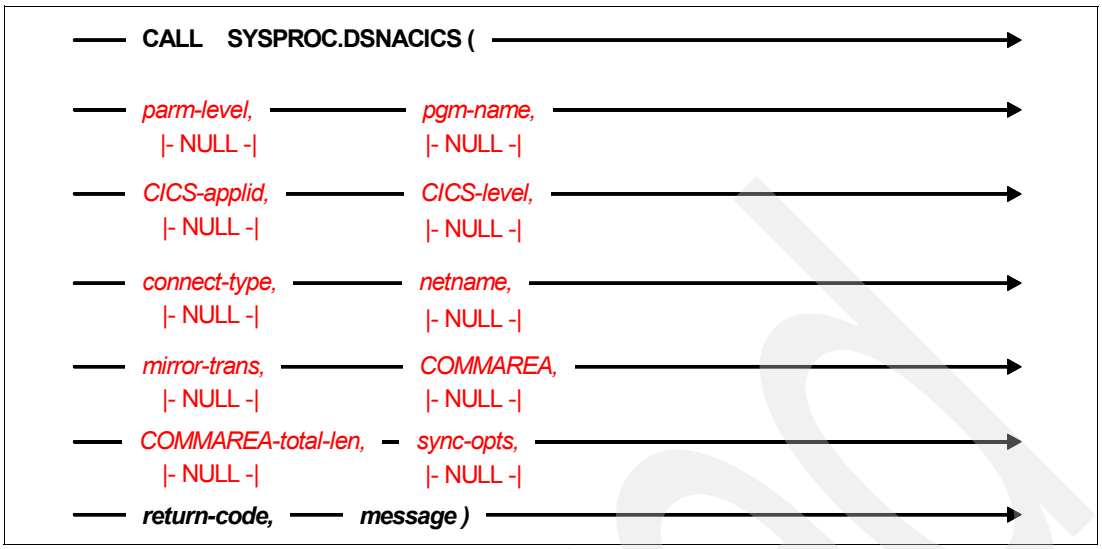

Figure 24-17 CALL DSNACICS stored procedure

DSNACICS option description

- ***parm-level:*** Specifies the level of the parameter list that is supplied to the stored procedure.

  This is an input parameter of type INTEGER. The value must be 1.

- ***pgm-level:*** Specifies the name of the CICS program that DSNACICS invokes. This is the name of the program that the CICS mirror transaction calls, not the CICS transaction name.

  This is an input parameter of type CHAR(8).

- ***CICS-applid:*** Specifies the applid of the CICS system to which DSNACICS connects.

  This is an input parameter of type CHAR(8).

- ***CICS-level:*** Specifies the level of the target CICS subsystem:

  1 - The CICS subsystem is CICS for MVS/ESA Version 4 Release 1, CICS Transaction Server for OS/390 Version 1 Release 1, or CICS Transaction Server for OS/390 Version 1 Release 2.

  2 - The CICS subsystem is CICS Transaction Server for OS/390 Version 1 Release 3 or later.

  This is an input parameter of type INTEGER.

- ***connect-type:*** Specifies whether the CICS connection is generic or specific. Possible values are GENERIC or SPECIFIC.

  This is an input parameter of type CHAR(8).

- ***netname:*** If the value of connection-type is SPECIFIC, specifies the name of the specific connection that is to be used. This value is ignored if the value of *connection-type* is GENERIC.

  This is an input parameter of type CHAR(8)

- ***mirror-trans:*** Specifies the name of the CICS mirror transaction to invoke. This mirror transaction calls the CICS server program that is specified in the pgm-name parameter. mirror-trans must be defined to the CICS server region, and the CICS resource definition for mirror-trans must specify DFHMIRS as the program that is associated with the transaction.

If this parameter contains blanks, DSNACICS passes a *mirror-trans* parameter value of null to the CICS EXCI interface. This allows an installation to override the transaction name in various CICS user-replaceable modules. If a CICS user exit routine does not specify a value for the mirror transaction name, CICS invokes CICS-supplied default mirror transaction CSMI.

This is an input parameter of type CHAR(4).

- **COMMAREA:** Specifies the communication area (COMMAREA) that is used to pass data between the DSNACICS caller and the CICS server program that DSNACICS calls.

  This is an input/output parameter of type VARCHAR(32704). In the length field of this parameter, specify the number of bytes that DSNACICS sends to the CICS server program.

- **COMMAREA-total-len:** Specifies the total length of the COMMAREA that the server program needs.

  This is an input parameter of type INTEGER. The length must be greater than or equal to the value that you specify in the length field of the COMMAREA parameter and less than or equal to 32704. When the CICS server program completes, DSNACICS passes the server program's entire COMMAREA, which is COMMAREA-total-len bytes in length, to the stored procedure caller.

- **sync-opts:** Specifies whether the calling program controls resource recovery, using two-phase commit protocols that are supported by RRS. Possible values are:

  1 - The client program controls commit processing. The CICS server region does not perform a syncpoint when the server program returns control to CICS. Also, the server program cannot take any explicit syncpoints. Doing so causes the server program to abnormally terminate.

  2 - The target CICS server region takes a syncpoint on successful completion of the server program. If this value is specified, the server program can take explicit syncpoints.

  When CICS has been set up to be an RRS resource manager, the client application can control commit processing using SQL COMMIT requests. DB2 for z/OS ensures that CICS is notified to commit any resources that the CICS server program modifies during two-phase commit processing.

  When CICS has not been set up to be an RRS resource manager, CICS forces syncpoint processing of all CICS resources at completion of the CICS server program. This commit processing is not coordinated with the commit processing of the client program.

  This option is ignored when CICS-level is 1. This is an input parameter of type INTEGER.

- ***return-code:*** Return code from the stored procedure. Possible values are:

  0 - The call completed successfully.

  12 - The request to run the CICS server program failed. The *msg-area* parameter contains messages that describe the error.

  This is an output parameter of type INTEGER.

- ***msg-area:*** Contains messages if an error occurs during stored procedure execution. The first messages in this area are generated by the stored procedure. Messages that are generated by CICS or the DSNACICX user exit routine might follow the first messages. The messages appear as a series of concatenated, viewable text strings.

  This is an output parameter of type VARCHAR(500).

USER EXIT ROUTINE

DSNACICS always calls user exit routine DSNACICX. You can use DSNACICX to change the values of DSNACICS input parameters before you pass those parameters to CICS. If you do not supply your own version of DSNACICX, DSNACICS calls the default DSNACICX, which modifies no values and does an immediate return to DSNACICS. The source code for the default version of DSNACICX is in member DSNASCIX in data set prefix.SDSNSAMP. The source code for a sample version of DSNACICX that is written in COBOL is in member DSNASCIO in data set prefix.SDSNSAMP.

OUTPUT

DSNACICS places the return code from DSNACICS execution in the *return-code* parameter. If the value of the return code is non-zero, DSNACICS puts its own error messages and any error messages that are generated by CICS and the DSNACICX user exit routine in the *msg-area* parameter.

The *COMMAREA* parameter contains the COMMAREA for the CICS server program that DSNACICS calls. The *COMMAREA* parameter has a VARCHAR type. Therefore, if the server program puts data other than character data in the COMMAREA, that data can become corrupted by code page translation as it is passed to the caller. To avoid code page translation, you can change the *COMMAREA* parameter in the CREATE PROCEDURE statement for DSNACICS to VARCHAR(32704) FOR BIT DATA. However, if you do so, the client program might need to do code page translation on any character data in the COMMAREA to make it readable.

Restrictions

Because DSNACICS uses the distributed program link (DPL) function to invoke CICS server programs, server programs that you invoke through DSNACICS can contain only the CICS API commands that the DPL function supports. The list of supported commands is documented in *CICS 3.1 Application Programming Reference,* SC33-1688.

Debugging

If you receive errors when you call DSNACICS, ask your system administrator to add a DSNDUMP DD statement in the startup procedure for the address space in which DSNACICS runs. The DSNDUMP DD statement causes DB2 to generate an SVC dump whenever DSNACICS issues an error message.

SYSPROC.DSNLEUSR

The DSNLEUSR stored procedure is a sample stored procedure that lets you store encrypted values in the translated authorization ID (NEWAUTHID) and password fields of the SYSIBM.USERNAMES table.

You provide all the values for a SYSIBM.USERNAMES row as input to DSNLEUSR. DSNLEUSR encrypts the translated authorization ID and password values before it inserts the row into SYSIBM.USERNAMES.

Load module name: DSNLEUSR

Package name: DSNLEUSR

Environment

DSNLEUSR has the following requirement:

z/OS Integrated Cryptographic Service Facility (ICSF) must be installed, configured, and active. See *Integrated Cryptographic Service Facility System Programmer's Guide,* SC23-3974 for more information.

Syntax

The syntax diagram in Figure 24-18 shows the SQL CALL statement for DSNLEUSR.

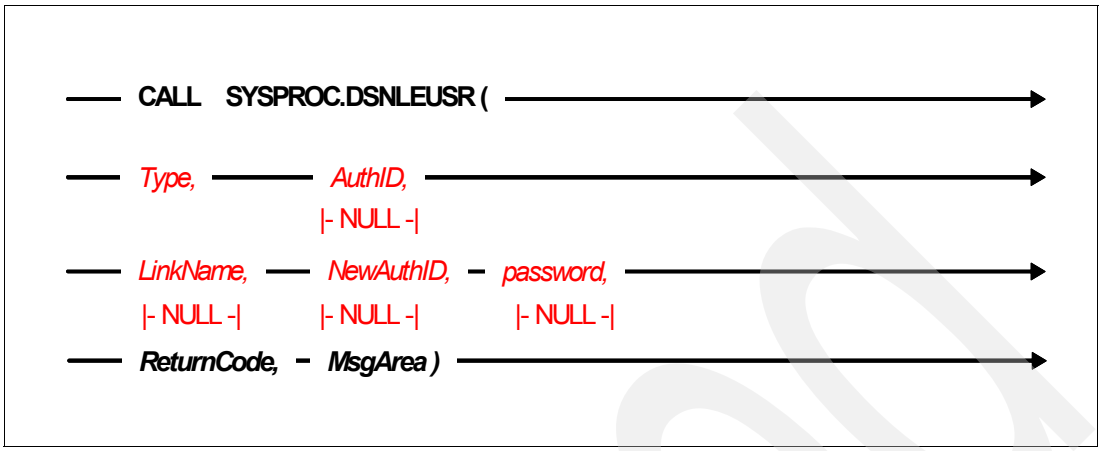

Figure 24-18 CALL DSNLEUSR stored procedure

DSNLEUSR option description

- ***Type:*** Specifies the value that is to be inserted into the TYPE column of SYSIBM.USERNAMES.

  This is an input parameter of type CHAR(1).

- ***AuthID:*** Specifies the value that is to be inserted into the AUTHID column of SYSIBM.USERNAMES.

  This is an input parameter of type VARCHAR(128). If you specify a null value, DSNLEUSR does not insert a value for AUTHID.

- ***LinkName:*** Specifies the value that is to be inserted into the LINKNAME column of SYSIBM.USERNAMES.

  This is an input parameter of type CHAR(8). If you specify a null value, DSNLEUSR does not insert a value for LINKNAME.

- ***NewAuthID:*** Specifies the value that is to be inserted into the NEWAUTHID column of SYSIBM.USERNAMES.

  This is an input parameter of type VARCHAR(119). Although the NEWAUTHID field of SYSIBM.USERNAMES is VARCHAR(128), your input value is restricted to 119 or fewer bytes.

  If you specify a null value, DSNLEUSR does not insert a value for NEWAUTHID.

- ***password:*** Specifies the value that is to be inserted into the PASSWORD column of SYSIBM.USERNAMES.

  This is an input parameter of type CHAR(8). Although the PASSWORD field of SYSIBM.USERNAMES is VARCHAR(24), your input value is restricted to 8 or fewer bytes.

  If you specify a null value, DSNLEUSR does not insert a value for PASSWORD.

- ***ReturnCode:***

  The return code from DSNLEUSR execution. Possible values are:

  0 - DSNLEUSR executed successfully.

  8 - The request to encrypt the translated authorization ID or password failed. *MsgArea* contains the following fields:

An unformatted SQLCA that describes the error.

A string that contains a DSNL045I message with the ICSF return code, the ICSF reason code, and the ICSF function that failed. The string immediately follows the SQLCA field and does not begin with a length field.

12 - The insert operation for the SYSIBM.USERNAMES row failed. *MsgArea* contains an SQLCA that describes the error.

16 - DSNLEUSR terminated because the DB2 subsystem is not in DB2 Version 8 new-function mode. MsgArea contains an SQLCA that describes the error.

This is an output parameter of type INTEGER.

- ***MsgArea:*** Contains information about DSNLEUSR execution. The information that is returned is described in the *ReturnCode* description.

  This is an output parameter of type VARCHAR(500).

Output

If DSNLEUSR executes successfully, it inserts a row into SYSIBM.USERNAMES with encrypted values for the NEWAUTHID and PASSWORD columns and returns 0 for the *ReturnCode* parameter value. If DSNLEUSR does not execute successfully, it returns a non-zero value for the *ReturnCode* value and additional diagnostic information for the *MsgArea* parameter value.

SYSPROC.DSNAIMS

DSNAIMS is a stored procedure that allows DB2 applications to invoke IMS transactions and commands easily, without maintaining their own connections to IMS.

DSNAIMS uses the IMS Open Transaction Manager Access (OTMA) API to connect to IMS and execute the transactions.

Environment

DSNAIMS runs in a WLM-established stored procedures address space. DSNAIMS (and DSNAIMS2) can share, for simplicity, a WLM environment, such as the DSNWLM_GENERAL of Table 24-13 on page 504, since they do not require any special STEPLIB, DD or NUMTCB. For performance, you might want to allocate DSNAIMS (and DSNIMS2) to a separate, dedicated DSNWLM_IMS WLM environment with the same characteristics as DSNWLM_GENERAL.

DSNAIMS requires DB2 with RRSAF enabled and IMS version 7 or later with OTMA Callable Interface enabled.

To use a two-phase commit process, you must have IMS Version 8 with UQ70789 or later.

Authorization

To set up and run DSNAIMS, you must be authorized the perform the following steps:

1. Use the job DSNTIJIM to issue the CREATE PROCEDURE statement for DSNAIMS and to grant the execution of DSNAIMS to PUBLIC. DSNTIJIM is provided in the SDSNSAMP data set. You need to customize DSNTIJIM to fit the parameters of your system.
2. Ensure that OTMA C/I is initialized. See *IMS Open Transaction Manager Access Guide and Reference* for an explanation of the C/I initialization.

Syntax

The syntax diagram in Figure 24-19 shows the SQL CALL statement for DSNAIMS.

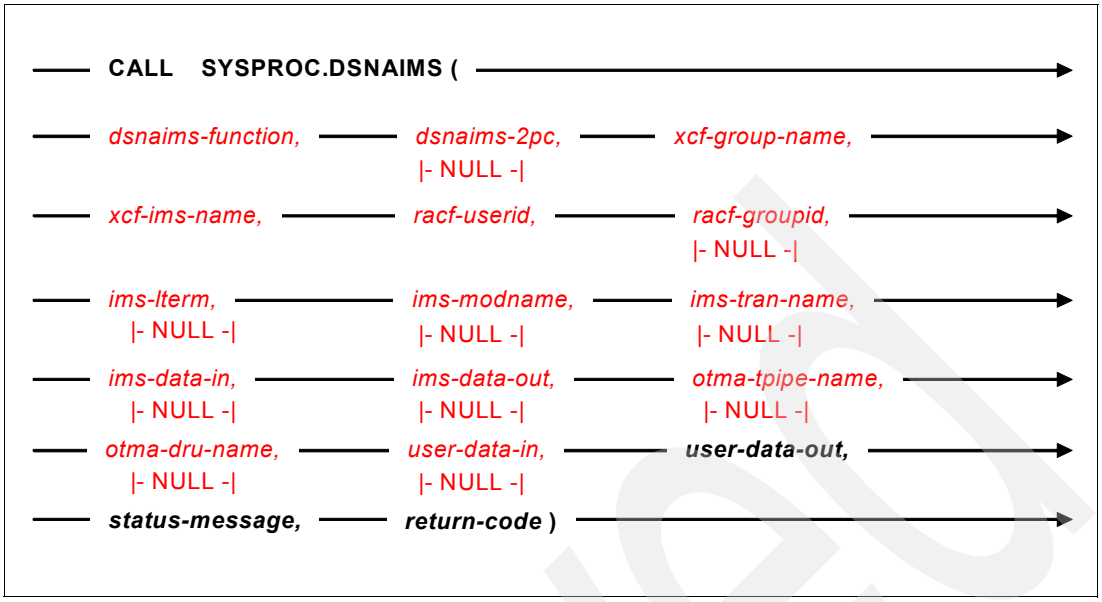

Figure 24-19 CALL DSNAIMS stored procedure

DSNAIMS option description

- ***dsnaims-function:*** A string that indicates whether the transaction is send-only, receive-only, or send-and-receive. Possible values are:

  SENDRECV - Sends and receives IMS data. SENDRECV invokes an IMS transaction or command and returns the result to the caller. The transaction can be an IMS full function or a fast path. SENDRECV does not support multiple iterations of a conversational transaction

  SEND - Sends IMS data. SEND invokes an IMS transaction or command, but does not receive IMS data. If result data exists, it can be retrieved with the RECEIVE function. A send-only transaction cannot be an IMS fast path transaction or a conversations transaction.

  RECEIVE - Receives IMS data. The data can be the result of a transaction or command initiated by the SEND function or an unsolicited output message from an IMS application. The RECEIVE function does not initiate an IMS transaction or command.

- ***dsnaims-2pc:*** Specifies whether to use a two-phase commit process to perform the transaction syncpoint service. Possible values are Y or N. For N, commits and rollbacks that are issued by the IMS transaction do not affect commit and rollback processing in the DB2 application that invokes DSNAIMS. Furthermore, IMS resources are not affected by commits and rollbacks that are issued by the calling DB2 application. If you specify Y, you must also specify SENDRECV. To use a two-phase commit process, you must set the IMS control region parameter (RRS) to Y.

  This parameter is optional. The default is N.

- ***xcf-group-name:*** Specifies the XCF group name that the IMS OTMA joins. You can obtain this name by viewing the GRNAME parameter in IMS PROCLIB member DFSPBxxx or by using the IMS command /DISPLAY OTMA.

- ***xcf-ims-name:*** Specifies the XCF member name that IMS uses for the XCF group. If IMS is not using the XRF or RSR feature, you can obtain the XCF member name from the OTMANM parameter in IMS PROCLIB member DFSPBxxx. If IMS is using the XRF or

RSR feature, you can obtain the XCF member name from the USERVAR parameter in IMS PROCLIB member DFSPBxxx.

- ***racf-userid:*** Specifies the RACF user ID that is used for IMS to perform the transaction or command authorization checking. This parameter is required if DSNAIMS is running APF-authorized. If DSNAIMS is running unauthorized, this parameter is ignored and the EXTERNAL SECURITY setting for the DSNAIMS stored procedure definition determines the user ID that is used by IMS.
- ***racf-groupid:*** Specifies the RACF group ID that is used for IMS to perform the transaction or command authorization checking. This field is used for stored procedures that are APF-authorized. It is ignored for other stored procedures.
- ***ims-lterm:*** Specifies an IMS LTERM name that is used to override the LTERM name in the I/O program communication block of the IMS application program. This field is used as an input and an output field:
  - For SENDRECV, the value is sent to IMS on input and can be updated by IMS on output.
  - For SEND, the parameter is IN only.
  - For RECEIVE, the parameter is OUT only.

  An empty or NULL value tells IMS to ignore the parameter.
- ***ims-modname:*** Specifies the formatting map name that is used by the server to map output data streams, such as 3270 streams. Although this invocation does not have IMS MFS support, the input MODNAME can be used as the map name to define the output data stream. This name is an 8-byte message output descriptor name that is placed in the I/O program communication block. When the message is inserted, IMS places this name in the message prefix with the map name in the program communication block of the IMS application program.

  For SENDRECV, the value is sent to IMS on input, and can be updated on output. For SEND, the parameter is IN only. For RECEIVE it is OUT only. IMS ignores the parameter when it is an empty or NULL value.
- ***ims-tran-name:*** Specifies the name of an IMS transaction or command that is sent to IMS. If the IMS command is longer than eight characters, specify the first eight characters (including the "/" of the command). Specify the remaining characters of the command in the ims-tran-name parameter. If you use an empty or NULL value, you must specify the full transaction name or command in the IMS-DATA-IN parameter.
- ***ims-data-in:*** Specifies the data that is sent to IMS. This parameter is required in each of the following cases:
  - Input data is required for IMS
  - No transaction name or command is passed in ims-tran-name
  - The command is longer than eight characters

  This parameter is ignored when for RECEIVE functions.
- ***ims-data-out:*** Data returned after successful completion of the transaction. This parameter is required for SENDRECV and RECEIVE functions. The parameter is ignored for SEND functions.
- ***otma-tpipe-name:*** Specifies an 8-byte user-defined communication session name that IMS uses for the input and output data for the transaction or the command in a SEND or a RECEIVE function. If the otma_tpipe_name parameter is used for a SEND function to generate an IMS output message, the same otma_pipe_name must be used to retrieve output data for the subsequent RECEIVE function.
- ***otma-dru-name:*** Specifies the name of an IMS user-defined exit routine, OTMA destination resolution user exit routine, if it is used. This IMS exit routine can format part of

the output prefix and can determine the output destination for an IMS ALT_PCB output. If an empty or null value is passed, IMS ignores this parameter.

***user-data-in**:* This optional parameter contains any data that is to be included in the IMS message prefix, so that the data can be accessed by IMS OTMA user exit routines (DFSYIOE0 and DFSYDRU0) and can be tracked by IMS log records. IMS applications that run in dependent regions do not access this data. The specified user data is not included in the output message prefix. You can use this parameter to store input and output correlator tokens or other information. It is ignored for RECEIXML schema processing.

The set of stored procedures that is provided to you for registration and removal of XML schema repositories is a collection of five stored procedures and a user-defined function, shown in Table 24-8.

Table 24-32 XML schema repository stored procedures

Name	Original name	Function	APF - authorized	Program - controlled
XSR_REGISTER	none	First procedure in XML schema registration process	No	No
XSR_ADDSCHEMADOC	none	Add every XML schema other than the primary XML schema document	No	No
XSR_COMPLETE	none	Final procedure in XML schema registration process	No	No
XSR_REMOVE	none	Remove all components of an XML schema	No	No
XDBDECOMPXML	none	Extracts values from serialized XML data and populates relational tables with the values	No	No

- VE functions.
- ***user-data-out:*** On output, this field contains the user-data-in in the IMS output prefix. IMS user exit routines (DFSYIOE0 and DFSYDRU0) can also create user-data-out for SENDRECV and RECEIVE functions. The parameter is not updated for SEND functions.
- ***status-message:*** Indicates any error message that is returned from the transaction or command, OTMA, RRS, or DSNAIMS.
- ***return-code:*** Indicates the return code that is returned for the transaction or command, OTMA, RRS, or DSNAIMS.

Connecting to multiple IMS subsystems with DSNAIMS

By default DSNAIMS connects to only one IMS subsystem at a time. The first request to DSNAIMS determines to which IMS subsystem the stored procedure connects. DSNAIMS attempts to reconnect to IMS only in the following cases:

- IMS is restarted and the saved connection is no longer valid.
- WLM loads another DSNAIMS task.

To connect to multiple IMS subsystems simultaneously, perform the following steps:

1. Make a copy of the DB2-supplied job DSNTIJIM and customize it to your environment.

2. Change the procedure name from SYSPROCC.DSNAIMS to another name, such as DSNAIMSB.
3. Change the WLM environment to a different name for DSNAIMSB.
4. Do no change the EXTERNAL NAME option. Leave it as DSNAIMS.
5. Run the new job to create a second instance of the stored procedure.
6. To ensure that you connect to the intended IMS target, consistently use the XFC group and member names that you associate with each stored procedure instance. For example:

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMS7GRP", "IMS7TMEM", ...)
CALL SYSPROC.DSNAIMSB("SENDRECV", "N", "IMS8GRP", "IMS8TMEM", ...)
```

SYSPROC.DSNAIMS2

DSNAIMS2 has the same characteristics of DSNAIMS above, with the addition of IMS multi-segment transaction support implemented with the new IN parameter OTMA_DATA_INSEG. For details on using the parameters, see also “Using the DSNAIMS2 stored procedure” on page 489.

24.2.5 Utility execution

The following procedures belong to this group:

- SYSPROC.ADMIN_UTL_SCHEDULE
- SYSPROC.ADMIN_UTL_SORT
- SYSPROC.ADMIN_TASK_ADD
- SYSPROC.ADMIN_TASK_REMOVE

SYSPROC.ADMIN_UTL_SCHEDULE

The SYSPROC.ADMIN_UTL_SCHEDULE stored procedure allows for parallel utility execution.

Load module name: DSNADMUM

Package name: DSNADMUM

Figure 24-20 illustrates the ADMIN_UTL_SCHEDULE CALL syntax.

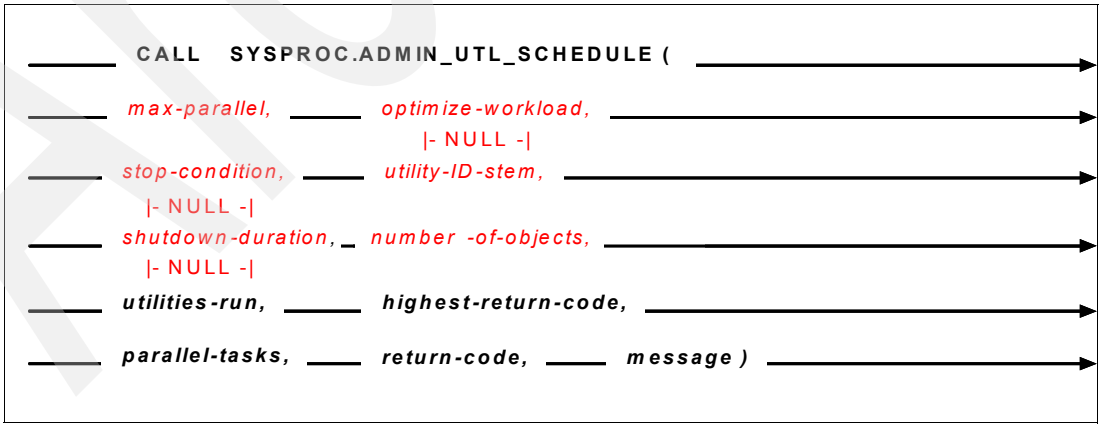

Figure 24-20 CALL ADMIN_UTL_SCHEDULE stored procedure

ADMIN_UTL_SCHEDULE option description

- ***max-parallel:*** Specifies the maximum number of parallel threads that may be started. The actual number may be lower than the requested number based on the optimizing sort result. Possible values are: 1 to 99.

  This is an input parameter of type SMALLINT and cannot be null.

- ***optimize-workload:*** Specifies whether the parallel utility executions should be sorted to achieve shortest overall execution time. Possible values are:

  NO or null - The workload is not to be sorted.

  YES - The workload is to be sorted.

  This is an input parameter of type VARCHAR(8). The default is NO.

- ***stop-condition:*** Specifies the utility execution condition after which ADMIN_UTL_SCHEDULE will not continue starting new utility executions in parallel, but will wait until all currently running utilities have completed and will then return to the caller. Possible values are:

  AUTHORIZ or null: No new utility executions will be started after one of the currently running utilities has encountered a return code from DSNUTILU of 12 or higher.

  WARNING: No new utility executions will be started after one of the currently running utilities has encountered a return code from DSNUTILU of 4 or higher.

  ERROR: No new utility executions will be started after one of the currently running utilities has encountered a return code from DSNUTILU of 8 or higher.

  This is an input parameter of type VARCHAR(8). The default is AUTHORIZ.

- ***utility-ID-stem:*** Specifies the first part of the utility ID of a utility execution in a parallel thread. The complete utility ID is dynamically created in the form *utility-ID-stem* followed by *TT* followed by *NNNNNN*, where:

  TT - The zero-padded number of the subtask executing the utility

  NNNNNN - A consecutive number of utilities executed in a subtask.

  For example, utilityidstem02000005 is the fifth utility execution that has been processed by the second subtask.

  This is an input parameter of type VARCHAR(8) and cannot be null.

- ***shutdown-duration:*** Specifies the number of seconds ADMIN_UTL_SCHEDULE will wait for a utility execution to complete before a shutdown is initiated. When a shutdown is initiated, current utility executions can run to completion, and no new utility will be started. Possible values are:

  null - A shutdown will not be performed.

  1 to 999999999999999 - A shutdown will be performed after this many seconds.

  This is an input parameter of type FLOAT(8). The default is null.

- ***number-of-objects:***

  input - Specifies the number of utility executions and their sorting objects that were passed in the SYSIBM.UTILITY_OBJ. Possible values are: 1 to 999999.

  output - Specifies the number of objects that were passed in SYSIBM.UTILITY_OBJ table that are found in the DB2 catalog.

  This is an input/output parameter of type INTEGER and cannot be null.

- ***utilities-run:*** Indicates the number of actual utility executions.

  This is an output parameter of type INTEGER.

- ***highest-return-code:*** Highest return code from DSNUTILU for all utility executions.

  This is an output parameter of type INTEGER.

- ***parallel-tasks:*** Indicates the actual number of parallel tasks that were started to execute the utility in parallel.

  This is an output parameter of the type SMALLINT.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - All parallel utility executions ran successfully.

  4 - The statistics for one or more sorting objects have not been gathered in the catalog.

  12 - An ADMIN_UTL_SCHEDULE error occurred or all the objects passed in the SYSIBM.UTILITY_OBJ table are not found in the DB2 catalog. The *message* parameter will contain details.

  This is an output parameter of type INTEGER

  Note: The ADMIN_UTL_SCHEDULE return code is different from the DSNUTILU *highest-return-code*. If for instance *stop-condition* AUTHORIZ is selected, and all passed utility executions end in a DSNUTILU *return code* of 8, ADMIN_UTL_SCHEDULE will still return *return-code* 0, even though the highest DSNUTILU return code is 8. If ADMIN_UTL_SCHEDULE encountered an internal error, such as subtasks cannot be started due to lack of memory in the address space, ADMIN_UTL_SCHEDULE will return a *return-code* higher than 4.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

Additional ADMIN_UTL_SCHEDULE input

In addition to the input parameters, the stored procedure reads from the created global temporary tables SYSIBM.UTILITY_OBJ and SYSIBM.UTILITY_STMT.

The stored procedure reads objects for utility execution from SYSIBM.UTILITY_OBJ. Table 24-33 shows the format of the created global temporary table SYSIBM.UTILITY_OBJ.

Table 24-33 SYSIBM.UTILITY_OBJ format of input row

Column name	Data type	Contents
OBJECTID	INTEGER	A unique positive identifier for the object the utility execution is associated with. When you insert multiple rows, increment OBJECTID by 1 for every insert, starting at 0.
STMTID	INTEGER	A statement row in SYSIBM.UTILITY_STMT.
TYPE	VARCHAR(10)	Object type: - TABLESPACE - INDEXSPACE - TABLE - INDEX - STOGROUP

Column name	Data type	Contents
QUALIFIER	VARCHAR(128)	Qualifier (Database or Creator) of the object in NAME, empty or null for STOGROUP. If the qualifier is not provided and the type of the object is TABLESPACE or INDEXSPACE, then the default database is DSNDB04. If the object is of the type TABLE or INDEX, the schema is the current SQL authorization ID.
NAME	VARCHAR(128)	Unqualified name of the object. NAME cannot be null. If the object no longer exists, it will be ignored and the corresponding utility will not be executed.
PART	SMALLINT	Partition number of the object for which the utility will be invoked. Null or 0 if the object is not partitioned.
RESTART	VARCHAR(8)	Restart parameter of DSNUTILU.
UTILITY_NAME	VARCHAR(20)	The name of the utility.UTILITY_NAME cannot be null. It is recommended to sort objects for the same utility. Possible values are: - CHECK DATA - CHECK INDEX - CHECK LOB - COPY - COPYTOCOPY - DIAGNOSE - LOAD - MERGECOPY - MODIFY RECOVERY - MODIFY STATISTICS - QUIESCE - REBUILD INDEX - RECOVER - REORG INDEX - REORG LOB - REORG TABLESPACE - REPAIR - REPORT RECOVERY - REPORT TABLESPACESET - RUNSTATS INDEX - RUNSTATS TABLESPACE - STOSPACE - UNLOAD UTILITY_NAME cannot be null.

The stored procedure reads the corresponding utility statements from SYSIBM.UTILITY_STMT. Table 24-34 shows the format of the created global temporary table SYSIBM.UTILITY_STMT.

Table 24-34 SYSIBM.UTILITY_STMT format of input row

Column name	Data type	Contents
STMTID	INTEGER	A unique positive identifier for a single utility execution statement.

Column name	Data type	Contents
STMTSEQ	INTEGER	If a utility statement exceeds 4000 characters, it can be split up and inserted into SYSIBM.UTILITY_STMT with the sequence starting at 0, and then being incremented with every insert. During the actual execution, the statement pieces are concatenated without any separation characters or blanks in between.
UTSTMT	VARCHAR(4000)	A utility statement or part of a utility statement. A placeholder &OBJECT. can be used to be replaced by the object name passed in SYSIBM.UTILITY_OBJ. A placeholder &THDINDEX. can be used to be replaced by the current thread index (01-99) of the utility being executed. You can use this when running REORG with SHRLEVEL CHANGE in parallel, so that you can specify a different mapping table to each thread of the utility execution. A placeholder &PARTNUM. can be used to be replaced by the partition number passed in SYSIBM.UTILITY_OBJ.

ADMIN_UTL_SCHEDULE output

This stored procedure returns the following output parameters:

- *number-of-objects*
- *utilities-run*
- *highest-return-code*
- *parallel-tasks*
- *return-code*
- *message*

In addition to the preceding output, the stored procedure returns two result sets.

The first result set is returned in the created global temporary table SYSIBM.UTILITY_SYSPRINT and contains the output from the individual utility executions. Table 24-35 shows the format of the created global temporary table SYSIBM.UTILITY_SYSPRINT.

Table 24-35 Result set row format for SYSIBM.UTILITY_SYSPRINT

Column name	Data type	Contents
OBJECTID	INTEGER	A unique positive identifier for the object the utility execution is associated with.
TEXTSEQ	INTEGER	Sequence number of utility execution output statements for the object whose unique identifier is specified in the OBJECTID column.
TEXT	VARCHAR(254)	A utility execution output statement.

The second result set is returned in the created global temporary table SYSIBM.UTILITY_RETCODE and contains the return code for each of the individual DSNUTILU executions. Table 24-36 shows the format of the output created global temporary table SYSIBM.UTILITY_RETCODE.

Table 24-36 Result set row format for SYSIBM.UTILITY_RETCODE

Column name	Data type	Contents
OBJECTID	INTEGER	A unique positive identifier for the object the utility execution is associated with.
RETCODE	INTEGER	Return code from DSNUTILU for this utility execution.

ADMIN_UTL_SCHEDULE usage notes

Exclusive and deferred utility execution on special objects

When the sort object is one of the following objects:

- Table spaces
  - DSNDB01.SYSUTILX
  - DSNDB06.SYSCOPY
  - DSNDB01.SYSLGRNX
- Tables
  - SYSIBM.SYSUTILX
  - SYSIBM.SYSCOPY
  - SYSIBM.SYSLGRNX
- Index spaces
  - DSNDB01.DSNLUX01
  - DSNDB01.DSNLUX02
- Indexes
  - SYSIBM.DSNLUX01
  - SYSIBM.DSNLUX02

the corresponding utility execution will be deferred and utility execution on these objects will be exclusive. That is, ADMIN_UTL_SCHEDULE waits until all parallel tasks have completed utility execution, and then these exclusive objects will be executed sequentially.

Intra-utility parallelism

Whenever possible, intra-utility parallelism should be used (even through ADMIN_UTL_SCHEDULE) for the utilities that support it, such as COPY, to achieve optimal results.

How the optimizer sorts the workload

A workload is sorted based on the utility requested and the catalog information. If COPY was requested for three tablespaces to be performed in parallel (TS1, TS2, TS3). And TS1 and TS2 have the same size and TS3 is 10 times larger than TS1, then the optimizer would recommend to use two threads and would execute TS1, TS2 sequentially in one thread, and TS3 in the second. The utilities would not finish faster if three threads were used because the anticipated runtime for TS1 and TS2 together is shorter than for TS3. Furthermore, a third task would create undesirable contention for system resources.

Hence, the optimizer relies on current catalog statistics gathered by the RUNSTATS utility. If the catalog information is not current, or has only partially been gathered, optimization may not be always optimal.

How to achieve best results

Since many utilities support multiple objects as parameters (such as RUNSTATS INDEX allows multiple indexes in its control statement as long as they belong to the same

tablespace), sometimes the question arises whether or not a utility such as RUNSTATS INDEX is better processed as a single utility execution or as individual utility executions. In general, if a utility does not support intra-utility parallelism, it will process objects sequentially and so you have faster execution through parallel utility execution. However, 10 parallel RUNSTATS INDEX executions are not 10 times faster than a single one with 10 indexes (provided they belong to the same tablespace).

With parallel execution there may be more scans and more contention so that the overall performance gains are less than expected. However, if the objects are unrelated, parallel utility execution when used with client programs will significantly shorten total execution time, also because of a number of remote stored procedure invocations, it is reduced to 1, and no redundant utility execution data is transferred between the client and the server.

Choosing the right maximum parallelism

ADMIN_UTL_SCHEDULE allows you to specify the maximum number of parallel subtasks that may be started between 1 to 99.

Every started subtask requires a minimum of two DB2 threads, more if the utility supports intra-utility parallelism. The maximum number of allied threads that can be allocated concurrently at a subsystem is determined by the CTHREAD parameter. You may consider increasing CTHREAD to run ADMIN_UTL_SCHEDULE with higher parallelism.

SYSPROC.ADMIN_UTL_SORT

The SYSPROC.ADMIN_UTL_SORT stored procedure sorts objects for parallel utility execution using JCL or the ADMIN_UTL_SCHEDULE stored procedure.

Load module name: DSNADMUS

Package name: DSNADMUS

Figure 24-21 illustrates the ADMIN_UTL_SORT CALL syntax.

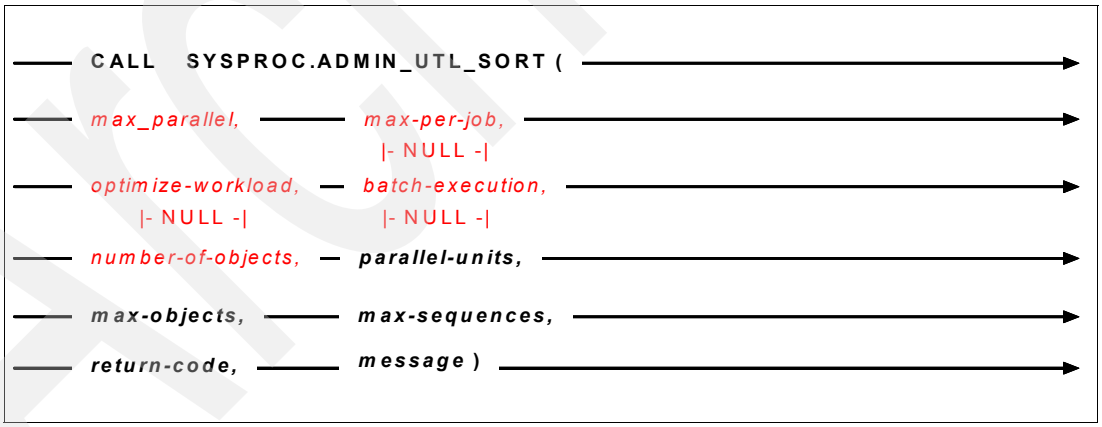

Figure 24-21 CALL ADMIN_UTL_SORT stored procedure

ADMIN_UTL_SORT option descriptions

- ***max-parallel:*** Specifies the maximum number of parallel units. The actual number may be lower than the requested number based on the optimizing sort result. Possible values are: 1 to 99.

  This is an input parameter of the type SMALLINT and cannot be null.

- ***max-per-job:*** Specifies the maximum number of steps per job for batch execution. Possible values are:

1 to 255 - Steps per job for batch execution

null - Online execution.

This is an input parameter of the type SMALLINT. This parameter cannot be null if *batch-execution* is YES.

- ***optimize-workload:*** Specifies whether the parallel units should be sorted to achieve shortest overall execution time. Possible values are:

  NO or null - The workload is not to be sorted.

  YES - The workload is to be sorted.

  This is an input parameter of type VARCHAR(8). The default value is NO.

- ***batch-execution:*** Indicates whether the objects should be sorted for online or batch (JCL) execution.

  NO or null - The workload is for online execution.

  YES - The workload is for batch execution.

  This is an input parameter of type VARCHAR(8). The default value is NO.

- ***number-of-objects:***

  input - Specifies the number of objects that were passed in SYSIBM.UTILITY_SORT_OBJ. Possible values are: 1 to 999999.

  output - Specifies the number of objects that were passed in SYSIBM.UTILITY_SORT_OBJ table that are found in the DB2 catalog.

  This is an input/output parameter of type INTEGER and cannot be null.

- ***parallel-units:*** Indicates the number of recommended parallel units.

  This is an output parameter of type SMALLINT.

- ***max-objects:*** Indicates the maximum number of objects in any parallel unit.

  This is an output parameter of type INTEGER.

- ***max-sequences:*** Indicates the number of jobs in any parallel unit.

  This is an output parameter of type INTEGER.

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - Sort ran successfully.

  4 - The statistics for one or more sorting objects have not been gathered in the catalog or the object no longer exists.

  12 - An ADMIN_UTL_SORT error occurred. The *message* parameter will contain the details.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. If no error occurred, then no message is returned.

  The first messages in this area are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

Additional ADMIN_UTL_SORT input

In addition to the input parameters, this stored procedure reads the objects for sorting and the corresponding utility names from the created global temporary table SYSIBM.UTILITY_SORT_OBJ.

Table 24-37 shows the format of the created global temporary table SYSIBM.UTILITY_SORT_OBJ.

Table 24-37 SYSIBM.UTILITY_SORT_OBJ input row format

Column name	Data type	Contents
OBJECTID	INTEGER	A unique positive identifier for the object the utility execution is associated with. When you insert multiple rows, increment OBJECTID by 1 for every insert, starting at 0.
TYPE	VARCHAR(10)	Object type: - TABLESPACE - INDEXSPACE - TABLE - INDEX - STOGROUP
QUALIFIER	VARCHAR(128)	Qualifier (Database or Creator) of the object in NAME, empty or null for STOGROUP. If the qualifier is not provided and the type of the object is TABLESPACE or INDEXSPACE, then the default database is DSNDB04. If the object is of the type TABLE or INDEX, the schema is the current SQL authorization ID. If the object no longer exists, it is ignored.
NAME	VARCHAR(128)	Unqualified name of the object. NAME cannot be null.
PART	SMALLINT	Partition number of the object for which the utility will be invoked. Null or 0 if the object is not partitioned.
UTILITY_NAME	VARCHAR(20)	Utility name. UTILITY_NAME cannot be null. It is recommended to sort objects for the same utility. Possible values are: - CHECK DATA - CHECK INDEX - CHECK LOB - COPY - COPYTOCOPY - DIAGNOSE - LOAD - MERGECOPY - MODIFY RECOVERY - MODIFY STATISTICS - QUIESCE - REBUILD INDEX - RECOVER - REORG INDEX - REORG LOB - REORG TABLESPACE - REPAIR - REPORT RECOVERY - REPORT TABLESPACESET - RUNSTATS INDEX - RUNSTATS TABLESPACE - STOSPACE - UNLOAD UTILITY_NAME cannot be null.

ADMIN_UTL_SORT output

This stored procedure returns the following output parameters:

- *number-of-objects*
- *parallel-units*
- *max-objects*
- *max-sequences*
- *return-code*
- *message*

In addition to the preceding output, the stored procedure returns one result set that contains the objects sorted into parallel execution units.

Table 24-38 shows the format of the result set returned in the created global temporary table SYSIBM.UTILITY_SORT_OUT.

Table 24-38 SYSIBM.UTILITY_SORT_OUT result set format

Column name	Data type	Contents
OBJECTID	INTEGER	A unique positive identifier for the object
UNIT	SMALLINT	Number of parallel execution units
UNIT_SEQ	INTEGER	Job sequence within parallel execution unit
UNIT_SEQ_POS	INTEGER	Step within job
EXCLUSIVE	CHAR(1)	Requires execution with nothing running in parallel.

24.3 Scheduling administrative stored procedures with the DB2 task scheduler

DB2 for z/OS V8 and DB2 9 for z/OS now provide out-of-the-box scheduling capabilities. The DB2 scheduler enables for server side scheduling and hence asynchronous execution of tasks. Tasks can be stored procedures as well as JCL jobs, they are executed according to a time or an event based schedule on behalf of a DB2 user. This allows for automation of administrative and monitoring tasks such as recovery, statistics collection and backups. In this chapter we provide a use case centric overview that shows how to employ the scheduler vehicle to ease and automate the execution of stored procedures. The scheduler SQL interface that is described here, is installed with the PTFs UK32046 (for Version 8) as well as UK32047 (for Version 9).

To work with the scheduler, it is crucial to understand the life cycle of a scheduled task. We therefore recommend to read “24.3.1, “A brief functional overview” on page 569” and “24.3.2, “Interacting with the scheduler” on page 570”. 24.3.3, “Scheduling stored procedures” on page 575” has the layout of a checklist, that is, when scheduling a task you should figure out the proper parameter settings by only reading the paragraphs that really apply to your specific scheduling scenario.

24.3.1 A brief functional overview

The task scheduler is installed as part of DB2 for z/OS V8 and V9. With the first startup of DB2, the scheduler is available and operable and can be accessed through a SQL interface. When DB2 stops, the scheduler stays up and running. This allows for example to run administrative tasks triggered by the DB2 stop or to recurringly run JCL jobs that do not

require an online DB2. Scheduler has to be restarted only for maintenance reasons. Tasks are stored in two redundant task lists which ensures high availability and recovery.

24.3.2 Interacting with the scheduler

The user interacts with the scheduler through an SQL API, which consists of two stored procedures and two user-defined functions (UDFs). With this interface, a user is capable of scheduling and removing tasks, as well as listing tasks and their last execution status. This SQL API does not have transactional characteristics. This means that if you issue a ROLLBACK after you scheduled a task, this task is not removed from the scheduler.

Before working with scheduler it is crucial to understand the lifecycle of a scheduled task as this implies to differentiate between scheduling a task, executing a task and controlling the task status. The following section provides a chronological rough overview of such a scheduled task lifecycle.

Scheduling a new task

A user or an application schedules a new task by simply calling the procedure SYSPROC.ADMIN_TASK_ADD. The only requirement imposed to schedule a task, is that the caller has execution rights granted on this stored procedure. The stored procedure parameters provide information for the following four major scheduling categories:

- What task has to be executed?
- When has the task to be executed?
- Under which security context has the task to be executed?
- Where has the task to be executed?

The parameters are verified and if they are found to be correct, the task is added to the scheduler's task lists, and the next execution of the task is scheduled. If no task name has been specified when calling the procedure, a system-generated task name is returned in the task_name output parameter. The return code and message output parameters provide information about possible scheduling errors.

Figure 24-22 shows a high-level overview of the components involved when a task is added or removed. The left frame stating "V91AMSTR / V91ADBM1" depicts the DB2 address spaces that implement the scheduler SQL interface. The right frame labeled "V91AADMT"illustrates the actual task scheduler, which runs in its own address space. A user or an application program calls one of the DB2-supplied scheduling stored procedures: SYSPROC.ADMIN_TASK_ADD or SYSPROC.ADMIN_TASK_REMOVE. More information about the removal of tasks can be found in "Removing a scheduled task" on page 575".

The procedure passes the provided scheduling parameters on to the scheduler. If the scheduler finds the parameters to be correct, it determines the next point in time when the scheduled task has to be executed. The scheduler then also materializes the task into two task lists, which are essentially the DB2 table SYSIBM.ADMIN_TASKS as well as a redundant VSAM data set.

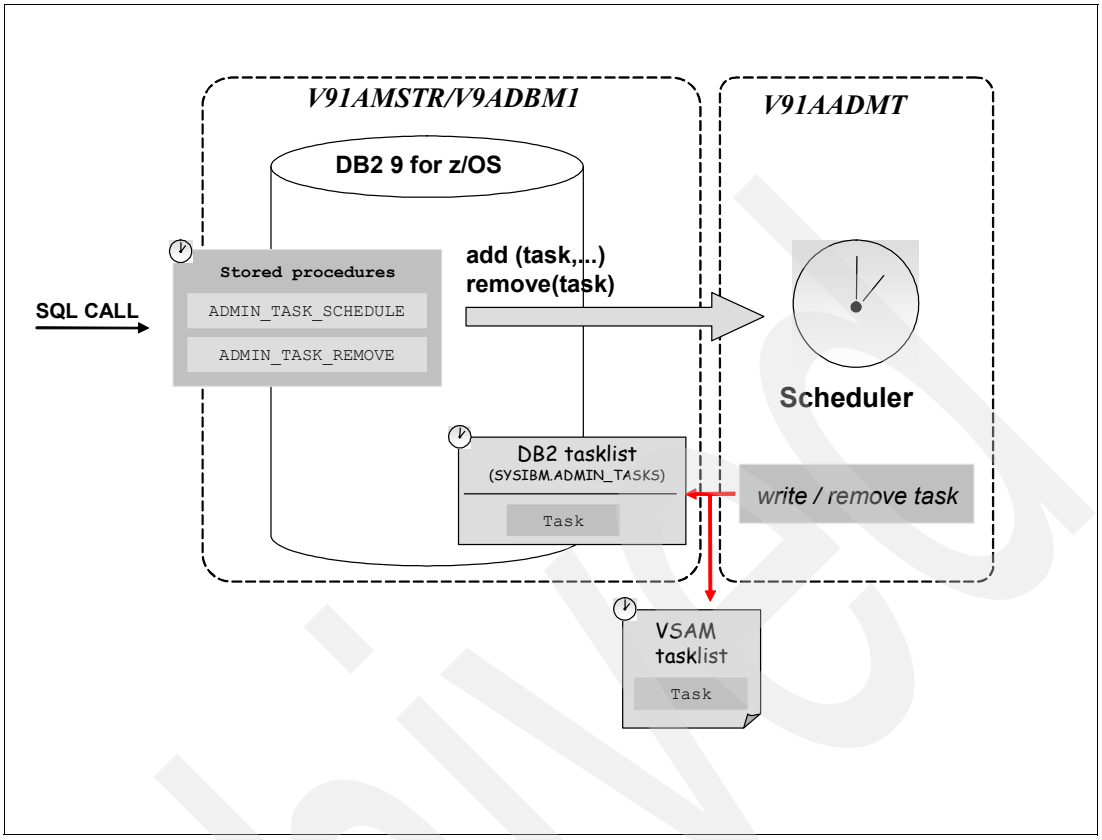

Figure 24-22 Schedule/Remove a task with the DB2 provided scheduler

For a complete syntax diagram of the ADMIN_TASK_ADD stored procedure, refer to “SYSPROC.ADMIN_TASK_ADD” on page 583 or *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840.

Listing the scheduled tasks

As soon as a task has been added to the task lists, it can be listed. A user who has execute rights granted on the UDF can view all scheduled tasks by just employing the table function DSNADM.ADMIN_TASK_LIST. The UDF returns one row for each task that has been scheduled. The parameters that have been specified when the task has been added are now column values of the returned table. In addition to the task parameters two more columns are returned which provide information about the CREATOR of the task and when the task has been LAST_MODIFIED.

The following SQL statement lists all task names that have been created under the user ID PAOLOR3 and sorts them according to their LAST_MODIFIED timestamp.

```
SELECT TASK_NAME, BEGIN_TIMESTAMP
   FROM TABLE(DSNADM.ADMIN_TASK_LIST()) AS TASKLIST
   WHERE CREATOR = 'PAOLOR3'
   ORDER BY LAST_MODIFIED;
```

Example 24-13 shows the output when executing the query in a dynamic SQL processing like SPUFI. Because of the LAST_MODIFIED predicate, the tasks are displayed in the same order in which they have been added to the task list.

Example 24-13 SPUFI output for ADMIN_TASK_LIST

```
---------+---------+---------+---------+---------+---------+---------+---------+---------
TASK_NAME                      LAST_MODIFIED
---------+---------+---------+---------+---------+---------+---------+---------+---------
REORG_JOB                      2007-12-19-09.05.00.000000
RUNSTATS_JOB                   2007-12-19-09.21.00.000000
TASK_ID_0003                   2007-12-19-09.44.00.000000
DSNE610I NUMBER OF ROWS DISPLAYED IS 3
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
---------+---------+---------+---------+---------+---------+---------+---------+---------
```

The task list contains three tasks, where the task TASK_ID_0003 is a system-generated task name. This generic task name is always created by the system whenever there is no task name specified in the scheduling parameters. The four-digit number always refers to task number *nnnn* in the task list and makes this name unique.

For more detailed information about the columns returned, refer to *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

Figure 24-23 illustrates the major components involved when a request to the table functions ADMIN_TASK_LIST or ADMIN_TASK_STATUS is made. The two major frames in this picture are the DB2 address spaces (V91AMSTR/V91ADBM1) as well as the scheduler address space (V91AADMT). When an SQL statement queries for example ADMIN_TASK_LIST, the scheduler is notified to ensure that both task lists (DB2, VSAM) are consistent. This is triggered by the illustrated synch() call. After that, the table function accesses the task list table SYSIBM.ADMIN_TASKS to retrieve and prepare the requested information. ADMIN_TASK_STATUS is explained in the section “Listing the execution status of the scheduled tasks” on page 574”.

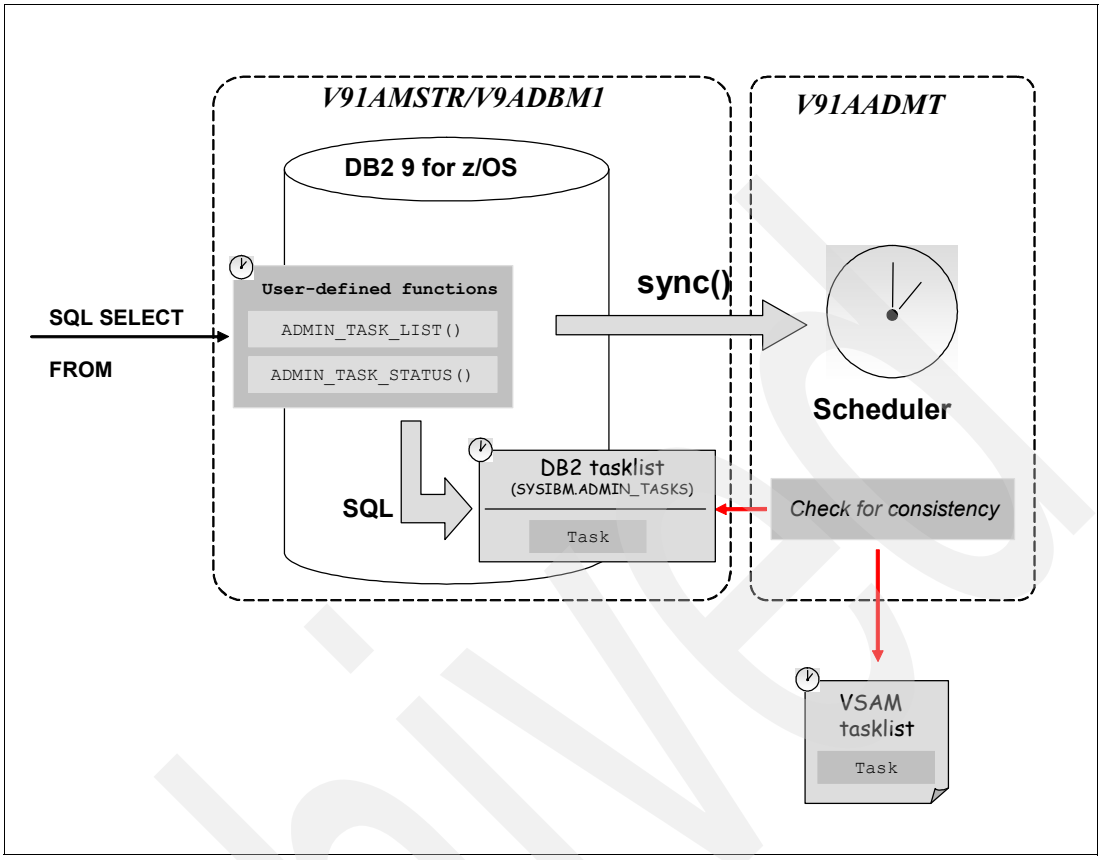

Figure 24-23 List the scheduled tasks and task status

For a complete description of the returned table, refer to *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

Executing a scheduled task

The scheduler executes a task either at the next point in time defined in a task schedule, or if a triggering event for a task occurs, such as a DB2 start event. Scheduler then first switches the security context to run under the user authority indicated in the scheduled task, or under a default user authority. After that, it calls either the defined stored procedure with the given parameters, or the defined JCL job. At the end of the execution, the scheduler stores the task execution status.

Figure 24-24 on page 574 provides an architectural overview of the components involved when the scheduler wakes up to execute a scheduled call to the DB2-supplied stored procedure WLM_REFRESH. As indicated with the *read task* box, the scheduler first reads the task which is to be executed from the task lists. It then uses a TCB to first switch the security context and then execute the CALL to the stored procedure on its attached distinct DB2 subsystem. When the CALL returns, the scheduler gathers possible return codes, messages, or SQLCODES, and populates the execution status columns in the two task lists, which is indicated with the *write execution status* box.

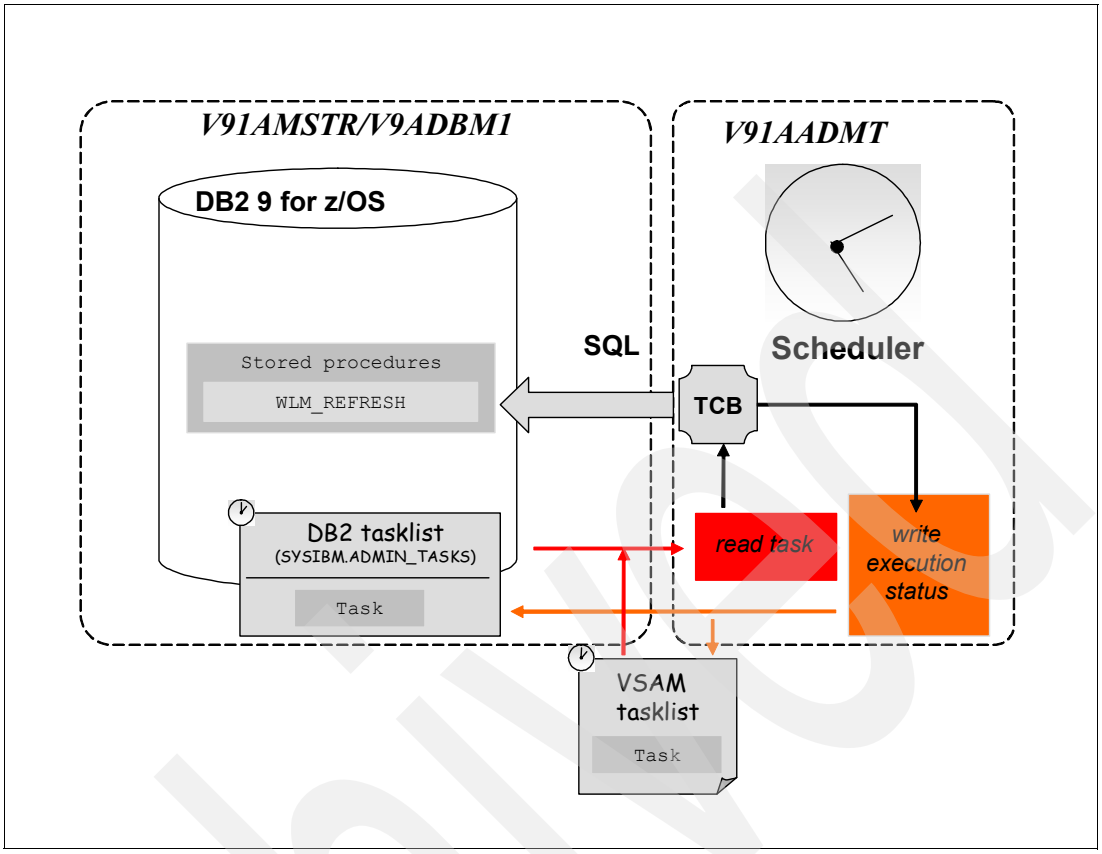

Figure 24-24 Executing a scheduled stored procedure task

Listing the execution status of the scheduled tasks

As soon as a task has been added with the help of the ADMIN_TASK_ADD stored procedure, the execution status of this task can be listed; however, at this point in time it contains NULL values. As soon as the task is executed for a first time, a non-null execution status is available. A user that has execution rights granted on the UDF lists the last execution status of all tasks by employing the table function DSNADM.ADMIN_TASK_STATUS. An SQL statement that queries the table function ADMIN_TASK_STATUS might look like the following:

```
SELECT * FROM TABLE(DSNADM.ADMIN_TASK_STATUS()) AS TASKSTATUS;
```

Example 24-14 shows the respective partial SPUFI output. Note, that the STATUS columns for task TASK_ID_0003 are not initialized, since the stored procedures have not been executed yet. The task execution for REORG_JOB has already completed, and the RUNSTATS_JOB is currently in its first execution.

Example 24-14 Partial ADMIN_TASK_STATUS output

```
---------+---------+---------+---------+---------+---------+---------+---------
TASK_NAME           STATUS      NUM_INVOCATIONS  START_TIMESTAMP
---------+---------+---------+---------+---------+---------+---------+--------
REORG_JOB           COMPLETED                 1  2007-12-19-09.07.00.000000
RUNSTATS_JOB        RUNNING                   1  2007-12-19-09.30.00.000000
TASK_ID_0003        ----------  ---------------  --------------------
DSNE610I NUMBER OF ROWS DISPLAYED IS 3
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
---------+---------+---------+---------+---------+---------+---------+---------
```

It is also possible to combine the ADMIN_TASK_LIST and ADMIN_TASK_STATUS output to realize more complex use cases. Example 24-15 shows that the query returns all tasks that are still active, that is, that will be executed in the future:

Example 24-15 List all active tasks

```
SELECT T.TASK_NAME
  FROM TABLE (DSNADM.ADMIN_TASK_LIST()) T,
       TABLE (DSNADM.ADMIN_TASK_STATUS()) S
  WHERE T.TASK_NAME = S.TASK_NAME AND
      ( S.NUM_INVOCATIONS IS NULL OR
        T.MAX_INVOCATIONS IS NULL OR
        S.NUM_INVOCATIONS < T.MAX_INVOCATIONS) AND
        T.END_TIMESTAMP > CURRENT TIMESTAMP
```

The query joins the two table functions on the TASK_NAME column. A task is then considered to be still valid, in case the END_TIMESTAMP column value is not expired, and basically the MAX_INVOCATIONS value is not yet reached.

For a high-level overview of the component involved when the task status is queried, refer to Figure 24-23 on page 573.

For a complete description of the returned table, refer to *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

Removing a scheduled task

An authorized DB2 user, who has the proper execution rights granted, can finally remove a scheduled task from the task list by calling the stored procedure SYSPROC.ADMIN_TASK_REMOVE. The task to be removed is identified by the TASK_NAME input parameter. Only the user that scheduled the task can remove it, or any user with SYSOPR, SYSCTRL, or SYSADM authority. The task can only be removed if it does not trigger the execution of other tasks, and if it is not currently executing. The respective return code and message are returned as output parameters.

Note: The scheduler does not perform any housekeeping of tasks that are no longer valid. That means that the user should always remove tasks that are not supposed to be executed by the scheduler anymore.

For example, a program could call the table functions ADMIN_TASK_LIST / STATUS to determine the tasks that are expired, and invoke the procedure ADMIN_TASK_REMOVE on the returned TASK_NAME column values. For a sample program that illustrates such housekeeping, refer to A.9.5, “Housekeeping with the scheduler” on page 866.

For a high-level overview of the components involved when calling ADMIN_TASK_REMOVE, see Figure 24-22 on page 571.

For a complete syntax diagram of the ADMIN_TASK_REMOVE stored procedure, refer to “SYSPROC.ADMIN_TASK_REMOVE” on page 588 or *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840.

24.3.3 Scheduling stored procedures

When scheduling an administrative task, it is crucial to understand the specific parameters that apply for a distinct scheduling scenario and how these parameters have to be initialized.

The following section provides a use case oriented checklist that a user should walk through in order to set the correct parameters for a desired scheduling scenario.

What task has to be executed?

An application or a user can schedule either stored procedures or JCL jobs. Stored procedures have the advantage that they closely integrate with DB2, but can only be invoked by the scheduler as long as DB2 is up and running, whereas JCL jobs can still be executed by the scheduler even if DB2 is shut down. However, JCL tasks do not have a comparable direct access to DB2 resources. The following section exclusively discusses the scheduling of stored procedures. For information about how to schedule JCL jobs, refer to *DB2 Version 9.1 for z/OS Administration Guide,* SC18-9840.

- ***The administrative task is a stored procedure***

  There are six scheduling parameters that determine the type of administrative task to be scheduled. Three of them relate to JCL jobs, which are:

  ```
  JCL_LIBRARY
  JCL_MEMBER
  JOB_WAIT
  ```

  When scheduling a stored procedure, these three scheduling parameters have to be set to NULL. The other three parameters determine the stored procedure that should be scheduled; they are:

  ```
  PROCEDURE_SCHEMA
  PROCEDURE_NAME
  PROCEDURE_INPUT
  ```

  Here, PROCEDURE_NAME is the only mandatory parameter. With a nullified PROCEDURE_SCHEMA scheduling parameter, the default schema is used for qualification of the scheduled stored procedure.

  The scheduled stored procedure does not have parameters

  Setting the PROCEDURE_INPUT scheduling parameter to NULL ensures that no parameters are passed to the scheduled stored procedure, when it is invoked by the scheduler.

  The scheduled stored procedure has input parameters

  Input parameters to the scheduled stored procedure have to be provided by the PROCEDURE_INPUT scheduling parameter that contains a valid SQL SELECT statement which returns a single row of data. When a scheduled stored procedure has to be executed, scheduler first evaluates this SQL statement and uses the returned column values or literals to populate the stored procedure parameters. The scheduler here performs a positional replacement. Thus it is the users responsibility to ensure that parameter values are returned in the right sequence.

  - ***The input parameters are static***

    The input parameters of a scheduled stored procedure task are considered to be static, whenever they are known at scheduling time and do not change during the task lifetime in the scheduler, i.e. they are the same for every recurring invocation of the task. In such a case the scheduling parameter PROCEDURE_INPUT contains a SQL SELECT statement that queries the table SYSIBM.SYSDUMMY1. The statement contains literals of compatible type for every parameter of the scheduled stored procedure.

    For a stored procedure that has two input parameters, one of type CHAR and the second of type INTEGER, the following SQL statement could be used in PROCEDURE_INPUT to provide static input parameters:

    ```
    SELECT 'TEST', 11 FROM SYSIBM.SYSDUMMY1;
    ```

For a Java sample that implements static input parameters, refer to A.9.1, “Use case - 1” on page 858.

– ***The input parameters are dynamic***

Input parameters that are not known at scheduling time of a stored procedure or that might change in between two successive scheduled invocations of a stored procedure are considered to be dynamic. Here, the scheduling parameter PROCEDURE_INPUT contains a SQL SELECT statement that queries a table where the latest stored procedure parameters have been inserted by the application or the user. The SQL statement in PROCEDURE_INPUT is static, but the content of the table can be changed during the stored procedures lifetime in the scheduler. Once more the SELECT statement has to quality one row of data, with compatible column data types for every input parameter. Dynamic and static parameters (literals) can both appear in the SELECT statement at the same time.

For a Java sample that implements dynamic input parameters, refer to A.9.3, “Use case - 3” on page 863.

– ***The parameters of a scheduled stored procedure have Unicode values***

The scheduler is able to retrieve and pass Unicode parameters when invoking a scheduled stored procedure. In order to invoke a stored procedure with Unicode parameters, a user or an application has to make use of the dynamic parameter approach described above. The table containing the input parameters has to be defined with a Unicode CCSID, such that the returned column values are in Unicode when they are passed on to the scheduled stored procedure. The PROCEDURE_INPUT scheduling parameter contains an EBCDIC SQL statement. Thus, the targeted table name and its associated column names must not contain any characters that cannot be expressed in EBCDIC. Otherwise, there is a chance that characters get broken. The scheduling parameters PROCEDURE_NAME and PROCEDURE_SCHEMA are defined in EBCDIC as well and thus behave similarly.

- ***The scheduled stored procedure also returns output parameters***

Every output parameter has to be qualified in the PROCEDURE_INPUT scheduling parameter. That means the SELECT statement has to contain literals of compatible type for every output parameter of the scheduled stored procedure. Here as well it has to be assured that the literal positions in the SELECT statement fits the position of the output parameters in the scheduled stored procedure.

When scheduling the DB2-supplied stored procedure WLM_REFRESH, which has two input parameters and two output parameters of type VARCHAR and INTEGER, one could use the following SQL statement in the PROCEDURE_INPUT scheduling parameter:

```
SELECT 'WLMENV1', 'V91A', 'Message', 0 from SYSIBM.SYSDUMMY1;
```

Refer to A.9.3, “Use case - 3” on page 863 to see a Java sample that schedules a stored procedure that contains output parameters.

– ***How to access the output from the scheduled stored procedure***

The scheduler does neither store nor make output parameters accessible to the user or the calling application. It is the responsibility of the stored procedure developer to materialize the needed information in a non-transient repository, for example a DB2 table or a data set. This also holds for result sets or temporary tables. For provided stored procedures that cannot be changed, a user has to wrap those into another stored procedure which performs the result set materialization. It is the wrapping stored procedure that has to be scheduled with the scheduler. In case of a DB2 error, the following debug information is accessible to the user:

```
SQLCOD
SQLSTATE
```

```
SQLERRP
SQLERRMC
```

This information is accessible via the ADMIN_TASK_LIST table function.

- ***The stored procedure executes SQL statements***

  After executing a stored procedure, the scheduler always performs a COMMIT. This behavior is independent of a successful or unsuccessful invocation of the stored procedure.

When does the task have to be executed?

The scheduler can execute a stored procedure once or many times, at certain fixed points in time, triggered by events like DB2 startup, or triggered by the execution of another task. The scheduling parameters of ADMIN_TASK_ADD allow the user or the application to determine when the task will be executed.

- ***Defining the schedule***

  There are five parameters that determine the scheduling behavior of the stored procedure. These are the time-based parameters

```
INTERVAL
POINT_IN_TIME
```

  as well as the event-based parameters

```
TRIGGER_TASK_NAME
TRIGGER_TASK_COND
TRIGGER_TASK_CODE
```

  – ***The stored procedure runs only once***

    To execute a stored procedure only once, the MAX_INVOCATION parameter has to be set to 1. All 5 schedule parameters are set to NULL. The task will be executed as soon as the validity time window allows (see section “Constraining the task execution” on page 580), which is either immediately or at a later point in time.

    A.9.1, “Use case - 1” on page 858 shows a Java sample that implements the scheduling of a stored procedure with MAX_INVOCATIONS = 1.

  – ***The stored procedure is a recurring task based on regular time intervals***

    To schedule regular time-based executions, the schedule parameters have to be NULL except for the INTERVAL scheduling parameter. This parameter is set to the number of minutes between the start of two successive executions.

    By setting INTERVAL to a value of 5, for example, the task is executed regularly, several times, every 5 minutes. The first execution starts as soon as the validity time window allows (see “Constraining the task execution” on page 580). The next execution starts 5 minutes after the last execution started, assuming that the last execution terminated in the meantime. That means if the stored procedure runs for about 7 minutes, the second execution is skipped. Thus the second execution is not 5 minutes after the first execution but rather 10 minutes. Two instances of the same task can never run in parallel.

  – ***The stored procedure is a recurring task based on non regular time intervals***

    The stored procedure is executed several times at irregular points in time defined by the user or the application, for example Monday and Friday at 10 p.m. To schedule such irregular executions, the schedule parameters are NULL, except for POINT_IN_TIME, which is set to a string formatted according to the UNIX cron format.

    Execution starts at the first point in time that is within the validity time window defined by the constraining parameters (see “Constraining the task execution” on page 580).

Once more, multiple instances of the same task cannot run in parallel, thus the next execution would be skipped if the previous one is still running.

How to specify one or several non regular points in time

The cron format consists of five time and date fields each separated by at least one blank. For each field, comma-separated lists and ranges indicated by a hyphen are allowed to implement recurring task execution characteristics. The fields supported are the following:

```
minutes:      0 - 59
hour:         0 - 23
day of month: 1 - 31
month:        1 - 12
day of week:  0 – 7  here, 0 or 7 refer to a Sunday
```

The following is an example of a cron string which schedules a task to be executed at 3:12 p.m. on December 31:

```
POINT_IN_TIME = '12 15 31 12 *'
```

A stored procedure that runs repeatedly at 6 a.m. on the 11th and 12th of every month and also every Saturday and Sunday would be scheduled like this:

```
POINT_IN_TIME = '0 6 11,12 * 6-7'
```

For more information about the UNIX cron format support implemented in the scheduler, refer to Chapter 14 of *DB2 Version 9.1 for z/OS Administration Guide,* SC18-9840.

A.9.3, “Use case - 3” on page 863 shows a Java sample that makes use of the POINT_IN_TIME scheduling parameter.

– ***The stored procedure execution is triggered***

To define dependencies between sequences of stored procedures, the triggering capabilities of the scheduler can be employed. Stored procedure triggering is implemented with the scheduling parameters

```
TRIGGER_TASK_NAME
TRIGGER_TASK_COND
TRIGGER_TASK_CODE
```

When these are employed, the following time-based scheduling parameters have to be set to NULL:

```
POINT_IN_TIME
INTERVAL
```

A triggered stored procedure is only executed when the triggering task completes its execution, or a triggering event occurs. Triggered stored procedures are also constraint to a single execution instance at any point in time. This means that a triggering task running every 3 minutes triggers a stored procedure that runs for 5 minutes. The second triggering event after 6 minutes will be skipped because the first triggered procedure execution after 3 minutes is still running. The stored procedure will successfully be triggered for a second time after 9 minutes.

- ***The stored procedure execution is triggered by a predefined DB2 event***

  There are two predefined events for the scheduler: DB2START and DB2STOP. A stored procedure execution can only be triggered when DB2 starts. Reacting on a DB2-stop event, does not make sense for a stored procedure execution. To schedule such DB2-start triggered stored procedure executions, use the following scheduling parameter + value combination:

  ```
  TRIGGER_TASK_NAME = "DB2START"
  ```

- ***The stored procedure execution is triggered by the execution of another scheduled task***

  To schedule a task-triggered execution of a stored procedure, set the TRIGGER_TASK_NAME scheduling parameter to a task-name of an already defined task. A.9.4, “Use case - 4” on page 865 shows a Java sample that schedules a stored procedure that is triggered by an already existing task.

- ***The execution of a stored procedure depends on the result of the triggering task***

  If the parameters TRIGGER_TASK_CODE and TRIGGER_TASK_COND both contain valid non-null values, the triggered stored procedure is executed only when the triggering task completes with an execution status that satisfies the SQLCODE and the condition, defined in those two scheduling parameters. If these parameters are NULL, the triggered task is executed each time the triggering task completes.

  DB2 events do not have results that could be triggered on. Therefore, TRIGGER_TASK_CODE and TRIGGER_TASK_COND have to be both NULL when the stored procedure execution is triggered by a DB2 event.

  Refer to the syntax diagram for possible TRIGGER_TASK_COND values.

► **Constraining the task execution**

There are three scheduling parameters that constrain the number of executions in the schedule. These constraining parameters are:

```
BEGIN_TIMESTAMP
END_TIMESTAMP
MAX_INVOCATIONS
```

With the help of BEGIN_TIMESTAMP and END_TIMESTAMP a user or application can define a validity time window for the stored procedure, that is, the stored procedure will not start executing before or after this time window. The MAX_INVOCATION scheduling parameter defines the maximum number of invocations allowed when the stored procedure has to be repeatedly executed.

– ***The task can be executed starting immediately***

  By setting the BEGIN_TIMESTAMP parameter to NULL, the validity time window of the stored procedure starts at the time when the task is added. This means task execution can start immediately.

– ***The task has to be executed after a later point in time***

  This requires shifting the beginning of the validity time window of the scheduled stored procedure to the desired point in time. By setting the BEGIN_TIMESTAMP scheduling parameter to a specific timestamp in the future it is guaranteed that the stored procedure is not executed before the provided timestamp.

  For a Java sample with a BEGIN_TIMESTAMP set to a later point in time, refer to A.9.1, “Use case - 1” on page 858.

– ***The task should never expire***

  If the END_TIMESTAMP scheduling parameter is set to NULL, the validity time window of the stored procedure never expires. Recurring or triggered executions of stored procedures would not be limited in time, but stay valid forever.

  For a Java sample that schedules a stored procedure that has no defined end for its validity time window, refer to A.9.1, “Use case - 1” on page 858.

– ***The task cannot be executed after a later point in time***

The end of the validity time window is determined by the value of the END_TIMESTAMP parameter. A stored procedure will not be executed after the timestamp defined in this scheduling parameter.

For a Java sample that schedules a stored procedure that is not executed after a certain point in time, refer to A.9.2, “Use case - 2” on page 861.

– ***The task is only valid within a certain time window***

Employ the following two time window scheduling parameters at the same time, to define the validity window of a task:

```
BEGIN_TIMESTAMP
END_TIMESTAMP
```

Consider, if BEGIN_TIMESTAMP and END_TIMESTAMP are not NULL, then both timestamps have to be in the future and END_TIMESTAMP has to be later than BEGIN_TIMESTAMP. See A.9.3, “Use case - 3” on page 863.

– ***The task should be executed not more than a certain number of times***

To limit the number of executions of, for example, a recurring or triggered stored procedure, the MAX_INVOCATIONS scheduling parameter can be used. It defines the maximum number of attempts the scheduler can execute a stored procedure. If both MAX_INVOCATIONS and END_TIMESTAMP are specified and the value in MAX_INVOCATIONS reached the defined limit, but END_TIMESTAMP is not yet exceeded, the task is no longer executed although it is still within the validity time window. If the value in END_TIMSTAMP is exceeded, but the number of MAX_INVOCATION is not yet reached, the stored procedure validity is nevertheless expired and it is no longer executed as well.

A.9.1, “Use case - 1” on page 858 shows a Java sample that implements the scheduling of a stored procedure with MAX_INVOCATIONS = 1.

– ***The task can be executed as many times as scheduled***

Setting the MAX_INVOCATIONS scheduling parameter to NULL implies that the number of invocations of a stored procedure is not constraint. Within the provided validity time interval, a stored procedure can be executed without limitations to the number of times it is invoked.

Under which security context has the stored procedure to be executed?

The first action that is taken by the scheduler when starting a stored procedure execution is to switch its security context to a defined authorization id. The switch to a certain execution user is controlled by the following two parameters:

```
USERID
PASSWORD
```

After the execution completes, the scheduler logs out and returns to its default security context.

- ***The stored procedure can be executed by any user***

  There is no special authorization required to execute the task. When scheduling the stored procedure, the USERID and PASSWORD parameters can be set to NULL. In this case the stored procedure is executed under the default execution user configured for the scheduler.

- ***The stored procedure has to be executed under a distinct user ID***

  To switch to a certain security context when executing a stored procedure, provide the required credentials in the USERID and PASSWORD scheduling parameters. The provided password is not stored. The credentials are only validated once, when the task is

scheduled. Employing parameter markers when calling ADMIN_TASK_ADD ensures that the password does not appear in the dynamic statement cache.

It is not secure to provide the password in clear text

Instead of transmitting a password in clear text to the DB2 server when scheduling a stored procedure, a user or an application can provide an encrypted one-time password. The recommended way is to provide a passticket in the PASSWORD parameter when calling ADMIN_TASK_ADD.

Where will the stored procedure be executed?

In a data sharing scenario, it generally cannot be predicted on which DB2 member a scheduled stored procedure will eventually be executed. Nevertheless it is possible to bind the execution of a scheduled stored procedure to a certain DB2 member. This member affinity is implemented by the scheduling parameter DB2_SSID.

- ***No data sharing group, only a standalone DB2 is available***

  The DB2_SSID scheduling parameter can be set to NULL here.

- ***A scheduled stored procedure can be executed on any DB2 member of a data sharing group***

  Set the DB2_SSID scheduling parameter to NULL, to allow all members in a data sharing environment to execute the stored procedure.

- ***A scheduled stored procedure has to be executed on a specific DB2 member of a data sharing group***

  By setting the DB2_SSID scheduling parameter to a distinct DB2 subsystem identifier, for example:

  ```
  DB2_SSID = "V91A"
  ```

  it is ensured that a scheduled stored procedure will only be executed by this DB2 subsystem.

- ***A stored procedure invoked by a DB2-start event can be triggered by any DB2 that starts in a data sharing group***

  To implement the triggering DB2-start event, initialize the TRIGGER_TASK_NAME scheduling parameter with DB2START. By setting the DB2_SSID parameter to NULL, the scheduled stored procedure execution will be triggered by every DB2 start in the data sharing group. It will be executed on the started DB2 member by its associated scheduler.

- ***A stored procedure invoked by a DB2-start event should only be triggered by a certain DB2 that starts up***

  To implement the triggering DB2-start event, initialize the TRIGGER_TASK_NAME scheduling parameter with DB2START. When setting the DB2_SSID scheduling parameter to a distinct DB2 subsystem identifier, for example V91A, it is ensured that the scheduled stored procedure execution is only triggered by a DB2 start event of the V91A subsystem. It will be executed on the DB2_SSID member by its associated scheduler.

24.3.4 Interpreting the last execution status

The scheduler makes the last execution status of every scheduled stored procedure accessible through the UDF table function DSNADM.ADMIN_TASK_STATUS. The status of the task execution reflects whether the task is currently executed, was executed, or could not be executed. Before a task is first scheduled, all columns that show the respective execution status contain NULL except for TASK_NAME. After the first execution, at least the START_TIMESTAMP, DB2_SSID, USERID, and STATUS columns contain a non-null value. These columns provide information about when, on which DB2 member, and under which

user ID the stored procedure status has been updated for the last time. Additionally the NUM_INVOCATIONS column provides information about the number of times this stored procedure was already executed, including the current execution. This counter is independent of a successful or unsuccessful execution of a stored procedure. The remainder of the execution status can be interpreted according to the different values of the STATUS column:

- **NOTRUN**: The scheduler was not able to start executing the stored procedure. The START_TIMESTAMP and the END_TIMESTAMP values are the same, and a MSG value states why the stored procedure execution could not be started. The DB2 execution status columns (SQLCODE, SQLSTATE, SQLERRMC, and SQLERRP) may contain additional debug information in case the reason not to execute the task is related to DB2 (for example DB2 connection could not be established).
- **RUNNING**: The scheduler started executing the stored procedure, which has not completed yet. All other execution status columns are NULL.
- **COMPLETED**: The last task execution completed. The START_TIMESTAMP column indicates when the task execution started, whereas the END_TIMESTAMP column indicates when the execution completed and the MSG column potentially contains an error or an informational message. The DB2 execution status columns (SQLCODE, SQLSTATE, SQLERRMC, and SQLERRP) might be filled with additional DB2 debugging information.
- **UNKNOWN**: The task execution status is unknown. This occurs when the scheduler is stopped while a stored procedure is executed.

The columns JOB_ID, MAXRC, COMPLETION_TYPE, SYSTEM_ABENDCD, and USER_ABEND_CD only refer to JCL jobs and can be neglected for stored procedure executions.

Example 24-14 on page 574 shows a partial ADMIN_TASK_STATUS output.

For more detailed information about the columns returned, refer to Chapter 3 of *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854.

24.3.5 Syntax diagrams

SYSPROC.ADMIN_TASK_ADD

This DB2 stored procedure adds a task to the scheduler task list. The new task is referenced by its name given by the user in the INOUT parameter TASK_NAME, otherwise an automatic name, *TASK_ID_nnnn (where nnnn is a 4 digit number),* will be given to this task by the task scheduler.

Load module name: DSNADMTA

Package name: DSNADMTA or DSNADMTU

Figure 24-25 illustrates the ADMIN_TASK_ADD CALL syntax.

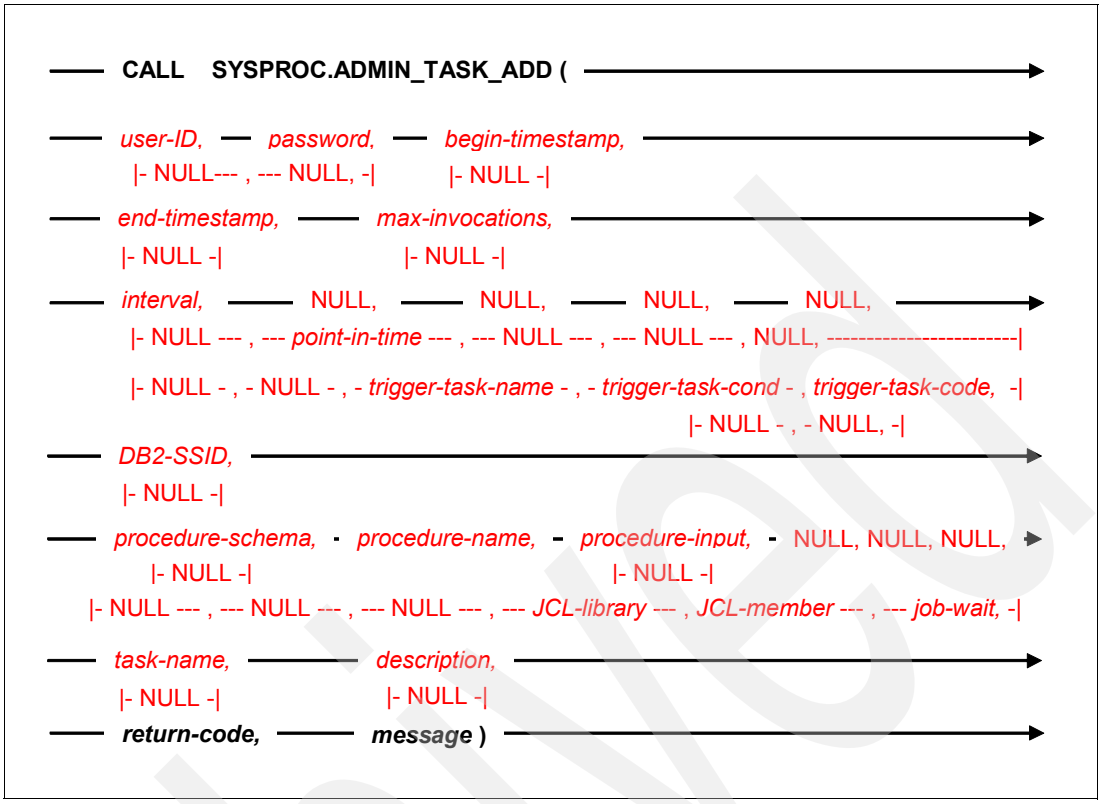

Figure 24-25 CALL ADMIN_TASK_ADD stored procedure

ADMIN_TASK_ADD option descriptions

- ***user-ID:*** Specifies the user ID under which the task execution is performed.

  If this parameter is set to NULL, task execution is performed with the default authorization ID associated with the administrative scheduler instead.

  This is an input parameter of type VARCHAR(128).

- ***password:*** Specifies the password associated with the input parameter *user-ID*.

  The value of *password* is passed to the stored procedure as part of payload, and is not encrypted. It is not stored in dynamic cache when parameter markers are used.

  Recommendation: Have the application that invokes this stored procedure pass an encrypted single-use password called a passticket.

  This parameter is NULL only when *user-ID* is set to NULL, and must be NULL when *user-ID* is NULL.

  This is an input parameter of type VARCHAR(24).

- ***begin-timestamp:*** Specifies when a task can first begin execution. The actual beginning of a task execution depends on how this and other parameters are set:

  Non-null value for *begin-timestamp*

  At *begin-timestamp* - The task execution begins at *begin-timestamp* if *point-in-time* and *trigger-task-name* are NULL.

  Next point in time defined at or after *begin-timestamp* - The task execution begins at the next point in time defined at or after *begin-timestamp* if *point-in-time* is non-null.

When *trigger-task-name* completes at or after *begin-timestamp* - The task execution begins the next time that *trigger-task-name* completes at or after *begin-timestamp*.

Null value for *begin-timestamp*

Immediately - The task execution begins immediately if *point-in-time* and *trigger-task-name* are NULL.

Next point in time defined -The task execution begins at the next point in time defined if *point-in-time* is non-null.

When trigger-task-name completes -The task execution begins the next time that *trigger-task-name* completes.

The value of this parameter cannot be in the past, and it cannot be later than *end-timestamp*.

For further details, refer to ““When does the task have to be executed?” on page 578”.

This is an input parameter of type TIMESTAMP.

- ***end-timestamp:*** Specifies when a task can last begin execution. If this parameter is set to NULL, then the task can continue to execute as scheduled indefinitely.

  The value of this parameter cannot be in the past, and it cannot be earlier than *begin-timestamp.*

  This is an input parameter of type TIMESTAMP.

- ***max-invocations:*** Specifies the maximum number of executions allowed for a task. This value applies to all schedules: triggered by events, recurring by time interval, and recurring by points in time. If this parameter is set to NULL, then there is no limit to the number of times this task can execute.

  For tasks that execute only one time, *max-invocations* must be set to 1 and *interval*, *point-in-time* and *trigger-task-name* must be NULL.

  If both *end-timestamp* and *max-invocations* are specified, the first limit reached takes precedence. That is, if *end-timestamp* is reached, even though the number of task executions so far has not reached *max-invocations*, the task will not be executed again. If *max-invocations* have occurred, the task will not be executed again even if *end-timestamp* is not reached.

  This is an input parameter of type INTEGER.

- ***interval:*** Defines a time duration between two executions of a repetitive regular task. The first execution occurs at *begin_timestamp*. If this parameter is set to NULL, the task is not regularly executed. If this parameter contains a non-null value, the parameters *point-in-time* and *trigger-task-name* must be set to NULL.

  This is an input parameter of type INTEGER.

- ***point-in-time:*** Defines one or more points in time when a task is executed. If this parameter is set to NULL, the task is not scheduled at fixed points in time. If this parameter contains a non-NULL value, the parameters *interval* and *trigger-task-name* have to be set to NULL.

  The *point-in-time* string has the UNIX cron format. The format contains the following pieces of information separated by blanks: given hour, given minute or minutes, given hour or hours, given day or days of the month, given month or months of the year, and given day or days of the week. For each part, you can specify one or several values, ranges, and so forth. For more detailed information, refer to “How to specify one or several non regular points in time” on page 579.

  This is an input parameter of type VARCHAR(400).

- ***trigger-task-name:*** Specifies the task name of the task which, when its execution is complete, will trigger the execution of this task.

  Task names of *DB2START* and *DB2STOP* are reserved for DB2 stop and start events respectively. Those events are handled by the scheduler associated with the DB2 subsystem that is starting or stopping.

  If this parameter is set to NULL, the execution of this task will not be triggered by another task. If this parameter contains a non-null value, the parameters *interval* and *point-in-time* must be set to NULL.

  This is an input parameter of type VARCHAR(128).

- ***trigger-task-cond:*** Specifies the type of comparison to be made to the return code after the execution of task *trigger-task-name*. Possible values are:

  GT - Greater than

  GE - Greater than or equal to

  EQ - Equal to

  LT - Less than

  LE - Less than or equal to

  NE - Not equal to

  If this parameter is set to NULL, the task execution is triggered without considering the return code of executing task *trigger-task-name*. This parameter must be set to NULL if *trigger-task-name* is set to NULL or is either *DB2START* or *DB2STOP*.

  This is an input parameter of type CHAR(2).

- ***trigger-task-code:*** Specifies the return code from executing *trigger-task-name*.

  If the execution of this task is triggered by a stored procedure, *trigger-task-code* contains the SQLCODE that must be returned by the triggering stored procedure in order for this task to execute.

  If the execution of this task is triggered by a JCL job, *trigger-task-code* contains the MAXRC that must be returned by the triggering job in order for this task to execute.

  To find out what is the MAXRC or SQLCODE of a task after execution, invoke the user defined function DSNADM.ADMIN_TASK_STATUS. ADMIN_TASK_STATUS returns these information in the columns MAXRC and SQLCODE.

  The following restrictions apply to the value of trigger-task-code:

  – If trigger-task-cond is null, then trigger-task-code must also be null.

  – If trigger-task-cond is non-null, then trigger-task-code must also be non-null.

  If *trigger-task-cond* and *trigger-task-code* are not null, they are used to test the return code from executing *trigger-task-name* to determine whether to execute this task or not.

  For example, if trigger-task-cond is set to “GE” and *trigger-task-code* is set to “8”, then this task will execute if and only if the previous execution of *trigger-task-name* returned a MAXRC (for a JCL job) or an SQLCODE (for a stored procedure) greater than or equal to 8.

  This is an input parameter of type INTEGER.

- ***DB2-SSID:*** Specifies the DB2 subsystem ID whose associated scheduler should execute the task.

  This parameter is used in a data sharing environment where, for example different DB2 members have different configurations and executing the task relies on a certain environment. However, specifying a value in *DB2*-SSID will prevent schedulers of other

members to execute the task, so that the task can only be executed as long as the scheduler of *DB2-SSID* is running.

For a task being triggered by a DB2 start or DB2 stop event in *trigger-task-name*, specifying a value in *DB2-SSID* will let the task be executed only when the named subsystem is starting and stopping. If no value is given, each member that starts or stops will trigger a local execution of the task, provided that the executions are serialized.

If this parameter is set to NULL, any of the scheduler will be allowed to execute the task.

This is an input parameter of type VARCHAR(4).

- ***procedure-schema:*** Specifies the schema of the DB2 stored procedure this task will execute. If this parameter is set to NULL, DB2 uses a default schema. This parameter must be set to NULL if *procedure-name* is set to NULL.

  This is an input parameter of type VARCHAR(128).

- ***procedure-name:*** Specifies the name of the DB2 stored procedure this task will execute. If this parameter is set to NULL, no stored procedure will be called. In this case, a JCL job must be specified.

  This is an input parameter of type VARCHAR(128).

- ***procedure-input:*** Specifies the input parameters of the DB2 stored procedure this task will execute. This parameter must contain a DB2 SELECT statement that returns one row of data. The returned value will be passed as parameter to the stored procedure.

  If this parameter is set to NULL, no parameter are passed to the stored procedure. This parameter has to be set to NULL when *procedure-name* is set to NULL.

  This is an input parameter of type VARCHAR(4096).

- ***JCL-library:*** Specifies the name of the data set where the JCL job to be executed is saved.

  If this parameter is set to NULL, no JCL job will be executed. In this case, a stored procedure must be specified.

  This is an input parameter of type VARCHAR(44).

- ***JCL-member:*** Specifies the name of the library member where JCL job to be executed is saved.

  If this parameter is set to NULL, the data set specified in *JCL-library* must be sequential and contain the JCL job to be executed. This parameter must be set to NULL if *JCL-library* is set to NULL.

  This is an input parameter of type VARCHAR(8).

- ***job-wait:*** Specifies whether scheduler executes the job synchronously or asynchronously. This parameter can only be set to NULL if *JCL-library* is set to NULL. Otherwise, it must be one of the possible values:

  NO - Asynchronous execution (scheduler does not wait for job to finish execution).

  YES - Synchronous execution (scheduler waits for job to finish execution).

  PURGE - Synchronous execution after which the job status in z/OS is purged

  This is an input parameter of type VARCHAR(8).

- ***task-name:*** Specifies a unique name assigned to the task.

  A unique task name is returned when the task is created with a NULL *task-name* value. This name is of the format TASK_ID_xxxx, where xxxx is 0001 for the first task named, 0002 for the second task, and so forth.

The following task names are reserved and cannot be given as the value of task-name:

- Names starting with TASK_ID_
- DB2START
- DB2STOP

This in an input/output parameter of type VARCHAR(128).

- ***description:*** Specifies a description assigned to the task.

  This is an input parameter of type VARCHAR(128).

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The task was added successfully.

  12 - The call did not complete successfully and the task has not been added. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. The first messages in this area, if any, are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

ADMIN_TASK_ADD output

This stored procedure returns the following output parameters:

- *task-name*
- *return-code*
- *message*

SYSPROC.ADMIN_TASK_REMOVE

The SYSPROC.ADMIN_TASK_REMOVE stored procedure removes a task from the task list of the scheduler.

If the task is currently running, it continues to execute until completion, and the task is not removed from the scheduler task list. If other tasks depend on the execution of the task to be removed, this task is not removed from the administrative scheduler task list.

Users with SYSOPR, SYSCTRL, or SYSADM authority can remove any task. Other users who have EXECUTE authority on this stored procedure can remove tasks that they added. Attempting to remove a task that was added by a different user returns an error in the output.

Load module name: DSNADMTR

Package name: DSNADMTR or DSNADMTU

Figure 24-26 illustrates the ADMIN_TASK_REMOVE CALL syntax.

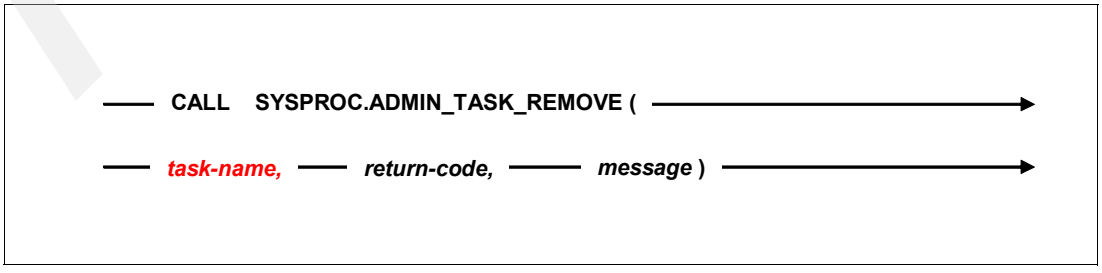

Figure 24-26 CALL ADMIN_TASK_REMOVE stored procedure

ADMIN_TASK_REMOVE option descriptions

- ***task-name:*** Specifies the task name of the task to be removed from the scheduler task list. The task name cannot be NULL.

  This is an input parameter of type VARCHAR(128).

- ***return-code:*** Provides the return code from the stored procedure. Possible values are:

  0 - The call completed successfully and the task was successfully removed.

  12 - The call did not complete successfully and the task has not been removed. The *message* output parameter contains messages describing the error.

  This is an output parameter of type INTEGER.

- ***message:*** Contains messages describing the error encountered by the stored procedure. The first messages in this area, if any, are generated by the stored procedure. Messages that are generated by DB2 might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

ADMIN_TASK_REMOVE output

This stored procedure returns the following output parameters:

- *return-code*
- *message*

24.4 Common SQL API - Administration functions common to all IBM data servers

There are many ways to obtain administrative information from IBM data servers such as DB2 for z/OS V8 and V9, LUW and Informix® IDS. This includes both remote access through an administrative routine as well as remote access through command line interface commands. The variety of access methods to administrative functions, their different syntax, and their security options have permeated tools, resulting in tight coupling between tool and data server versions, high implementation complexity on the tools side, and slow integration or reuse.

The common SQL API described here attempts to solve these problems for the common application development and administration tooling solution space, benefiting tools such as the Data Studio Administration Console (name of the tool is not finalized yet, still subject to rebranding) and future tooling efforts across all data servers, such as performance tools.

24.4.1 A brief functional overview

The common SQL API is a collection of common-signature and signature-stable stored procedures across all IBM data servers that support tooling usage scenarios. By providing a single access method to administrative functions and using the same security options, the common SQL API provides loose coupling between tool and data server platform and version, reduces the complexity of tool implementation, and makes integration of tools much easier. On DB2 for z/OS V8 and V9, the common SQL API currently consists of three stored procedures that basically get configuration data as well as data for resource exploitation. The provided stored procedures and the respective scope of data collection are:

SYSPROC.GET_MESSAGE - Stored procedure to retrieve the short message text of a provided SQLCODE.

SYSPROC.GET_SYSTEM_INFO - Stored procedure to retrieve system information such as:

- Operating system information
  a. Name and release
  b. CPU model, number of online processors, processor identifier, and the serial number of the online processors
  c. Real storage size
- Product information
  a. Primary JES name, release, node name, and held output class
  b. Security software name and FMID
  c. DFSMS™ release
  d. TSO release
  e. VTAM release
- DB2 MEPL
- Apply status of a SYSMOD
- WLM classification rules that apply to DB2-related workload for subsystem types DB2 and DDF

SYSPROC.GET_CONFIG - Stored procedure to get the data server configuration information. If GET_CONFIG is called in a data sharing environment, only information that is accessible on the DB2 subsystem that executes the stored procedure is gathered. The information collected is the following:

- Data sharing group information

  In a data sharing environment, items (b) and (c) are subsystem-specific and are only collected for the server where the stored procedure is executed.

  – Group name, level, mode, protocol level, attach name
  – DB2 member, subsystem ID, command prefix, status, level
  – z/OS system name, IRLM subsystem, IRLM procedure
  – SCA structure size, status, utilization,
  – LOCK1 structure size, number of entries
  – Number of list entries, utilization
- DB2 subsystem parameters

  The subsystem parameters are only collected for the DB2 subsystem that executes the GET_CONFIG stored procedure, even in a data sharing environment.

  – Control block name
  – Install panel name
  – Install panel field name
  – Location on install panel
  – Subsystem parameter name
  – Subsystem parameter setting
- Data distribution facility information

  Information listed in this section is only retrieved for the DB2 subsystem that executes the GET_CONFIG stored procedure, even in a data sharing environment.

  – DDF status
  – Location name, Lu-name, Generic Lu-name
  – IP-address, TCP/IP Port, Resynchronization port
  – SQL domain, re-synchronization port

  - DT, CONDBAT, MDBAT, ADBAT, QUEDBAT, INADBAT, CONQUED, DSCDBAT, INACONN
- Connected DB2 subsystem
  - Subsystem ID of the DB2 that currently executes GET_CONFIG.
- Resource limit facility information
  - Table name of the active resource limit specification
- Active log data set information
  - Active log data set names, volume names
- Timestamp of last DB2 restart

All three common SQL API stored procedures feature the same signature. This signature stability and commonality are achieved by using simple XML documents as input and output parameters. The data server information that is collected by the respective stored procedure is thus wrapped into an XML document and returned as a BLOB to the caller. All XML documents, either input or output, are described by a common XML Document Type Definition (DTD). Version, platform and technology differences are expressed through different key or value pairs in these so-called hierarchical XML property lists.

A CALL to the common signature is depicted in Figure 24-27.

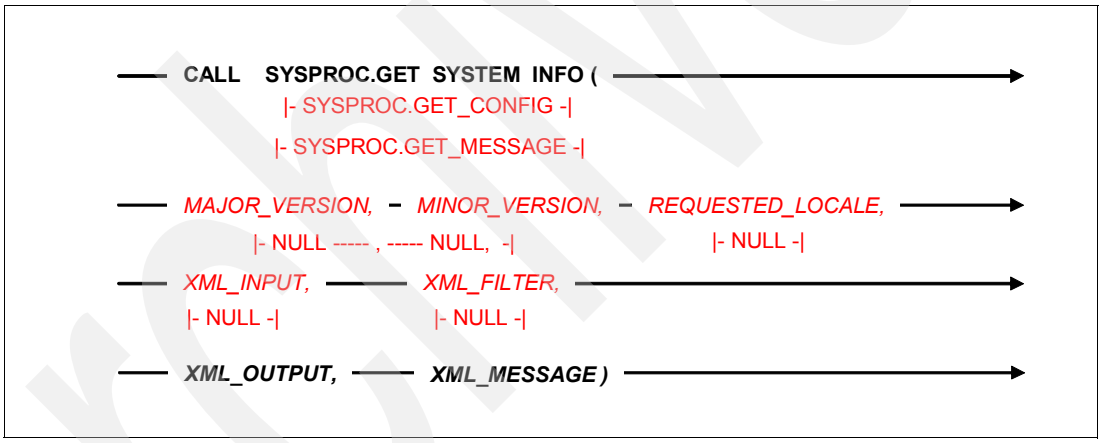

Figure 24-27 Common SQL API signature

The description of the respective procedure parameters is in Table 24-39.

Table 24-39 Common SQL API parameters

Parameter	Description
MAJOR_VERSION *INTEGER,* *INOUT*	All XML documents (XML_INPUT, XML_OUTPUT, XML_MESSAGE) are versioned after a major and a minor version number. The stored procedure will return the output XML documents in the major version indicated in this parameter. If the provided version is not supported an error is returned. The lowest major version supported starts at 1. The highest supported major version for all the XML documents is always returned in the output parameter. This parameter has to be used in conjunction with MINOR_VERSION

Parameter	Description
MINOR_VERSION *INTEGER,* *INOUT*	Similarly as with major version, the stored procedure will return the output XML documents in the minor version indicated in this parameter. If the provided version is not supported an error is returned. The lowest minor version supported starts at 0. The highest supported minor version for all the XML documents is always returned in the output parameter. This parameter has to be used in conjunction with MAJOR_VERSION.
REQUESTED_LOCALE *VARCHAR(33),* *CCSID EBCDIC,* *IN*	By using this parameter the caller can request localized content to be returned in the XML_OUTPUT and XML_MESSAGE parameters. If the language indicated is not supported on the server-side, the output documents are returned in a default language.
XML_INPUT *BLOB(2G) AS LOCATOR,* *IN*	Some stored procedures are capable of using an XML input document to receive special runtime parameters. If an input document is supported, the caller passes it encoded in UTF-8 in this parameter to the stored procedure. Alternatively, the provided input document can be used in Complete mode. In this case the stored procedure returns a valid template input document that can be used in a successive call to provide runtime parameters.
XML_FILTER *BLOB (4K),* *IN*	The stored procedure will always return the complete XML output document unless an XML_FILTER is provided by the caller. It is specified as a valid but restricted XPath query string in UTF-8, that qualifies and returns a single element in the XML tree.
XML_OUTPUT *BLOB(2G) AS LOCATOR,* *IN*	If the call to the stored procedure could successfully be executed, this parameter returns the complete XML output document for the respective stored procedure. Alternatively if an XML filter has been applied only the single qualifying string is returned. The content of this parameter is encoded in UTF-8.
XML_MESSAGE *BLOB(64K),* *OUT*	If the call to the stored procedure results in an SQL warning being raised, this parameter may contain a complete XML message document. This provides detailed information on the warning condition encountered as well as additional debug information to resolve the problem. The content of this parameter is encoded in UTF-8.

24.4.2 Working with the Common SQL API

The signature described above is very flexible and powerful once the parameters are well understood. In this section we provide an overview about how to work efficiently with the common signature and we give an introduction to the XML format that is used to serialize the information retrieved. A sample is provided that shows how the GET_MESSAGE stored procedure can be called from a Java program with special emphasis on proper parameter initialization and encoding.

XML Property List format

All XML parameter documents adhere to a single, common parameter list DTD. This DTD is flexible enough to represent hierarchical structures and binary data. The concept of parameter lists originated in Mac OS X and has been adopted as a serialization format in many other applications. For Property List Programming Topics for Core Foundation, see

http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html

The associated DTD can be found at:

http://www.apple.com/DTDs/PropertyList-1.0.dtd

A single piece of information is described by entries in the XML document, which are usually grouped using nested dictionaries (dict). These entries are basically a set of key or value pairs. A value associated with a key is further structured to include the actual *Value'* a (translated) *Display Name*, and a (translated) *Hint* key. Optionally, a (translated) *Display Unit* key may be provided to correctly interpret the value. A sample entry that contains information about the *Real Storage Size* is shown in Example 24-16.

Example 24-16 Typical grouping of key or value pairs

```
<key>Real Storage Size</key>
<dict>
   <key>Display Name</key><string>Real Storage Size</string>
   <key>Value</key><integer>2048</integer>
   <key>Display Unit</key><string>MB</string>
   <key>Hint</key><string></string>
</dict>
```

The entry *Display Unit* will be omitted if no unit of measure is applicable to the value being described. The details and content for the specific entries depend on each stored procedure. The type for the value may be <integer>, <string>, <date>, <real>, or an <array>. If it is an <array>, all elements in the array must be <integer>, <string>, <date>, or <real>. For a Complete-Mode document the keys <true> and <false> are supported.

XML documents used by the common SQL API

There are three different types of XML documents in the common SQL API. These are

XML_INPUT
XML_OUTPUT
XML_MESSAGE

Whereas XML_INPUT and XML_OUTPUT documents differ for each of the Common SQL API stored procedures, the XML_MESSAGE structure is shared among them all. In the following section we provide an overview of the basic structure of the three XML documents.

XML_INPUT

The XML_INPUT document is used to provide special runtime information to a stored procedure, which basically influences the amount and the type of the returned XML output document content. The major structure of the document consists of a set of key or value pairs that are common to all stored procedures that support input documents, and two sections containing Required Parameters and Optional Parameters that are stored procedure specific. The general structure of an XML input document is listed in Example 24-17. Note that the key or value pairs in *italic* are used by all stored procedures, whereas the pairs in bold are procedure-specific.

Example 24-17 XML_INPUT excerpt

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
   <dict>
      <key>Document Type Name</key>
      <string>Data Server Message Input</string>
```

```
      <key>Document Type Major Version</key><integer>1</integer>
      <key>Document Type Minor Version</key><integer>0</integer>
      <key>Document Locale</key><string>en_US</string>
      <key>Complete</key><false/>
      <key>Required Parameters</key>
      <dict>
      ...
      </dict>
      <key>Optional Parameters</key>
      <dict>
      ...
      </dict>
   </dict>
</dict>
</plist>
```

The value of the `Document Type Name` key determines whether the XML document is an input or an output document, as well as the stored procedure this XML document is associated with. In Example 24-17 the string value *Data Server Message Input* indicates an input document for the GET_MESSAGE stored procedure.

Complete Mode vs. non-Complete Mode

Every stored procedure that supports an input document can be called in either Complete Mode or non-Complete Mode. The Complete entry in the common section of an XML_INPUT document determines this behavior. In the example above, the procedure is called in a non-Complete Mode, because it employs the following entry:

```
<key>Complete</key><false/>
```

Note, the Complete entry is optional and can be omitted if the value is false. In this case the only required entries are:

```
Document Type Name, Required Parameters, Optional Parameters
```

Non-Complete Mode implies a regular call to the stored procedure that returns an XML output document. For example, calling GET_MESSAGE with a non-Complete Mode input document will return an output document containing the respective short message text.

If the value for the Complete key is true, for example `<key>Complete</key><true/>`, the called stored procedure will operate in the Complete Mode.

This means that a caller wants the stored procedure to only return a template XML input document that looks similar to the one in Example 24-17. This template XML input document is returned in the XML_OUTPUT parameter and can then be populated with the Required Parameters and Optional Parameters. In a second successive call to the stored procedure this augmented, now non-Complete Mode input document can be used to retrieve the desired information from the data server contained in the XML output document.

Example 24-18 is a minimal XML input document in Complete Mode. Note that there is no stored procedure-specific information in there, so that it can be used for any Common SQL API stored procedure that supports an input document. Should the XML input document for Complete Mode contain XML elements in addition to the ones shown, those other entries are ignored and will not be retained in the document returned.

Example 24-18 Complete Mode input document

```
<?xml version="1.0"  encoding="UTF-8"?>
<plist version="1.0">
```

```
<dict>
   <key>Complete</key><true/>
</dict>
</plist>
```

As mentioned, when calling a procedure in Complete Mode the returned XML document is a full XML input document in non-Complete Mode, including a Document Type and sections for all possible required and optional parameters. There are additional Hint sections included, that might provide additional information on how to interpret distinct sections in the XML document.

Figure 24-28 depicts the typical work flow when employing the Common SQL API stored procedures that support an input document. When the Complete Mode approach is employed, two calls to the stored procedure are required. The interaction would start with the upper CALL statement in the figure. Note that the XML_OUTPUT content is reused in the second CALL as XML_INTPUT content.

Alternatively, if a valid non-Complete Mode XML input document is already available, one CALL is sufficient. A user or application program would start with the lower CALL statement illustrated in Figure 24-28.

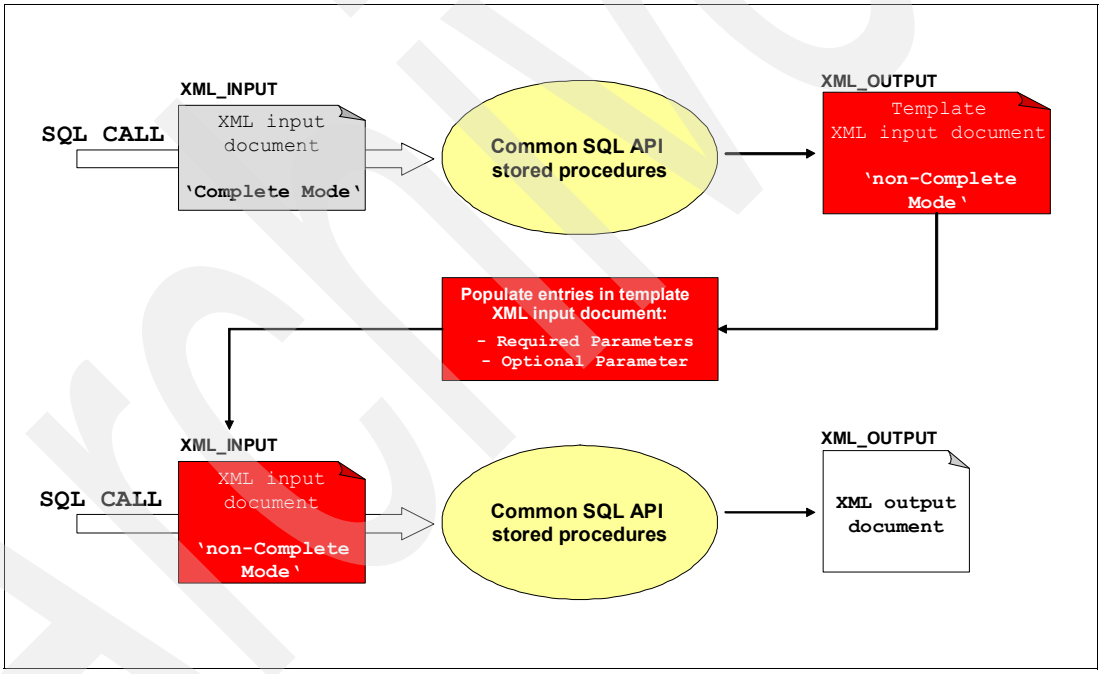

Figure 24-28 Complete Mode vs. non-Complete Mode work flow

XML_OUTPUT

If the Common SQL API stored procedure has been invoked in Complete Mode, the XML_OUTPUT parameter returns a template XML input document. This document can be used in the XML_INPUT parameter for the same stored procedure in a second call in non-Complete Mode as illustrated in Figure 24-28. In this case the XML_OUTPUT parameter returns the stored procedure-dependent data server information, wrapped into an XML document. This output document contains a header section that implements the same keys for all Common SQL API stored procedures. These common keys are highlighted in *italic* in Example 24-19.

Example 24-19 XML_OUTPUT excerpt

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
   <key>Document Type Name</key>
   <string>Data Server Message Output</string>
   <key>Document Type Major Version</key><integer>1</integer>
   <key>Document Type Minor Version</key><integer>0</integer>
   <key>Data Server Product Name</key><string>DSN</string>
   <key>Data Server Product Version</key><string>8.1.5</string>
   <key>Data Server Major Version</key><integer>8</integer>
   <key>Data Server Minor Version</key><integer>1</integer>
   <key>Data Server Platform</key><string>z/OS</string>
   <key>Document Locale</key><string>en_US</string>
      Document type specific data here
</dict>
</plist>
```

Once more the `Document Type Name` key refers to the stored procedure that is associated with the output document as well as to the type of the document. The above example is thus extracted from an XML output document obtained through a GET_MESSAGE stored procedure call. The other key or value pairs in the common header of the output document describe version and the environment information for the stored procedure call. The stored procedure-specific output information is placed underneath the common header section, indicated in the example by the placeholder:

```
Document type specific data here
```

XML_MESSAGE

When a stored procedure encounters an internal processing error, or an invalid input parameter, a warning with the respective SQLCODE is created. In this case, all output parameters are set to NULL, except for the MAJOR_VERSION, MINIOR_VERSION, and the XML_MESSAGE output parameter. XML_MESSAGE will return an XML message document containing more detailed error and debug information for the situation.

The XML message document structure is common across all Common SQL API stored procedures and features always the same value for the `Document Type Name` key: *Data Server Message.* Example 24-20 illustrates an XML message document.

Example 24-20 XML_MESSAGE sample

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
   <key>Document Type Name</key>
   <string>Data Server Message</string>
   <key>Document Type Major Version</key><integer>1</integer>
   <key>Document Type Minor Version</key><integer>0</integer>
   <key>Data Server Product Name</key><string>DSN</string>
   <key>Data Server Product Version</key><string>8.1.5</string>
   <key>Data Server Major Version</key><integer>8</integer>
   <key>Data Server Minor Version</key><integer>1</integer>
   <key>Data Server Platform</key><string>z/OS</string>
   <key>Document Locale</key><string>en_US</string>
   <key>Short Message Text</key>
   <dict>
```

```
      <key>Display Name</key><string>Short Message Text</string>
      <key>Value</key>
      <string>...additional description of warning...</string>
      <key>Hint</key><string></string>
   </dict>
</dict>
</plist>
```

Here the string:*"...additional description of warning..."* will contain a message ID and the corresponding short message text in case of a real error.

For example, when GET_MESSAGE is called with improperly initialized parameters, the following XML message document is returned with the entry `Short Message Text` that contains debug information, as shown in Example 24-21.

Example 24-21 Short Message Text

```
<key>Short Message Text</key>
<dict>
   <key>Display Name</key><string>Short Message Text</string>
   <key>Value</key>
   <string>DSNA630I  DSNADMGM A PARAMETER FORMAT OR
      CONTENT ERROR WAS FOUND. If parameters 1
      and 2 are set to NULL, all other input parameters
      must be set to NULL as well, but the value of
      parameter 5 is not NULL.</string>
   <key>Hint</key><string></string>
</dict>
```

Error handling

The three common SQL API stored procedures implement a common set of SQLCODEs and SQLSTATEs that either indicate an error or a warning. The general idea of the stored procedures is to refrain from returning error SQLCODEs, but rather return warnings for most problem scenarios, if possible. The advantage of returning a positive SQLCODE (warning) is basically that the output parameters are still accessible. This allows the utilization of the XML message output document for efficient problem determination and resolution. For some warning scenarios, the stored procedures are even able to construct and retrieve a complete XLM output document. However, as a general rule the XML_OUTPUT and XML_MESSAGE parameters are mutually exclusive, that is, either the XML output or the XML message document is returned in a warning scenario. The following table lists the common SQLCODE and SQLSTATES that are implemented by all three stored procedures, as well as which document is returned. The error SQLCODEs are listed in Table 24-40.

Table 24-40 The error SQL codes

Error SQL codes	Document returned in	Scenario
SQLCODE: -20457 SQLSTATE: 38554	No output document returned	The version indicated in MAJOR_VERSION or MINOR_VERSION parameter is not supported.

The warning codes are listed in Table 24-41.

Table 24-41 The warning SQL codes

Warning SQL codes	Document returned in	Scenario
SQLCODE: +20458 SQLSTATE: 01H54	XML_MESSAGE	The procedure encountered a problem in the format or content of a provided input parameter.
SQLCODE: +20460 SQLSTATE: 01H56	XML_OUTPUT	The stored procedure supports a higher parameter version than the one returned in the output document.
SQLCODE: +20459 SQLSTATE: 01H55	XML_MESSAGE	The procedure encountered an internal error situation.
SQLCODE: +20461 SQLSTATE: 01H57	XML_OUTPUT	The output returned is in an alternate locale than the one specified in REQUESTED_LOCALE.

If available, it is always recommended to check the document returned in XML_MESSAGE parameter.

Severity of SQLCODEs and SQLSTATEs

The SQLCODEs and SQLSTATEs are associated with a certain severity. In case multiple error scenarios occur concurrently, the SQLCODE / SQLSTATE combination with the highest severity is always returned. Multiple error states are never returned to the user.

1. -20457 / 38554
   Unsupported parameter version (highest priority).
2. +20458 / 01H54 and +20459 / 01H55
   Parameter format or content error and internal processing error.
3. +20461 / 01H57
   Output document returned in different locale.
4. +20460 / 01H56
   Higher version is supported.

Note: The warnings SQLCODEs and SQLSTATEs +20458/01H54 and +20459/01H55 share the same severity. This is possible since they are mutually exclusive and the scenarios will never occur at the same time.

Versioning

The signature stability for all three Common SQL API stored procedures is a result of the flexibility that is provided by employing XML documents as parameters. All XML documents are versioned with the help of a major and a minor version, starting with an initial version of 1.0. With the introduction of a new version, a new structure or new key or value pairs could be supported. For a single call to any stored procedure the major and minor version of all documents involved are constant. That means it is ensured that the XML input document always has the same version as the generated XML output or message document.

A caller can request a certain major and minor version for the output documents by setting the MAJOR_VERSION and MINOR_VERSION parameters of the stored procedure accordingly. The returned XML output document (either XML_OUTPUT or XML_MESSAGE) contains two key or value pairs in the common header section indicating the actual major and minor version. These pairs are shown in Example 24-22.

Example 24-22 Key or value pairs for the version of the XML_OUTPUT or XML_MESSAGE document

```
<key>Document Type Major Version</key><integer>1</integer>
<key>Document Type Minor Version</key><integer>0</integer>
```

The MAJOR_VERSION and MINOR_VERSION input parameters have to be specified together, that is, both must be NULL or both must be non-NULL. Otherwise, the Common SQL API stored procedure will raise error -20457. In case MAJOR_VERSION and MINOR_VERSION are non-NULL, they have to specify a supported document version, otherwise error -20457 will be returned again. See the section Error Handling for further information.

The XML_INPUT document features optional key or value pairs for `Document Type Major Version` and `Document Type Minor Version` in the common header section, which indicate the version of the input document. To ensure that the input document is of the same version as the output document, the `Document Type Major Version` and `Document Type Minor Version` have to match the MAJOR_VERSION and MINOR_VERSION parameters in the signature. If they are not identical, a +20458 SQLCODE is raised. This avoids scenarios where a higher major or minor version adds new required parameters to the input document that are not yet supported in the lower output document version.

Again, if they are present, `Document Type Major Version` and `Document Type Minor Version` must be specified together in the XML input document, that is, both must be specified with a valid and supported version, or both must not be specified. Otherwise, a +20458 SQLCODE is raised.

When the call did not result in an error SQLCODE, that is, the specified version is supported, the highest supported major and minor version is always returned in the MAJOR_VERSION and MINOR_VERSION output parameters.

Note: To provide further user assistance to determine the latest major and minor versions supported by the currently installed Common SQL API stored procedures, all input parameters, especially MAJOR_VERSION and MINOR_VERSION, can be set to NULL. Such a call completes successfully and returns only the highest supported version numbers in the associated output parameters. In this case no further processing is done by the procedure.

Locale handling

The content of the `Display Name`, `Hint`, and `Display Unit` keys is language-sensitive and will be translated to the language specified in the value for the REQUESTED_LOCALE parameter. If the language is not supported by the current version of the stored procedure, then a default locale is employed, which is *n_US*. Therefore, the caller should always compare the value of the REQUESTED_LOCALE parameter with the actual locale returned in the XML output documents. This is indicated by the key or value pair in the common header section shown in Example 24-23.

Example 24-23 Requested Locale

```
<key>Document Locale</key><string>en_US</string>
```

Note: Version 1.0 of all Common SQL API stored procedures only supports the locale en_US.

SQL access and security

Any user that has the EXECUTE privileges granted to the respective Common SQL API stored procedure is able to successfully CALL it. No other special authorization than the one shown in Example 24-24 is required to, for example, CALL GET_CONFIG.

Example 24-24 Security level

```
GRANT EXECUTE ON PROCEDURE SYSPROC.GET_CONFIG TO xxx
```

GET in touch with the provided stored procedures

So far the description of the Common SQL API was focussed on general concepts and understanding of the flexibility that the common signature offers by implementing XML documents. In this section deeper insight into the distinct stored procedures is provided.

SYSPROC.GET_MESSAGE

The GET_MESSAGE stored procedure returns the short message text that is associated with a certain SQLCODE. The SQLCODE and, if available, some message tokens are passed to the stored procedure while being wrapped into the XML input document. The short message text is returned in the XML output document.

Any XML input document that is associated with the GET_MESSAGE stored procedure must contain the key or value pair in its common header section shown in Example 24-25.

Example 24-25 XLM input

```
<key>Document Type Name</key><string>Data Server Message Input</string>
```

Any GET_MESSAGE XML output document contains the key or value pair of Example 24-26.

Example 24-26 XML output

```
<key>Document Type Name</key><string>Data Server Message Output</string>
```

Since GET_MESSAGE supports an XML input document, it can be called in Complete Mode. A typical workflow including a call in Complete Mode could be comprised of the following steps:

Complete Mode - Invoke GET_MESSAGE with an XML input document in Complete Mode. Set the following parameter values:

```
MAJOR_VERSION = 1
MINOR_VERSION = 0
REQUESTED_LOCALE = ”en_US”
XML_FILTER = NULL
```

Example 24-27 shows a Complete Mode XML input document that could be used for the XML_INPUT parameter.

Example 24-27 Complete Mode’ XML input

```
<?xml version="1.0"  encoding="UTF-8"?>
<plist version="1.0">
<dict>
   <key>Complete</key><true/>
</dict>
</plist>
```

Valid XML input document - After the invocation in Complete Mode, a template XML input document is returned via the XML_OUTPUT parameter. This document can now be augmented with the respective SQLCODE and the message tokens, if applicable. It then can be used in a second call. Also note that the highest supported major and minor versions are returned in the output parameters MAJOR_VERSION and MINOR_VERSION. An augmented XML input document could look like Example 24-28 when used for a second call to GET_MESSAGE.

Example 24-28 GET_MESSAGE XML_INPUT document

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
   <dict>
      <key>Document Type Name</key>
      <string>Data Server Message Input</string>
      <key>Document Type Major Version</key><integer>1</integer>
      <key>Document Type Minor Version</key><integer>0</integer>
      <key>Document Locale</key><string>en_US</string>
      <key>Required Parameters</key>
      <dict>
         <key>Display Name</key>
         <string>Required Parameters</string>
         <key>SQLCODE</key>
         <dict>
            <key>Display Name</key>
            <string>SQLCODE</string>
            <key>Value</key><integer>-805</integer>
            <key>Hint</key><string></string>
         </dict>
         <key>Hint</key><string></string>
      </dict>
      <key>Optional Parameters</key>
      <dict>
         <key>Display Name</key>
         <string>Optional Parameters</string>
         <key>Message Tokens</key>
         <dict>
            <key>Display Name</key>
            <string>Message Tokens</string>
            <key>Value</key>
            <array>
               <string>REDBOOK-consistency-token</string>
               <string>REDBOOK-plan-name</string>
               <string>8</string>
            </array>
            <key>Hint</key>
            <string></string>
      </dict>
      <key>Hint</key><string></string>
   </dict>
</dict>
</plist>
```

Note that the key Complete = false is assumed when it is not provided. As indicated in **bold**, the section containing the SQLCODE has been augmented with a -805, the stored procedure should get the short message text for it. Additionally, some message tokens have been added to the optional parameter section. Invoking GET_MESSAGE with this XML input document returns the XML_OUTPUT document shown in Example 24-29.

Example 24-29 GET_MESSAGE XML_OUTPUT document

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
```

```
   <dict>
   <key>Document Type Name</key>
   <string>Data Server Message Output</string>
   <key>Document Type Major Version</key><integer>1</integer>
   <key>Document Type Minor Version</key><integer>0</integer>
   <key>Data Server Product Name</key><string>DSN</string>
   <key>Data Server Product Version</key><string>8.1.5</string>
   <key>Data Server Major Version</key><integer>8</integer>
   <key>Data Server Minor Version</key><integer>1</integer>
   <key>Data Server Platform</key><string>z/OS</string>
   <key>Document Locale</key><string>en_US</string>
   <key>Short Message Text</key>
   <dict><key>Display Name</key><string>Short Message Text</string>
      <key>Value</key>
      <string>DSNT408I SQLCODE = -805, ERROR: DBRM OR PACKAGE
              NAME REDBOOK-consistency-token NOT FOUND IN PLAN
              REDBOOK-plan-name. REASON 8
      </string>
      <key>Hint</key><string></string>
   </dict>
</dict>
</plist>
```

SYSPROC.GET_SYSTEM_INFO

GET_SYSTEM_INFO collects system-related information and returns it in the XML output document. The stored procedure can be called with an XML input document that allows to specify an SMPCSI data set name and a list of SYSMODs whose status will be queried from the SMPCSI data set in the information gathering process. In order to query the status of a SYSMOD, it is assumed that the software components are administered using SMP/E, and furthermore that the SMPCSI data set is up-to-date. The SMPCSI and SYSMOD keys are contained in the Optional Parameter section, thus GET_SYSTEM_INFO can also be called without XML input parameter document. Without an XML input document there will be no SYSMOD status returned in the XML output document.

Any XML input document that is associated with the GET_SYSTEM_INFO stored procedure contains the key or value pair shown in Example 24-30 in its common header section.

Example 24-30 XML input documents associated with the GET_SYSTEM_INFO

```
<key>Document Type Name</key><string>Data Server System Input</string>
```

Any GET_SYSTEM_INFO XML output document contains the key or value pair shown in Example 24-31.

Example 24-31 XML output

```
<key>Document Type Name</key><string>Data Server System Output</string>
```

Since an XML input document is supported for GET_SYSTEM_INFO, the stored procedure can be called in Complete Mode to retrieve a valid XML input document. The Complete Mode document passed in the XML_INPUT parameter might look like Example 24-32.

Example 24-32 Complete Mode' document passed in the XML_INPUT

```
<?xml version="1.0"  encoding="UTF-8"?>
```

```
<plist version="1.0">
<dict>
   <key>Complete</key><true/>
</dict>
</plist>
```

Calling GET_SYSTEM_INFO in Complete Mode returns a template XML input document in the XML_OUTPUT parameter. The non-Complete Mode XML input document still has to be augmented with the name of the SMPCSI data set and a list of SYSMODs, and looks like Example 24-33.

Example 24-33 GET_SYSTEM_INFO XML_INPUT document

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
   <key>Document Type Name</key>
   <string>Data Server System Input</string>
   <key>Document Type Major Version</key><integer>1</integer>
   <key>Document Type Minor Version</key><integer>0</integer>
   <key>Document Locale</key><string>en_US</string>
   <key>Optional Parameters</key>
   <dict>
      <key>Display Name</key><string>Optional Parameters</string>
      <key>SMPCSI Data Set</key>
      <dict>
         <key>Display Name</key><string>SMPCSI Data Set</string>
         <key>Value</key><string>SMPE.SMPCSI.CSI</string>
         <key>Hint</key><string></string>
      </dict>
      <key>SYSMOD</key>
      <dict>
         <key>Display Name</key><string>SYSMOD</string>
         <key>Value</key>
         <array>
            <string>UK32046</string>
            <string>UK32047</string>
            <string>UK32061</string>
         </array>
         <key>Hint</key><string></string>
      </dict>
      <key>Hint</key><string></string>
   </dict>
</dict>
</plist>
```

Note that the key Complete = false is assumed when it is not provided. To finally obtain the system information and additional SYSMOD information, set the following parameter values so as to employ the augmented XML input document:

```
MAJOR_VERSION = 1
MINOR_VERSION = 0
REQUESTED_LOCALE = “en_US”
XML_FILTER = NULL.
```

Example 24-34 shows a short excerpt of the XML output document.

Example 24-34 GET_SYSTEM_INFO XML_OUTPUT document

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
   <key>Document Type Name</key>
   <string>Data Server System Output</string>
   <key>Document Type Major Version</key><integer>1</integer>
   <key>Document Type Minor Version</key><integer>0</integer>
   <key>Data Server Product Name</key><string>DSN</string>
   <key>Data Server Product Version</key><string>8.1.5</string>
   <key>Data Server Major Version</key><integer>8</integer>
   <key>Data Server Minor Version</key><integer>1</integer>
   <key>Data Server Platform</key><string>z/OS</string>
   <key>Document Locale</key><string>en_US</string>
   <key>Operating System Information</key>
   <dict>
      <key>Display Name</key><string>Operating System Information</string>
      <key>Name and Release</key>
   <dict>
         <key>Display Name</key><string>Name and Release</string>
         <key>Value</key><string>z/OS 01.07.01</string>
         <key>Hint</key><string></string>
      </dict><key>CPU</key>
      <dict>
         <key>Display Name</key><string>CPU</string>
         <key>Model</key>
         <dict>
            <key>Display Name</key><string>Model</string>
            <key>Value</key><string>4381</string>
            <key>Hint</key><string></string>
         </dict>
            ...Further document type specific data here...
      </dict>
   </dict>
</dict>
</plist>
```

SYSPROC.GET_CONFIG

The GET_CONFIG stored procedure returns data server configuration information in the XML_OUTPUT parameter. There is no XML input document supported for GET_CONFIG, thus the stored procedure cannot be called in Complete Mode.

Any XML output document that is associated with the GET_CONFIG stored procedure contains the key or value pair in its common header section, as shown in Example 24-35.

Example 24-35 XML output documents associated with the GET_CONFI

```
<key>Document Type Name</key><string>Data Server Configuration Output</string>
```

When calling GET_CONFIG the XML_INPUT, the parameter has to be nullified. Basically there are three ways to nullify this input parameter: Pass a NULL reference (which implies setting the indicator host variable to -1 in embedded SQL, or a NULL reference in Java), blanks only, or an empty string. To obtain the data server configuration output, set

```
MAJOR_VERSION = 1
MINOR_VERSION = 0
REQUESTED_LOCALE = “en_US”
XML_INPUT = NULL
XML_FILTER = NULL
```

Example 24-36 shows a small excerpt of the XML output document.

Example 24-36 GET_CONFIG XML_OUTPUT document

```
<?xml version="1.0" encoding="UTF-8"?>
<dict>
   <key>Document Type Name</key>
   <string>Data Server Configuration Output</string>
   <key>Document Type Major Version</key><integer>1</integer>
   <key>Document Type Minor Version</key><integer>0</integer>
   <key>Data Server Product Name</key><string>DSN</string>
   <key>Data Server Product Version</key><string>8.1.5</string>
   <key>Data Server Major Version</key><integer>8</integer>
   <key>Data Server Minor Version</key><integer>1</integer>
   <key>Data Server Platform</key><string>z/OS</string>
   <key>Document Locale</key><string>en_US</string>
   <key>Common Data Sharing Group Information</key>
   <dict>
      <key>Display Name</key>
      <string>Common Data Sharing Group Information</string>
      <key>Data Sharing Group Name</key>
      <dict>
         <key>Display Name</key>
         <string>Data Sharing Group Name</string>
         <key>Value</key>
         <string>........</string>
         <key>Hint</key>
         <string></string>
      </dict>
            Further document type specific data here...
   </dict>
</dict>
</plist>
```

Note that the highest supported major and minor versions are additionally returned in the output parameters MAJOR_VERSION and MINOR_VERSION.

Note: APAR PK64298 (PTFs UK37310 and PK37311 respectively for DB2 Version 8 and DB2 9 for z/OS) add options for the DB2-supplied stored procedures SYSPROC.GET_CONFIG, SYSPROC.GET_MESSAGE, and SYSPROC.GET_SYSTEM_INFO.

XML_FILTER

Especially the procedures GET_SYSTEM_INFO and GET_CONFIG return a vast amount of information in the XML output parameter document. In some cases it might not be necessary to get the complete XML output document, but instead only a single value could be required. Here, the complete XML document can be retrieved and then parsed for this specific value; however, this approach involves some application programming overhead. Alternatively, the XML_FILTER parameter can very easily be employed. This input parameter allows to specify

an XPath query string (only restricted axes are supported) that qualifies one single value only. A successful CALL to the procedure then returns the queried single value as a string in the XML_OUTPUT parameter instead of the complete XML output document.

Sample XML_FILTERs

GET_CONFIG - IP Address:

```
"/plist/dict/key[.='DB2 Subsystem Specific Information']/following-sibling::dict[1]
/key[.='V91A']/following-sibling::dict[1]
/key[.='DB2 Distributed Access Information']/following-sibling::dict[1]
/key[.='IP Address']/following-sibling::dict[1]
/key[.='Value']/following-sibling::string[1]"

XML_OUTPUT: "::9.30.189.213"
```

GET_CONFIG - ZParm value (APPENSCH):

```
"/plist/dict/key[.='DB2 Subsystem Specific Information']/following-sibling::dict[1]
/key[.='V91A']/following-sibling::dict[1]
/key[.='DB2 Subsystem Parameters']/following-sibling::dict[1]
/key[.='DSNHDECP']/following-sibling::dict[1]
/key[.='APPENSCH']/following-sibling::dict[1]
/key[.='Subsystem Parameter Value']/following-sibling::dict[1]
/key[.='Value']/following-sibling::string[1]"

XML_OUTPUT: "EBCDIC"
```

GET_MESSAGE - Short Message Text:

```
"/plist/dict/key[.='Short Message Text']/following-sibling::dict[1]
/key[.='Value']/following-sibling::string[1]"

XML_OUTPUT: "DSNT408I SQLCODE = -805, ERROR: DBRM OR PACKAGE NAME
REDBOOK-consistency-token NOT FOUND IN PLAN REDBOOK-plan-name. REASON 8"
```

GET_SYSTEM_INFO - Real Storage:

```
"/plist/dict/key[.='Operating System Information']/following-sibling::dict[1]
/key[.='Real Storage Size']/following-sibling::dict[1]
/key[.='Value']/following-sibling::integer[1]"

XML_OUTPUT: "256"
```

Note: The XPath expression must always be absolute from the root node.

Full XML XPath definition is not supported in the Common SQL API, it only offers limited capabilities. For example, currently XML_FILTER cannot be used for filtering elements in an array. For a complete list of all XPath limitations, refer to *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840.

CALLING the CSA stored procedures

For a comprehensive Java sample that employs all three stored procedures, refer to A.10, "Invoking the Common SQL API stored procedures" on page 870.

Furthermore, Appendix B contains a very simple Java program that invokes the GET_CONFIG stored procedure with a valid XPath to only retrieve the value of the IP Address. Refer to A.10.1, "Simple GET_CONFIG invocation with a valid XPath" on page 883.

24.5 Using the DB2-supplied stored procedures

In this section, we list the complete applications written in Java that call DB2-supplied stored procedures. The actual listings are in Appendix A, “Samples for using DB2-supplied stored procedures” on page 807, and are available for download as described in Appendix B, “Additional material” on page 887. To compile and run the applications, you have to install the Java 2 SDK, Standard Edition, Version 1.4.0 or higher. If you develop on Windows, we recommend setting the PATH variable to include the path to the Java 2 SDK executables (javac.exe, java.exe, javadoc.exe, etc.) so that you can use them from any directory without having to type the full path of the command. Read the README file in your Java directory for detailed instructions on how to set the PATH permanently.

Create a directory where you will be saving your source code files. Open a command prompt window and change to that directory. Once you have created a file, such as AdminSystemInformation.java there, compile it with the following command:

```
javac AdminSystemInformation.java
```

The compiler stores the byte code program for the class or classes defined in the source file with the extension .class. To execute the byte code program, enter the following command:

```
java AdminSystemInformation DBALIAS USERID PASSWORD
```

Most of the programs require you to enter the database alias, the user ID, and the password to run the program.

24.5.1 Source code for activating DB2-supplied stored procedures

We describe the sample source code for the DB2-supplied stored procedures in Appendix A, “Samples for using DB2-supplied stored procedures” on page 807. The source code is available as described in B.1.9, “Sample code to invoke DB2-supplied stored procedures” on page 891.

Table 24-42 shows the functions available and where they are listed in Appendix A, “Samples for using DB2-supplied stored procedures” on page 807.

Table 24-42 Source code for DB2 stored procedures invocation

Name	Description	DB2 provided function	Source code
AdminSystemInformation	Display DB2 system information	ADMIN_INFO_SSID, ADMIN_INFO_HOST, DSNWZP, DSNUTILU,	A.1, “Display DB2 system information with AdminSystemInformation” on page 808
AdminWLMRefresh	Refresh a WLM environment	WLM_REFRESH	A.2, “Refresh a WLM environment with AdminWLMRefresh” on page 817
AdminDB2Command	Issue DB2 commands	ADMIN_COMMAND_D2	A.3, “Issue DB2 commands with AdminDB2Command” on page 819
AdminUtilityExecution	Automate RUNSTATS	DSNACCOX, ADMIN_UTL_SCHEDULE	A.4, “Automate RUNSTATS with AdminUtilityExecution” on page 827

Name	Description	DB2 provided function	Source code
AdminDataSet	Manage data sets	ADMIN_DS_WRITE, ADMIN_DS_BROWSE, ADMIN_DS_RENAME, ADMIN_DS_SEARCH, ADMIN_DS_LIST, ADMIN_DS_DELETE	A.5, “Manage data sets with AdminDataSet” on page 838
AdminJob	Submit JCL to compress an existing PDS	ADMIN_JOB_SUBMIT, ADMIN_JOB_QUERY, ADMIN_JOB_FETCH, ADMIN_JOB_CANCEL	A.6, “Submit JCL with AdminJob” on page 845
AdminUNIXCommand	Issue USS commands	ADMIN_COMMAND_UNIX	A.7, “Issue USS commands with AdminUNIXCommand” on page 852
AdminDSNSubcommand	Issue a REBIND PACKAGE command	ADMIN_COMMAND_DSN	A.8, “Issue DSN subcommands with AdminDSNSubcommand” on page 855
AdminSchedule1 AdminScheduleR Trigger: USER.TR_WLM_REFR	Multiple Use Cases and code for scheduling administrative tasks. Code for housekeeping of expired tasks	ADMIN_TASK_ADD, ADMIN_TASK_REMOVE	A.9, “Task Scheduler Sample Use cases” on page 858
SPDriver, SPWrapper, GetConfigDriver	Invoke Common SQL API stored procedures, XML handling with Jakarta Commons Configuration	GET_SYSTEM_INFO, GET_MESSAGE, GET_CONFIG	A.10, “Invoking the Common SQL API stored procedures” on page 870

24.6 Summary

In this chapter we discussed the stored procedures that are shipped with DB2 and how to use them in an application program. Whether you code your application program in Java or any other programming language such as C, the applications will enable you to correctly implement calling the stored procedure, passing and retrieving parameters and result sets, and handling error conditions correctly.

25

Using LOBs and XML

In this chapter we introduce some considerations on accessing large objects (LOBs) from stored procedures.

This chapter contains the following:

- Introduction to LOBs
- Setting up the environment for sample LOB tables
- Support for LOBs in Java
- Stored procedure returning a BLOB column
- Stored procedure returning a CLOB column
- Introduction to XML
  - Setting up the environment for sample XML tables
  - Use of the XML data type in stored procedure parameters
  - Use of the XML data type in stored procedure result sets

25.1 Introduction to LOBs

The term *large object* and the acronym *LOB* refer to database objects that you can use to store large amounts of data. A DB2 LOB is a varying-length character string that can contain up to (2 GB - 1) of data.

The three DB2 LOB data types are:

- Binary large object (BLOB)

  Use a BLOB to store binary data such as pictures, voice, and mixed media.

- Character large object (CLOB)

  Use a CLOB to store SBCS or mixed character data such as documents.

- Double-byte character large object (DBCLOB)

  Use a DBCLOB to store data that consists of only DBCS data.

Working with LOBs involves defining the LOBs to DB2, moving the LOB data into DB2 tables, then using SQL operations to manipulate the data. This chapter concentrates on accessing LOB data through stored procedures. For general information on LOBs, see *LOBs with DB2 for z/OS: Stronger and Faster,* SG24-7270. For information on defining LOBs to DB2, see Chapter 5 of *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854. For information on how DB2 utilities manipulate LOB data, see Part 2 of *DB2 Version 9.1 for z/OS Utility Guide and Reference,* SC18-9855.

There are four basic steps in defining LOBs and moving data to DB2:

1. Define a DB2 table with a column of the appropriate LOB type and a row identifier (ROWID) column. Define only one ROWID column even if there are multiple LOB columns in the table. The LOB column holds information about the LOB not the LOB data itself. The table that contains the LOB information is called the base table. DB2 uses the ROWID column to locate your LOB data. You need only one ROWID column in a table that contains one or more LOB columns. You can define the LOB column and the ROWID column in a CREATE TABLE or ALTER TABLE statement. if you are adding a LOB column and a ROWID column to an existing table, you must use two ALTER TABLE statements. Add the ROWID with the first ALTER TABLE statement and the LOB column with the second in DB2 V8 if no ROWID column exists. When you define an LOB column (by using a CREATE TABLE or ALTER TABLE statement), DB2 implicitly creates a ROWID column and appends it as the last column of the table.
2. Create the table space and table to hold the LOB data. The table space and table are called LOB table space and auxiliary table. If your base table is non partitioned, you must create one LOB table space and one auxiliary table for each LOB column. If your base table is partitioned, for each LOB column you must create one LOB table space and one auxiliary table for each partition. For example, if your base table has three partitions, you must create three LOB table spaces and three auxiliary tables for each LOB column. Create these objects using the CREATE LOB TABLESPACE and CREATE AUXILIARY TABLE statements.
3. Create an index on the auxiliary table.
4. Put the LOB data into DB2. If the total length of an LOB column and the base table row is less than 32 KB, you can use the LOAD utility to put the data in DB2. Otherwise, you must use INSERT or UPDATE statements. Even though the data is stored in the auxiliary table, the LOAD utility statement or INSERT statement specifies the base table. Using INSERT can be complicated because your application will need enough storage to hold the entire value that goes into the LOB column.

Note: In DB2 V9, if the base table is created implicitly, all those LOB-related objects are created automatically. This in turn is now independent from the CURRENT RULES setting. DB2 still supports SET CURRENT RULES = 'STD' automatic creation. The automatic creation does not apply for explicitly created table spaces.

25.2 Setting up the environment for sample LOB tables

DB2 provides you with sample job DSNTEJ7, which creates and populates a table with LOB data. This sample job along with others demonstrates how to work with LOB objects. Those jobs and the programs used in those jobs can be found in data set <hlq>.SDSNSAMP. Those jobs are referred to as Phase 7 of the Installation Verification Procedure. All job names start with DSNTEJ7. For our purposes we only ran job DSNTEJ7 to create and populate table DSN8910.EMP_PHOTO_RESUME.

We used the IBM supplied sample tables for our case study. The LOB table DDL used is listed in Example 25-1.

Example 25-1 LOB table used in the case study

```
CREATE TABLE  DSN8910.EMP_PHOTO_RESUME
 (      EMPNO  CHAR( 06 )   NOT NULL,
    EMP_ROWID  ROWID NOT NULL GENERATED ALWAYS,
   PSEG_PHOTO  BLOB( 500K ),
    BMP_PHOTO  BLOB( 100K ),
       RESUME  CLOB(   5K ),
  PRIMARY KEY  ( EMPNO ) )
           IN  DSN8D91L.DSN8S91B
        CCSID  EBCDIC;
```

25.3 Support for LOBs in Java

The maximum size allowed for an LOB parameter defined as OUT or INOUT on a Callable Statement is the same as any LOB (2 GB-1). Stored procedures do not support LOB locators to be used as input/output parameters. LOB locators can be used with all languages except Java.

25.4 Stored procedure returning a BLOB column

In this section we show how to implement a Java stored procedure handling a BLOB column.

25.4.1 Description of the EmpPhot.java stored procedure

A Java stored procedure called EmpPhotJ was developed to return a BLOB column. The stored procedure takes the employee number in input and returns the BMP_PHOTO column in a BLOB format in output. The DDL used for creating the stored procedure EMPPHOTJ is shown in Example 25-2.

Example 25-2 Sample CREATE PROCEDURE with BLOB

```
CREATE PROCEDURE DEVL7083.EMPPHOTJ ( IN  EMPNO      CHARACTER(6),
                                     OUT EMPPHOTO    BLOB(100K),
                                     OUT OUTPUTMESSAGE VARCHAR(250))
```

```
    EXTERNAL NAME 'EmpPhotJ.GetEmpDtls'
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    COLLID DEVL7083
    PROGRAM TYPE SUB
    WLM ENVIRONMENT DB9ADJC2
```

Tip: Before you can execute this CREATE PROCEDURE statement, the executable load module must exist in the CLASSPATH for your Java stored procedures. Refer to Chapter 13, “Java stored procedures” on page 181 to learn more about how to set up Java stored procedures.

The sample code for the Java stored procedure is in Example 25-3.

Example 25-3 EmpPhotJ.java

```
import java.sql.*;
import java.io.*;
import java.math.*;

public class EmpPhotJ {

public static void GetEmpDtls(
       String  empno,
       Blob[] emp_photo,                                                    1
       String[] outputMessage)
  {
       Connection conndb2 = null;
       String sql = " ";
       outputMessage[0] = " ";
        try {
            // Use an existing connection to DB2

            conndb2 = DriverManager.getConnection("jdbc:default:connection");
            Statement stmtdb2 = conndb2.createStatement();

            sql = "SELECT EMPNO,BMP_PHOTO from DSN8810.EMP_PHOTO_RESUME"
                + " WHERE EMPNO = '"+ empno + "'";
            ResultSet rs =  stmtdb2.executeQuery(sql) ;
            if (rs.next())
              {
               empno     = rs.getString("EMPNO");
               emp_photo[0]   = rs.getBlob("BMP_PHOTO");                    2
              }
        }
    catch (SQLException e)
    {
           outputMessage[0]  = "SQLException raised, SQLState = "
           + e.getSQLState() + " SQLCODE = " + e.getErrorCode()
           + " :" + e.getMessage();
    }
        catch (Exception e) {
            outputMessage[0] = e.toString();
        }
    }
}
```

Be aware of the following points about the code listed in Example 25-2:

- The emp_photo variable is defined as an output variable of type java.sql.Blob.
- The getBlob method is used to retrieve the BLOB column.
- LOB materialization is important when using BLOBs in a stored procedure. LOB materialization means that DB2 places an LOB value into contiguous storage in a data space. Because LOB values can be very large, DB2 avoids materializing LOB data until absolutely necessary. However, DB2 must materialize LOBs when your application program moves an LOB into or out of a stored procedure.
- You cannot use LOB locators as input or output parameters for Java stored procedures.

25.4.2 Invoking the EmpPhotJ stored procedure

We created the Java servlet EmpPhotoSpServlet.java to invoke the stored procedure. The servlet runs on a WebSphere Application Server V5 for Windows.The servlet code converts the BLOB data returned by the stored procedure into a binary output stream, and writes it out to the Web page. The points to note about this code (listed in Example 25-4) are:

1. Import the required standard Java classes for servlet and JDBC calls.
2. The servlet uses the Java Universal Driver to connect to the database on z/OS. Statement **"**`Class.forName("com.ibm.db2.jcc.DB2Driver")`**"** loads the classes for the Java Universal Driver (JCC). The JCC driver comes with DB2 Connect V8; in case you are on DB2 Connect V7, you need to use different classes.

Example 25-4 EmpPhotoSpServlet.java

```
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;                              1.
import java.io.*;
import java.sql.*;

public class EmpPhotoSpServlet extends HttpServlet implements Servlet {

   public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
      InputStream inps = null;
      int nread;

      try {
         resp.setContentType("image/bmp");
         OutputStream out = resp.getOutputStream();
         Class.forName("com.ibm.db2.jcc.DB2Driver");                        2.
         Connection con =
            DriverManager.getConnection(
               "jdbc:db2://wtsc63.itso.ibm.com:12345/DB9A",
               "paolor7",
               "bhas11");                                                   3.
         CallableStatement cstmt =
            con.prepareCall("CALL DEVL7083.EMPPHOTJ(?,?,?)");
         cstmt.setString(1, "000130");
         cstmt.registerOutParameter(2, Types.BLOB);                         4
         cstmt.registerOutParameter(3, Types.VARCHAR);
         cstmt.execute();
         resp.setContentType("image/bmp");                                  5
         inps = cstmt.getBlob(2).getBinaryStream();                         6
```

```
            byte[] buf = new byte[1024];
            while ((nread = inps.read(buf)) > 0)
               out.write(buf, 0, nread);                                    7

         } catch (Exception e) {
            System.out.println("Error found" + e.toString());
         }

      }

   }
```

3. Once the classes for the Universal Driver are loaded, the format of the connection string decides the type of connection. We can use either a Type 4 or a Type 2 connection. In our example we are using a Type 4 connection. The syntax for the Type 4 connection string is shown in Example 25-5.

Example 25-5 Type4 Connection in a java Universal Driver

```
DriverManager.getConnection(Url,uid,pwd)

url - jdbc:db2://server:port/databasename
uid - userid
pwd - password

server       - wtsc63.itso.ibm.com
port         - 12345
databasename - DB9A (Location Name for z/OS)
userid       - paolor7
password     - bhas11
```

4. Register the output parameter as a BLOB.
5. The content type of the response needs to be set to image/bmp; by default the content is text.
6. getBinaryStream Method converts the BLOB data to a binary stream of data and assigns it to a variable inps of class Inputstream.
7. We read the input stream in chunks of 1024 bytes and write it to the output, in our case to the Web browser. We create a byte[]array of size 1024 and use the read method. The read method populates the buffer and returns the number of bytes read. We need to invoke the read method in a loop until there are no more bytes to be read.

25.4.3 Invoking the servlet EmpPhotoSpServlet

The servlet was developed and tested on WebSphere Application Developer V5. The sample URL is:

`http://localhost:9080/DBASPSERV/servlet/EmpPhotoSpServlet`

25.4.4 Handling large BLOB columns

You need to be aware of the following issues while handling BLOBs in Java stored procedures or applications.

Assume that you have a column defined as a BLOB of 100 MB, and that you have a picture of 0.5 MB stored in this column. When you issue the getBlob() Method to retrieve the BLOB data, DB2 tries to allocate and look for a storage of about 100 MB even though the data in the

column is only 0.5 MB. When handling large LOB columns, it is better to retrieve BLOB data in chunks. Otherwise, you will get an outofMemory abend.

Example 25-6 shows a stored procedure EXTRACT_JAR that extracts a BLOB column named JAR_DATA from a DB2 Catalog Table SYSIBM.SYSJAROBJECTS. The stored procedure creates an HFS file that contains the extracted jar file.The JAR_DATA column is defined as BLOB of 100 MB, and holds the jar file.

Example 25-6 Java stored procedure handling large BLOBs

```
/*  Logic for extracting the  Blob data from a DB2 Table
   First we get the total length of the BLob Data and store
   it in variable totLenBlob.
   Then we do a SUBSTRING to read the Blob in chunks of 4k
   , we invoke the substring function inside a loop , check if we
   reached the end of the Blob and then exit .
   spos - is the starposition for the substring function , needs to be
          increamented after each invocation.
   len  - is the third argument of the substring function , it
          contains the length of the blob that needs to be extracted
          in each invocation.
 */
import java.sql.*;
import java.io.*;

public class ExtractJarSp {

   public static void GetJarFile(
      String schemaName,
      String jarID,
      String fileName,
      String[] outputMessage) {
      Connection conndb2 = null;
      outputMessage[0] = " ";
      try {
         Blob jarBlob;
         InputStream inpStream = null;
         ;
         String sqltxt = null;
         int nread = 0;
         int totLenBlob = 0;
         int spos = 1; /* start pos - argument to   substring function */
         int len = 0; /* length of string to be  extracted */
         int bufflen = 4096; /* buffer length */
         char exit = 'n';
         byte[] byteArray = new byte[4096]; /* buffer of 4K */
         FileOutputStream outFile = new FileOutputStream(fileName);
         conndb2 = DriverManager.getConnection("jdbc:default:connection");
         Statement stmt = conndb2.createStatement();

         sqltxt =
            "SELECT Length(JAR_DATA) FROM SYSIBM.SYSJAROBJECTS WHERE JAR_ID = "
               + "'"
               + jarID
               + "'"
               + "and JARSCHEMA = '"
               + schemaName
               + "'";
         ResultSet rs = stmt.executeQuery(sqltxt);
         if (rs.next())
```

```
            totLenBlob = rs.getInt(1);
         if (totLenBlob < bufflen)
            len = totLenBlob;
         else
            len = bufflen;
         while (exit == 'n') {
            sqltxt =
               "SELECT SUBSTR(JAR_DATA,"
                  + spos
                  + ","
                  + len
                  + ") "
                  + " FROM SYSIBM.SYSJAROBJECTS WHERE JAR_ID = "
                  + "'"
                  + jarID
                  + "'"
                  + "and JARSCHEMA = '"
                  + schemaName
                  + "'";
            rs = stmt.executeQuery(sqltxt);
            if (rs.next())
               inpStream = rs.getBlob(1).getBinaryStream();
            nread = inpStream.read(byteArray);
            outFile.write(byteArray, 0, nread);
            spos = spos + len;
            if (spos >= totLenBlob) /* no more data to read */
               exit = 'y';
            if ((totLenBlob - spos) > bufflen)
               len = bufflen;
            else
               len = totLenBlob - spos + 1;
         }
         outFile.close();
         stmt.close();
         rs.close();

         File fname = new File("Employee.jar");
         System.out.println("Extracted jar is  " + fname.getAbsolutePath());
      } catch (SQLException e) {
         outputMessage[0] =
            "SQLException raised, SQLState = "
               + e.getSQLState()
               + " SQLCODE = "
               + e.getErrorCode()
               + " :"
               + e.getMessage();
      } catch (Exception e) {
         outputMessage[0] = e.toString();
      }
   }
}
```

In our example, we read the BLOB data in chunks of 4 KB, we issue a SUBSTRING command multiple times, and at each invocation we read a 4 KB chunk, and write it to an HFS file.

The stored procedure takes three input parameters: SCHEMANAME, JAR_ID, and the HFS File Name. It extracts the BLOB column and puts it into the file name. Example 25-7 shows the DDL used for creating the stored procedure.

Example 25-7 DDL for EXTRACT_JAR stored procedure

```
CREATE PROCEDURE DEVL7083.EXTRACT_JAR
( IN SCHEMANAME CHARACTER(8),
  IN JARID        CHAR(18),
  IN FILENAME     VARCHAR(100),
  OUT OUTPUTMESSAGE VARCHAR(250))
EXTERNAL NAME 'ExtractJarSp.GetJarFile'
LANGUAGE JAVA
PARAMETER STYLE JAVA
COLLID DSNJDBC
PROGRAM TYPE SUB
WLM ENVIRONMENT DB9ADJC2
```

25.5 Stored procedure returning a CLOB column

A Java stored procedure called EmpClobJ has been developed to return a CLOB column. The stored procedure takes the employee number as input and returns the RESUME column in a CLOB format. The DDL used for the CLOB example of the Java stored procedure can be found in Example 25-8.

Example 25-8 DDL for EMPCLOB stored procedure

```
CREATE PROCEDURE DEVL7083.EMPCLOBJ ( IN  EMPNO      CHARACTER(6),
                                     OUT EMPCLOB    CLOB(5K),
                                     OUT OUTPUTMESSAGE VARCHAR(250))
    EXTERNAL NAME 'EmpClobJ.GetClobDtls'
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    COLLID DEVL7083
    PROGRAM TYPE SUB
    WLM ENVIRONMENT DB9ADJC2
```

The DDL used for creating the stored procedure can be found in Example 25-9.

Example 25-9 EmpClobJ java

```
import java.sql.*;
import java.io.*;
import java.math.*;

public class EmpClobJ {

public static void GetClobDtls(
      String  empno,
      Clob[] empClob,                                                  1
      String[] outputMessage)
 {
      Connection conndb2 = null;
      String sql = " ";
      outputMessage[0] = " ";
       try {
           // Use an existing connection to DB2

           conndb2 = DriverManager.getConnection("jdbc:default:connection");
           Statement stmtdb2 = conndb2.createStatement();

           sql = "SELECT EMPNO,RESUME from DSN8810.EMP_PHOTO_RESUME"
```

```
                + " WHERE EMPNO = '"+ empno + "'";
            ResultSet rs =  stmtdb2.executeQuery(sql) ;
            if (rs.next())
              {
               empno     = rs.getString("EMPNO");
               empClob[0]   = rs.getClob("RESUME");                              2
              }
        }
    catch (SQLException e)
    {
           outputMessage[0]  = "SQLException raised, SQLState = "
           + e.getSQLState() + " SQLCODE = " + e.getErrorCode()
           + " :" + e.getMessage();
    }
        catch (Exception e) {
            outputMessage[0] = e.toString();
        }
    }
}
```

Note the following:

- The empClob variable is defined as an output variable of the type java.sql.Clob.
- The getClob method is used to retrieve the CLOB column.

25.5.1 Invoking the EmpClobJ stored procedure

We created the Java servlet EmpClobSpServlet.java to invoke the stored procedure. The servlet runs on a WebSphere Application Server V5 for Windows. The servlet code converts the CLOB data returned by the stored procedure into an ASCII output stream, and writes it out to the Web page. The code for the servlet is shown in Example 25-10.

Example 25-10 EmpClobSpServlet

```
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.sql.*;



public class EmpClobSpServlet extends HttpServlet implements Servlet {


   public void doGet(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {
         Reader  inps = null;
         int nread;

         try {
            resp.setContentType("text/html");
            PrintWriter out = resp.getWriter();
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            Connection con =
               DriverManager.getConnection(
                  "jdbc:db2://wtsc63.itso.ibm.com:12345/DB9A",
```

```
                    "paolor7",
                    "bhas11");
            CallableStatement cstmt =
               con.prepareCall("CALL DEVL7083.EMPCLOBJ(?,?,?)");
            cstmt.setString(1, "000130");
            cstmt.registerOutParameter(2, Types.CLOB);
            cstmt.registerOutParameter(3, Types.VARCHAR);
            cstmt.execute();
            inps = cstmt.getClob(2).getCharacterStream() ;
              out.println("<HTML><BODY>");
              out.println("<P><B> This page is built by a java servlet EmpClobSpServlet "
               + " which calls a DB2 Stored Procedure which in turn returns a Resume Data
stored as a CLOB on DB2 V8 . </P>");
            out.println("</B><P>");
            char[] buf = new char[1024];
            while ((nread = inps.read(buf)) > 0)
               out.write(buf, 0, nread);
               out.println("</P></BODY><HTML>");
         } catch (Exception e) {
            System.out.println("Error found" + e.toString());
         }

      }

   }
```

25.6 Introduction to XML

XML is a language that uses tags to describe data. Since XML is not fixed format, it can combine many diverse types of data into many diverse structures. This lack of fixed format makes it easy to exchange data between different companies and different hardware and software technologies.

IBM, recognizing the need for XML support, has introduced, with DB2 9 for z/OS as well as with DB2 for Linux, UNIX and Windows pureXML™, a collection of XML capabilities that are built into the DB2 9 family. DB2 is now an hybrid database fully supporting relational and XML data.

Today, XML is predominant in most organizations and hosts an abundance of business information on public and private Web sites. This is because XML is vendor and platform independent, and is a very flexible data model for structured data, semi-structured data, and schema-less data. It is also self-describing and easy to extend. Furthermore, XML can be easily transformed into other XML documents or even into different formats such as HTML. Therefore, XML is the de facto standard for exchanging data across different systems, platforms, applications, and organizations.

Beyond XML for data exchange, enterprises are keeping large amounts of business-critical data permanently in XML format. This has various reasons, such as a need to keep it for auditing and regulatory compliance. Also, in life science applications, for example, the data is highly complex and hierarchical in nature and yet may contain significant amounts of unstructured information. Most of today's genomic data is still kept in proprietary flat file formats, but major efforts are under way to move to XML. These proprietary flat files can be accessed using WebSphere Federated Server technology.

Web services and SOA use XML as the base for most services and data. Almost every implementation of service-oriented architecture (SOA) includes XML at some point.

XML document storage

The XML column data type is provided for storage of XML data in DB2 tables. Most SQL statements support the XML data type. This enables you to perform many common database operations with XML data, such as creating tables with XML columns, adding XML columns to existing tables, creating indexes over XML columns, creating triggers on tables with XML columns, and inserting, updating, or deleting XML documents.

XML document retrieval

You can use SQL to retrieve entire documents from XML columns, just as you retrieve data from any other type of column. When you need to retrieve portions of documents, you can specify XPath expressions, through SQL with XML extensions (SQL/XML).

Application development

Application development support of XML enables applications to combine XML and relational data access and storage. The following programming languages support the new XML data type:

- Assembler
- C and C++ (embedded SQL or DB2 ODBC)
- COBOL
- Java (JDBC and SQLJ)
- PL/

25.6.1 Using the XML data type

A CREATE TABLE statement that includes the use of the XML data type is shown in Figure 25-1.

```
CREATE TABLE XMLTAB1 (C1 CHAR(10), C2 XML)
```

Figure 25-1 CREATE TABLE with XML column

The result of this statement is a table consisting of two columns, character column C1 and column C2 with data type XML.

As a result of issuing the CREATE TABLE statement shown in Figure 25-1, the following objects have also been created implicitly by DB2 to support the XML column:

- A column called DB2_GENERATED_DOC_ID_FOR_XML. We refer to this column as DocID column from now on. DocID uniquely represents each row. This column is hidden. For each table, DB2 only needs one DocID column even if you would add additional rows with data type XML. This DocID is defined as generated always, meaning that you cannot update the DocID column.
- A unique index on the DocID column that is defined as NOT NULL. This index is known as a document ID index.
- An XML table space. The XML table space always uses the Unicode UTF-8 encoding scheme.
- An XML table with columns docid, min_nodeid, and xmldata.
- A NodeID index on the XML table with key DocID and min_nodeid

Note: These DB2-required objects are always generated automatically. Special register CURRENT RULES has no influence on this.

25.6.2 Setting up the environment for sample XML tables

As for any other new functions, the sample DB2 objects which are created when you execute the IVP have been extended for the use of the new XML data type. The XML data type is used in 5 new sample tables which are created using job DSNTEJ1.

A whole new set of tables containing XML columns are now defined through execution of job DSNTEJ1. Currently the IVP does not contain any help to populate those tables. The sample database on DB2 for LUW contains the same tables. Here they are populated with a couple of rows containing XML documents as well as relational data. Unfortunately, the structure of the tables is not the same as for DB2 for z/OS.

Refer to Chapter 3, “Our case study” on page 23 if you want to learn an easy way to populate these tables on DB2 for z/OS with the data rows that reside on DB2 for LUW using Data Studio.

The DDL used in job DSNTEJ1 to create the existing sample tables is shown in Example 25-11, Example 25-12, Example 25-13, Example 25-14, and Example 25-15.

Example 25-11 DSN8910.PRODUCT sample table

```
CREATE TABLE DSN8910.PRODUCT
  ( PID             VARCHAR(10)   NOT NULL PRIMARY KEY
   ,NAME            VARCHAR(128)
   ,PRICE           DECIMAL(30, 2)
   ,PROMOPRICE      DECIMAL(30, 2)
   ,PROMOSTART      DATE
   ,PROMOEND        DATE
   ,DESCRIPTION     XML )
  IN DSN8D91X.DSN8S91X
  CCSID EBCDIC;
```

Example 25-12 DSN8910.CUSTOMER sample table

```
CREATE TABLE DSN8910.CUSTOMER
  ( CID             BIGINT       NOT NULL PRIMARY KEY,
    INFO            XML,
    HISTORY         XML )
  IN DSN8D91X.DSN8S91X
  CCSID EBCDIC;
```

Example 25-13 DSN8910.PURCHASEORDER sample table

```
CREATE TABLE DSN8910.PURCHASEORDER
  ( POID            BIGINT       NOT NULL PRIMARY KEY,
    STATUS          VARCHAR(10)  NOT NULL WITH DEFAULT 'New',
    PORDER          XML )
  IN DSN8D91X.DSN8S91X
  CCSID EBCDIC;
```

Example 25-14 DSN8910.CATALOG sample table

```
CREATE TABLE DSN8910.CATALOG
  ( NAME            VARCHAR(128) NOT NULL PRIMARY KEY,
    CATLOG          XML )
```

```
  IN DSN8D91X.DSN8S91X
  CCSID EBCDIC;
```

Example 25-15 DSN8910.SUPPLIERS sample table

```
CREATE TABLE DSN8910.SUPPLIERS
  ( SID              VARCHAR(10) NOT NULL PRIMARY KEY,
    ADDR             XML )
  IN DSN8D91X.DSN8S91X
  CCSID EBCDIC;
```

25.6.3 Use of the XML data type in stored procedure parameters

In case you plan to retrieve XML documents from relational tables containing columns defined with data type XML, or in case you plan to insert XML documents into your DB2 table columns defined with data type XML—it is the same doing it using stored procedures as doing it using any other application.

However, even if inserting and retrieving data is the same as using other applications, IN, OUT, and INOUT parameters can never have data type XML associated to them.

If you tried to use data type XML for an IN, OUT, or INOUT parameter of your stored procedure, you would end up in SQLCODE -20060, as shown in Figure 25-2, where we tried to create a very simple native SQL stored procedure. The provided example contains almost no programming logic because our objective here is to provide the information that is important regarding the usage of the XML data type. If you are interested to learn more about native SQL stored procedures in general, or if you are looking for more complex examples, we recommend that you refer to Chapter 15, “Native SQL procedures” on page 253.

```
CREATE PROCEDURE DEVL7083.PURCHASE_XML_SELECT1(
            OUT POID INTEGER,
            OUT PORDER XML)
 VERSION VERSION1
 ISOLATION LEVEL CS
 RESULT SETS 1
 LANGUAGE SQL
-------------------------------------------------------------------------
-- NATIVE SQL STORED PROCEDURE
-------------------------------------------------------------------------
P1: BEGIN
  SELECT POID, PORDER FROM  DEVL7083.PURCHASEORDER2;
---------+---------+---------+---------+---------+---------+---------+-----
DSNT408I SQLCODE = -20060, ERROR:  UNSUPPORTED DATA TYPE XML ENCOUNTERED IN
         SQL PARAMETER PORDER
DSNT418I SQLSTATE   = 560AB SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNHSMS1 SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = 11 0  0  -1  169  3102 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'0000000B'  X'00000000'  X'00000000'  X'FFFFFFFF'
         X'000000A9'  X'00000C1E' SQL DIAGNOSTIC INFORMATION
```

Figure 25-2 SQLCODE -20060 error message

Retrieving XML data using stored procedures

There are several ways to retrieve data stored in XML documents. The first and very simple approach is to retrieve the XML documents “as is” from your XML column. When doing so, the data is returned to your application as data type XML.

As a result, the attempt to select the contents of an XML column, PORDER in our case, into an output variable defined as CHAR or VARCHAR fails, as shown in Figure 25-3.

```
CREATE PROCEDURE XML_SELECT_STAR1
( OUT PORDEROUT VARCHAR(2000) )
 VERSION VERSION1
 ISOLATION LEVEL CS
 RESULT SETS 1
P1: BEGIN
  SELECT PORDER
    INTO PORDEROUT FROM DEVL7083.PURCHASEORDER1  ;
 END P1#
---------+---------+---------+---------+---------+---------+---------+------
DSNT408I SQLCODE = -408, ERROR:  THE VALUE IS NOT COMPATIBLE WITH THE DATA
         TYPE OF ITS TARGET. TARGET NAME IS PORDEROUT
DSNT418I SQLSTATE   = 42821 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNXOYPL SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = -300 0  1  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'FFFFFED4'  X'00000000'  X'00000001'  X'FFFFFFFF'
         X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
```

Figure 25-3 Assignment of XML data type to CHAR OUT parameter fails

To overcome this problem and to be able to return the contents of your XML data column as OUT parameter to your calling application, you must make sure that the contents of your XML column are cast to another data type. XMLSERIALIZE is the only function that you can use to initiate the casting. The only two data types that are valid target data types for this cast are CLOB and DBCLOB.

Refer to Figure 25-4, which shows how to use XMLSERIALZE to cast the XML data type, for example, to CLOB data type. Since CLOB and VARCHAR are data types compatible for cast, there is no need to also change the definition of OUT parameter PORDEROUT to CLOB.

```
CREATE PROCEDURE DEVL7083.XMLSEL1
( OUT PORDEROUT VARCHAR(2000))
 VERSION VERSION1
 ISOLATION LEVEL CS
 RESULT SETS 1
P1: BEGIN
      SELECT XMLSERIALIZE(PORDER AS CLOB)
      INTO PORDEROUT
      FROM DEVL7083.PURCHASEORDER1  ;
 END P1#
```

Figure 25-4 XMLSERIALIZE function to cast to CLOB

In case you are not interested in the whole XML document and are using the XMLQUERY function to retrieve parts of your XML documents with XPATH statement syntax, the returned data type is also XML. There is no way to directly cast the result of XMLQUERY to any other data type directly. You must use XMLSERIALIZE here as well as the OUT function to finally cast the results from XML to either CLOB or DBCLOB.

Figure 25-5 on page 624 shows a sample XML document that is provided through our sample data for column PORDER of table DSN8910.PURCHASEORDER.

```
<PurchaseOrder xmlns="http://posample.org" PoNum="5000" OrderDate="2006-02-18"
Status="Unshipped">
   <item>
      <partid>100-100-01</partid>
      <name>Snow Shovel, Basic 22 inch</name>
      <quantity>3</quantity>
      <price>9.99</price>
   </item>
   <item>
      <partid>100-103-01</partid>
      <name>Snow Shovel, Super Deluxe 26 inch</name>
      <quantity>5</quantity>
      <price>49.99</price>
   </item>
</PurchaseOrder>
```

Figure 25-5 Sample XML document from PORDER column

Refer to Figure 25-6 to verify the above statement. Our attempt to create a native stored procedure in which we used an XMLQUERY statement to determine the value that is passed to our OUT parameter PORDEROUT fails, because the result of the XMLQUERY statement is represented in data type XML.

```
CREATE PROCEDURE XMLSEL2
( OUT PORDEROUT VARCHAR(2000))
 VERSION VERSION1
 ISOLATION LEVEL CS
 RESULT SETS 1
P1: BEGIN
       SELECT XMLQUERY('//*' PASSING PORDER)
       INTO PORDEROUT
       FROM DEVL7083.PURCHASEORDER1  ;
 END P1#
---------+---------+---------+---------+---------+---------+---------+----
DSNT408I SQLCODE = -408, ERROR:  THE VALUE IS NOT COMPATIBLE WITH THE DATA
         TYPE OF ITS TARGET. TARGET NAME IS PORDEROUT
DSNT418I SQLSTATE   = 42821 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNXOYPL SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = -300 0  1  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'FFFFFED4'  X'00000000'  X'00000001'  X'FFFFFFFF'
         X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
```

Figure 25-6 Failing attempt to assign XMLQUERY result to parameter

As stated above, the workaround for this is to use scalar function XMLSERIALIZE to cast data types from XML to either CLOB of DBCLOB. Example 25-7 on page 625 shows you once again how this can be done.

```
CREATE PROCEDURE XMLSEL3
( OUT PORDEROUT VARCHAR(2000))
 VERSION VERSION1
 ISOLATION LEVEL CS
 RESULT SETS 1
P1: BEGIN
       SELECT XMLSERIALIZE(XMLQUERY('//*' PASSING PORDER) AS CLOB)
       INTO PORDEROUT
       FROM DEVL7083.PURCHASEORDER1  ;
 END P1#
```

Figure 25-7 XMLSERIALIZE around XMLQUERY scalar function

The result of the execution of stored procedure XMLSEL3 is the whole XML document as shown in Figure 25-5 on page 624 in a serialized format, that is, as a long character string.

Inserting XML data using stored procedures

So far we have talked about how to retrieve XML data using stored procedures. Let us now have a look at the other way, that is, how to insert XML data using stored procedures.

XML data is usually available as TEXT data. That means that you would either talk about it using data type CHAR, VARCHAR, CLOB, or DBCLOB. That means that passing an XML document to a stored procedure through a parameter does not cause you any problems at all since it is typically not delivered to your application in XML data type format.

As a consequence you would pass your XML text string to your stored procedure using IN or INOUT parameters defined as:

- CHAR
- VARCHAR
- CLOB
- DBCLOB

As a result, your stored procedure definition, including the definition of IN parameters, could look as shown in Figure 25-8. In this trivial example, you can see that the IN parameter that is passing the XML document is defined as VARCHAR(2000). You could do the same thing by using any of the four data types described above. Refer to Example 25-13 if you would like to see the DDL that we used to create table DSN8910.PURCHASEORDER.

```
CREATE PROCEDURE XMLINS1
( IN PORDERIN  VARCHAR(2000),
  IN POIDIN    INTEGER,
  IN STATUSIN  CHAR(10)   )
 VERSION VERSION1
 ISOLATION LEVEL CS
 RESULT SETS 1
P1: BEGIN
       INSERT INTO DSN8910.PURCHASEORDER
       VALUES (POIDIN, STATUSIN, PORDERIN);
 END P1#
```

Figure 25-8 Simple stored procedures passing an XML document through the IN parameter

To run this stored procedure, we used Data Studio as the calling program. Refer to Chapter 27, “The IBM Data Studio” on page 643 if you want to learn more about what Data Studio is and how to use this new tool to execute stored procedures.

In Data Studio, if you open the server view on the lower left corner, identify the schema your stored procedure is defined in, click **Expand** to see all stored procedures of your schema, right-click your stored procedure and press Run, a little window pops up where you are asked to enter the values for your IN parameters, as shown in Figure 25-9.

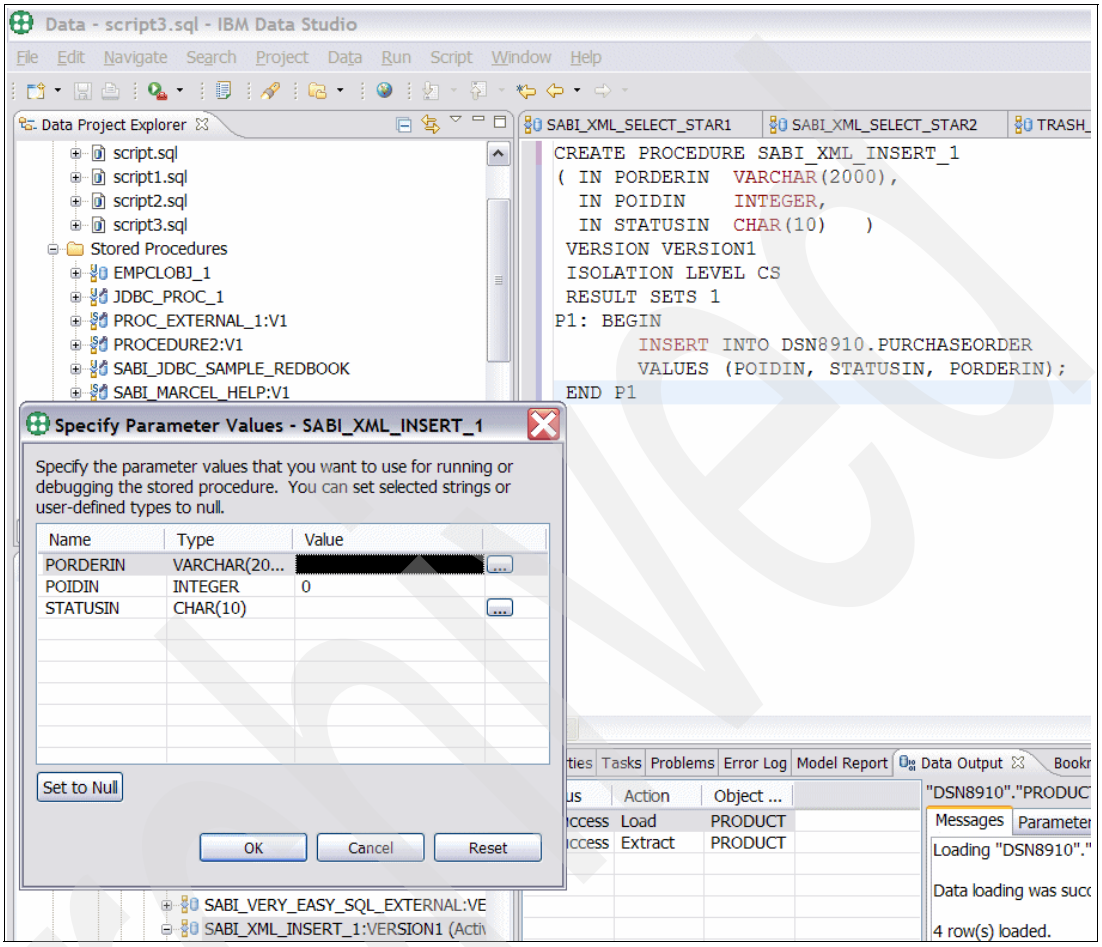

Figure 25-9 Parameter input panel of Data Studio

Let us assume that we enter the parameters as shown in Figure 25-10 and press **OK**. The parameters are then passed to the stored procedure and inserted into the specified columns. DB2 converts the XML document that has been passed in as VARCHAR text format to XML data type.

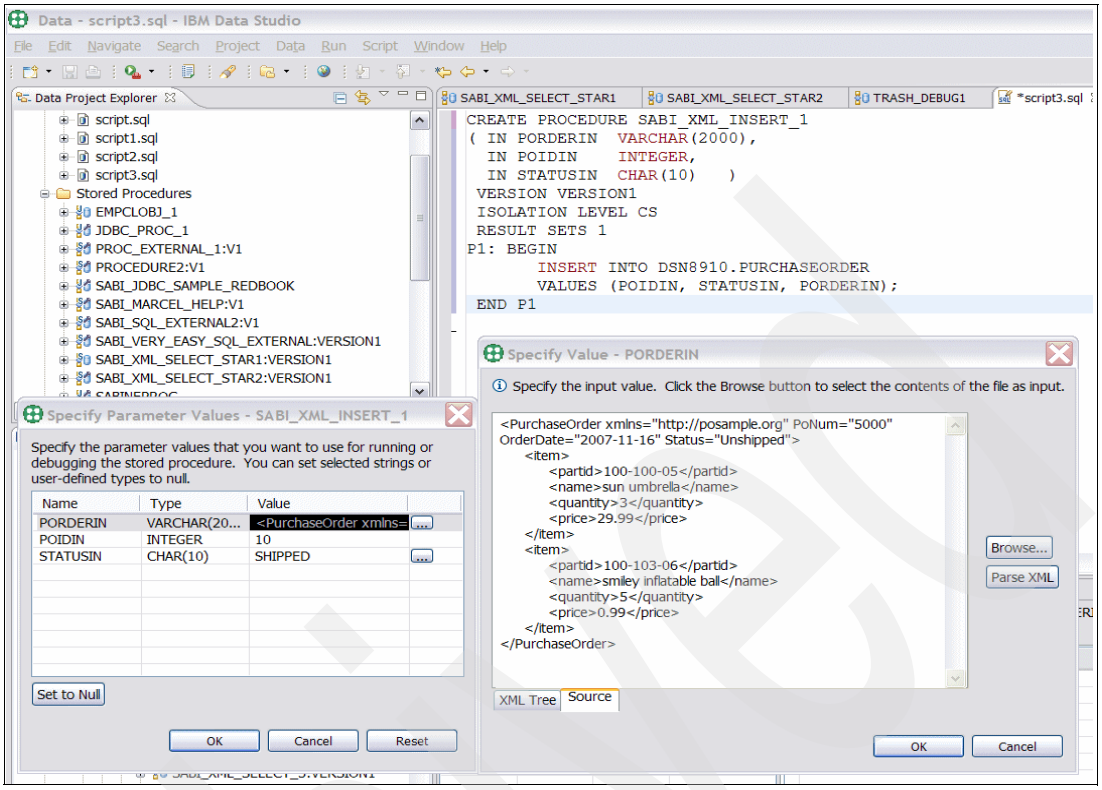

Figure 25-10 Specified values in Data Studio for IN parameters

25.6.4 Use of the XML data type in stored procedure result sets

Different from what you just read about the use of XML data type in OUT and INOUT parameters of a stored procedure, there is nothing you need to worry about or code differently than you would for any other data type when talking about result sets.

Let us take a look at Figure 25-11, which is very similar to what we showed you in Figure 25-2. Figure 25-11shows a CREATE statement for a native SQL stored procedure.

```
CREATE PROCEDURE DEVL7083.XMLSEL4()
 VERSION VERSION1
 ISOLATION LEVEL CS
 RESULT SETS 1
 LANGUAGE SQL
-------------------------------------------------------
-- NATIVE SQL STORED PROCEDURE
-------------------------------------------------------
P1: BEGIN
   DECLARE C1 CURSOR WITH RETURN FOR
      SELECT POID, PORDER FROM  DEVL7083.PURCHASEORDER2;
   OPEN C1;
END
```

Figure 25-11 Create procedure with cursor and result set

In contrast to the one in Figure 25-2, this procedure does not accept or return any parameters. Instead, we are generating a result set containing all the purchase-IDs and the XML documents stored in XML column PORDER. The creation of this stored procedure

works fine. Also, when we run this procedure using Data Studio again, we get back all rows that are currently stored in DEVL7083.PURCHASEORDER2, as shown in Figure 25-12.

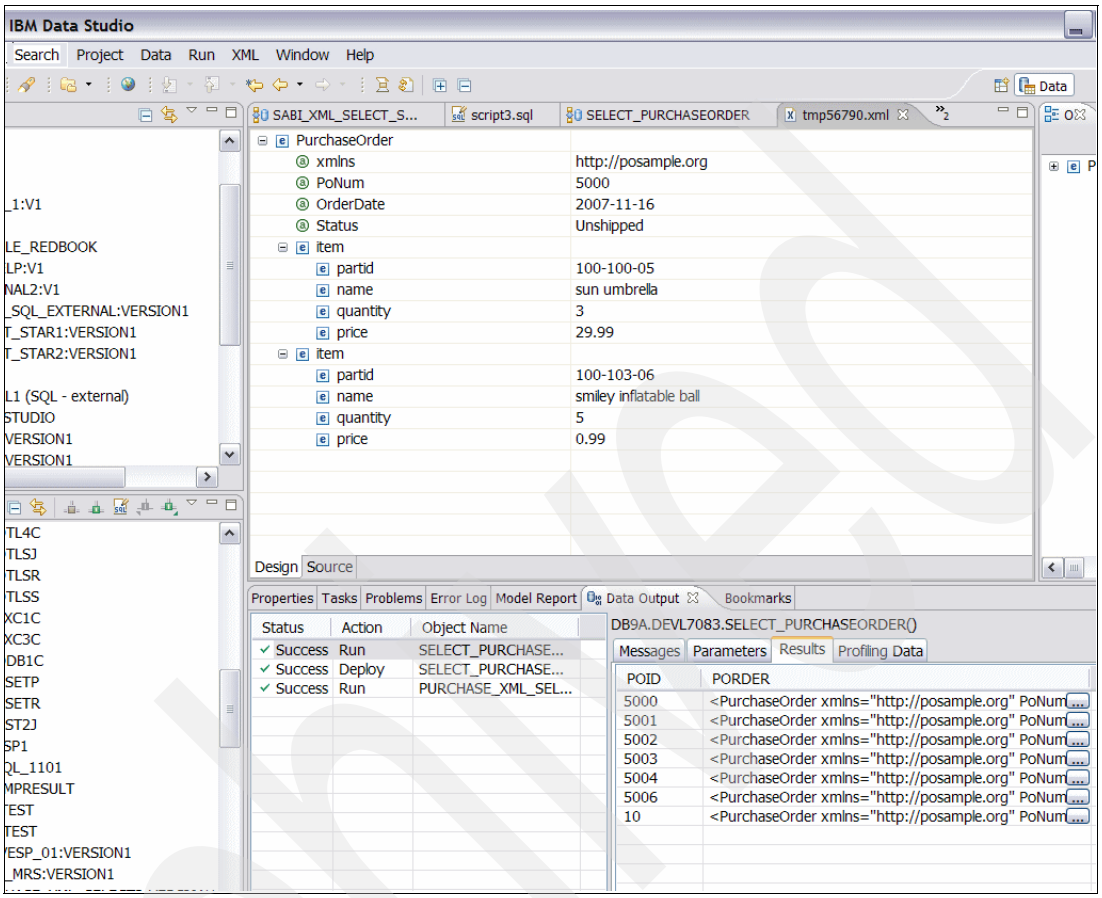

Figure 25-12 Data Studio output of returned result set

26

Using triggers and UDFs

In this chapter we discuss how you can extend the functionality of a trigger by invoking a stored procedure or a user-defined function. We discuss different methods of passing parameters and discuss the factors that influence which of these (stored procedure or user-defined function) should be deployed. We show how you can handle errors. Finally, a stored procedure and a user-defined function can call each other and we explore this option.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions can be found in Appendix B, “Additional material” on page 887.

Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 24 to decide what components are applicable to your environment.

This chapter contains the following:

- Introduction
- Passing parameters to a stored procedure
- Error handling in triggers
- Stored procedures versus user-defined functions
- Stored procedures calling user-defined functions
- user-defined functions calling stored procedures

26.1 Introduction

A trigger body can include only SQL statements and built-in functions. If you want the trigger to perform actions or use logic that is not available in SQL statements or built-in functions, you need to write a stored procedure or a user-defined function, and invoke it from the trigger body.

If you are not familiar with the definition of a trigger, refer to Chapter 12, “Using triggers for active data” in *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841 or Chapter 3, “Triggers” in *DB2 for z/OS Application Programming Topics*, SG24-6300 for details.

Example 26-1 shows how trigger EMPTRIG3 on table EMP invokes user-defined function EMPAUDTU once for each row whenever a salary increase of more than 10% occurs. The parameters passed are the employee number and the old and new salary. Notice that the user-defined function is invoked with a VALUES statement. Also note that the delimiter used at the end of the CREATE TRIGGER statement is a number sign (#) rather than an asterisk. You need to use the number sign because the trigger includes an SQL statement within the trigger action (also called the trigger body) and that SQL statement is delimited by an asterisk.

Example 26-1 Trigger invoking a UDF with a VALUES clause

```
CREATE TRIGGER DEVL7083.EMPTRIG3 NO CASCADE
BEFORE UPDATE OF SALARY ON DEVL7083.EMP
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
SELECT
  CASE
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '1' THEN COALESCE(RAISE_ERROR('75001','JUNK1'),' ')
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '2' THEN COALESCE(RAISE_ERROR('75002','JUNK2'),' ')
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '3' THEN COALESCE(RAISE_ERROR('75003','JUNK3'),' ')
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '4' THEN COALESCE(RAISE_ERROR('75004','JUNK4'),' ')
  END
FROM SYSIBM.SYSDUMMY1;
END#
```

Trigger EMPTRIG3 has access to the value of the SALARY column before and after the update by using the transition variables identified in the REFERENCING clause. The UDF EMPAUDTU will take the three variables supplied (EMPNO, old SALARY and new SALARY) and return a value of 1, 2, 3 or 4 depending on the percentage of the employee raise. The trigger will then use the CASE statement to return a SQLSTATE value of between 75001 and 75004 along with a respective literal of JUNK1 through JUNK4. We tested this trigger by issuing an UPDATE statement on the EMP table in IBM Data Studio. Figure 26-1 shows the SQLSTATE of 75002 and the message text of JUNK2 that is returned by EMPAUDTU. For more details on using IBM Data Studio to test DB2 applications, see Chapter 27, “The IBM Data Studio” on page 643.

```
Starting run

UPDATE DEVL7083.EMP
  SET SALARY = SALARY * 1.12
  WHERE EMPNO = '000070'

com.ibm.db2.jcc.b.SqlException: APPLICATION RAISED ERROR WITH DIAGNOSTIC TEXT: JUNK2.
SQLCODE=-438, SQLSTATE=75002, DRIVER=3.50.152
```

Figure 26-1 Messages tab in IBM Data Studio Data Output View showing trigger test results

An alternative way to conditionally invoke a user-defined function from a trigger, based on the number of rows of a table, is to issue a SELECT statement against the transition table. Transition tables are discussed in 26.2.2, “Using transition tables” on page 633.

Similarly, Example 26-2 shows a trigger that accomplishes the same result by calling a stored procedure instead of invoking a trigger. Stored procedure EMPAUDTS is invoked under the same conditions and is passed the same parameters. Since the trigger is invoking a stored procedure, a CALL statement is used in place of the VALUES statement that was used for a user-defined function.

Example 26-2 Trigger invoking a stored procedure with a CALL statement

```
CREATE TRIGGER DEVL7083.EMPTRIG1
AFTER UPDATE OF SALARY ON DEVL7083.EMP
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.10)
BEGIN ATOMIC
 CALL   DEVL7083.EMPAUDTS(N.EMPNO,O.SALARY,N.SALARY);
END#
```

Since trigger EMPTRIG1 is an AFTER trigger on an UPDATE statement, stored procedure EMPAUDTS has access to the before and after values of the SALARY column. The stored procedure displays the employee number and the before and after salary if the employee’s salary was updated by more than 10%. Figure 26-2 shows the data that is displayed in SDSF for the started task for the WLM address space.

```
++ START OF EMPAUDTS STARTING ++
WS-EMPNO = 000070
WS-OLDSALARY = 004172571
WS-NEWSALARY = 004673279
++ END OF EMPAUDTS ++
```

Figure 26-2 SDSF output showing DISPLAY results for stored procedure invoked by trigger

The parameters of a stored procedure call from a trigger must be literals, transition variables (see 26.2.1, “Using transition variables” on page 633), transition tables (see 26.2.2, “Using transition tables” on page 633), or expressions.

Note that a transition variable or transition table is not affected after being returned from a stored procedure invoked from a trigger. This is true regardless of how the corresponding parameter is defined in the CREATE PROCEDURE—IN, OUT or INOUT.

Also note that a user-defined function or stored procedure invoked from a trigger must be at the local server. These in turn can access DB2 objects at a remote server. In addition, when invoked by a BEFORE trigger, the stored procedure or user-defined function cannot refer to the subject table that causes the trigger to be fired.

As you will see from the discussion above, there is a substantial overlap in the functionality of user-defined functions and stored procedures when invoked by a trigger. However, you must consider the error handling requirements before making the decision. We discuss this in 26.4, “Stored procedures versus user-defined functions” on page 636.

26.2 Passing parameters to a stored procedure

The triggered action (stored procedure or user-defined function) can refer to the values in the set of affected rows. This is supported through the use of transition variables and transition tables. Transition variables refer to the values of a single row, and this is discussed in 26.2.1, “Using transition variables” on page 633. Transition tables refer to the complete set of values of all affected rows, and this is discussed in 26.2.2, “Using transition tables” on page 633.

Table 26-1 summarizes the allowable combinations of transition variables and transition tables that you can specify for the various trigger types. The MERGE statement, introduced in DB2 9 for z/OS, performs an UPDATE if the row exists and an INSERT if the row does not exist. Any INSERT or UPDATE triggers that are defined on the table are fired depending on what operation actually gets executed, so we do not provide any distinction in the following table for the MERGE operation. INSTEAD OF triggers, also introduced in DB2 9 for z/OS, result in an INSERT, UPDATE or DELETE operation on a base table when one of those operations is requested on a view. INSTEAD OF triggers, as well as BEFORE triggers, must be defined with a granularity of FOR EACH ROW.

Table 26-1 Allowable combination of attributes in a trigger definition

Granularity	Activation time	triggering SQL operation	Transition variables allowed	Transition tables allowed
FOR EACH ROW	BEFORE	INSERT	NEW	None
		UPDATE	OLD,NEW	None
		DELETE	OLD	None
	AFTER	INSERT	NEW	NEW TABLE
		UPDATE	OLD,NEW	OLD TABLE, NEW TABLE
		DELETE	OLD	OLD TABLE
	INSTEAD OF	INSERT	NEW	NEW TABLE
		UPDATE	OLD,NEW	OLD TABLE, NEW TABLE
		DELETE	OLD	NEW TABLE
FOR EACH STATEMENT	AFTER	INSERT	None	NEW TABLE
		UPDATE	None	OLD TABLE, NEW TABLE

Granularity	Activation time	triggering SQL operation	Transition variables allowed	Transition tables allowed
		DELETE	None	OLD TABLE

26.2.1 Using transition variables

Transition variables are only applicable for triggers whose granularity is FOR EACH ROW, meaning that the trigger is fired once for each row that is inserted, updated or deleted. Transition variables are allowed in components of a trigger definition in the same way that host variables are allowed in an SQL statement outside the body of a trigger. Transition variables can be referenced in the trigger action condition (the WHEN clause) and in the trigger action (also called the trigger body). They use the names of the columns in the subject table qualified by a specific name that identifies whether the reference is to the old value (before the INSERT, UPDATE or DELETE) or to the new value (after the INSERT, UPDATE or DELETE). In Example 26-3 trigger EMPTRIG1 calls stored procedure EMPAUDTS and uses `O` and `N` as the correlation names to designate the before and after (old and new) values for the transition variables.

Example 26-3 Trigger invoking a stored procedure with transition variables

```
CREATE TRIGGER DEVL7083.EMPTRIG1
AFTER UPDATE OF SALARY ON DEVL7083.EMP
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.10)
BEGIN ATOMIC
 CALL   DEVL7083.EMPAUDTS(N.EMPNO,O.SALARY,N.SALARY);
END#
```

The list of parameters must be compatible with the parameter list defined in the linkage section of the stored procedure and the PROCEDURE DIVISION USING... statement. For the sample stored procedure EMPAUDTS, the linkage section looks like this in COBOL:

```
LINKAGE SECTION.
01  PEMPNO        PIC X(6).
01  POLDSALARY    PIC S9(7)V9(2) COMP-3.
01  PNEWSALARY    PIC S9(7)V9(2) COMP-3.
```

The procedure division for the stored procedure looks like this in COBOL:

```
PROCEDURE DIVISION USING PEMPNO, POLDSALARY, PNEWSALARY.
```

26.2.2 Using transition tables

Transition tables are only applicable for AFTER triggers and INSTEAD OF triggers, because they allow you to access the state of the affected rows before and after the execution of the triggering action. Transition tables are allowed in components of a trigger definition in the same way that table names are allowed in SQL statements that are not associated with a trigger. The name of the transition table can be referenced in the trigger action (also called the trigger body).

Transition tables are read-only. Like transition variables, transition tables also use the names of the columns of the subject table, but the old and new transition tables each have a qualifier specified that allows the complete set of affected rows to be treated as a table.

In Example 26-4 trigger EMPTRIG2 calls stored procedure EMPAUDTX using OT and NT as the qualifiers to designate the before and after (old and new) transition table values.

Example 26-4 Trigger invoking a stored procedure with transition tables

```
CREATE TRIGGER DEVL7083.EMPTRIG2
AFTER  UPDATE OF SALARY ON DEVL7083.EMP
REFERENCING OLD TABLE AS OT
            NEW TABLE AS NT
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
 CALL   DEVL7083.EMPAUDTX(TABLE OT,TABLE NT);
END#
```

We now describe how to access transition tables in a stored procedure, but the same applies to a user-defined function.

To access transition tables in a stored procedure, use table locators, which are pointers to the transition tables. You declare table locators as input parameters in the CREATE PROCEDURE statement using the TABLE LIKE *table-name* AS LOCATOR clause. See Chapter 5 of *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854 for more information.

The five basic steps to accessing transition tables in a stored procedure are:

1. Declare input parameters to receive table locators. You must define each parameter that receives a table locator as an unsigned 4-byte integer. This is shown in Example 26-5 for COBOL. This step is optional and it is required only if you plan to use the locator later in the program and need to save it. In general, for COBOL you can use the locators from the LINKAGE SECTION directly.

   Example 26-5 Declaring input variables for table locators

   ```
   01 WS-TRIG-TBL-ID-OLD SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
   ```

2. Declare table locators. The syntax varies with the application language. See Chapter 9, “Embedding SQL statements in host languages” of the *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841 for information on the syntax for C, C++, COBOL, and PL/I. See Chapter 6 of *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854 for information on the syntax for SQL procedures. This is shown in Example 26-6.

   Example 26-6 Declaring table locators

   ```
   LINKAGE SECTION.
   01  TRIG-TBL-ID-OLD SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
   01  TRIG-TBL-ID-NEW SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
   ```

3. Declare a cursor to access the rows in each transition table. This is shown in Example 26-7.

   Example 26-7 Declaring a cursor

   ```
   ****      CURSOR FOR RETRIEVING "BEFORE" AND "AFTER" IMAGES
             EXEC SQL DECLARE C1
               CURSOR FOR
               SELECT
                   OLDTAB.EMPNO
                 , OLDTAB.SALARY
                 , NEWTAB.SALARY
   ```

```
            FROM TABLE(:TRIG-TBL-ID-OLD LIKE EMP) AS OLDTAB
               , TABLE(:TRIG-TBL-ID-NEW LIKE EMP) AS NEWTAB
            ORDER BY EMPNO
          END-EXEC.
```

4. Assign the input parameter values to the table locators. This is shown in Example 26-8.

Example 26-8 Setting values of table locators

```
PROCEDURE DIVISION USING TRIG-TBL-ID-OLD, TRIG-TBL-ID-NEW.
```

5. Access rows from the transition tables using the cursors that are declared for the transition tables. This is shown in Example 26-9.

Example 26-9 Accessing the transition tables

```
EXEC SQL
   OPEN  C1
END-EXEC.
...
EXEC SQL
  FETCH C1
  INTO  :WS-EMPNO
      , :WS-OLDSALARY
      , :WS-NEWSALARY
END-EXEC.
...
EXEC SQL
  CLOSE C1
END-EXEC.
```

Now let's return to our transition table example from Example 26-4 on page 634. Since trigger EMPTRIG2 is an AFTER trigger on an UPDATE statement, stored procedure EMPAUDTX has access to the before and after copies of the row being updated by using the transition tables. Stored procedure EMPAUDTX fetches the data from the before and after transition tables and displays the old and new salary, as well as the SQLCODE returned from the FETCH from each transition table if the employee's salary was updated by more than 10%. The displayed output is shown in Figure 26-3.

```
++ START OF EMPAUDTX STARTING ++
++ SQLCODE AFTER OPEN   = 000000000
++ SQLCODE AFTER FETCH  = 000000000
WS-EMPNO = 000070
OLD SLARY = 004673279
NEW SLARY = 005234072
++ SQLCODE AFTER FETCH  = 000000100
++ SQLCODE AFTER CLOSE  = 000000000
++ END OF EMPAUDTX ++
```

Figure 26-3 SDSF output showing DISPLAY results for stored procedure with transition tables

26.3 Error handling in triggers

Severe errors that occur during the execution of a triggered SQL statement are returned with SQLCODE -901, -906, -911 and -913 (along with the corresponding SQLSTATEs).
Non-severe errors raised by a triggered SQL statement through the SIGNAL SQLSTATE statement or an SQL statement containing a RAISE_ERROR function are returned with the

specified SQLSTATE, and the SQLCODE is always -438. Other non-severe errors are returned with an SQLCODE -723 and SQLSTATE 09000. Warnings are not returned.

Another option to cause errors in a stored procedure to return an error to the trigger is to issue a ROLLBACK in the stored procedure when you encounter an error. It is incorrect syntax to issue a COMMIT or a ROLLBACK in a stored procedure that is called by a trigger. This is stated in Chapter 5 of *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854. Since a ROLLBACK is not allowed in a stored procedure that is called by a trigger, the ROLLBACK will receive an SQLCODE of -751 and, as a result of that error, an SQLCODE of -723 will be returned for the original SQL statement that caused the trigger to be invoked. Therefore, the original SQL statement will fail and none of the changes will be committed. In a way the ROLLBACK in the stored procedure works, just not directly.

If your stored procedure is written in Java, then you could throw a Java exception which would cause an exit from the Java program. This in turn will cause the trigger to fail, which will cause an error SQLCODE of -723 to be returned for the original SQL statement that caused the trigger to be invoked.

The ability to handle errors in a trigger is severely limited, especially when it calls a stored procedure, as we have shown above. For this reason, you must decide carefully whether you use a stored procedure or a user-defined function. This is discussed next in 26.4, “Stored procedures versus user-defined functions” on page 636.

26.4 Stored procedures versus user-defined functions

There are two common uses of triggers:

- Data validation
- Data propagation

Data validation deals with invoking complex business logic to determine whether or not a certain action (INSERT, UPDATE, or DELETE) should be permitted. This is typically achieved with a user-defined function invoked by a BEFORE trigger. Data propagation deals with invoking complex business logic after a certain action has taken place. This is typically achieved with a stored procedure invoked by an AFTER trigger. Figure 26-4 shows how a user-defined function can be used for data validation.

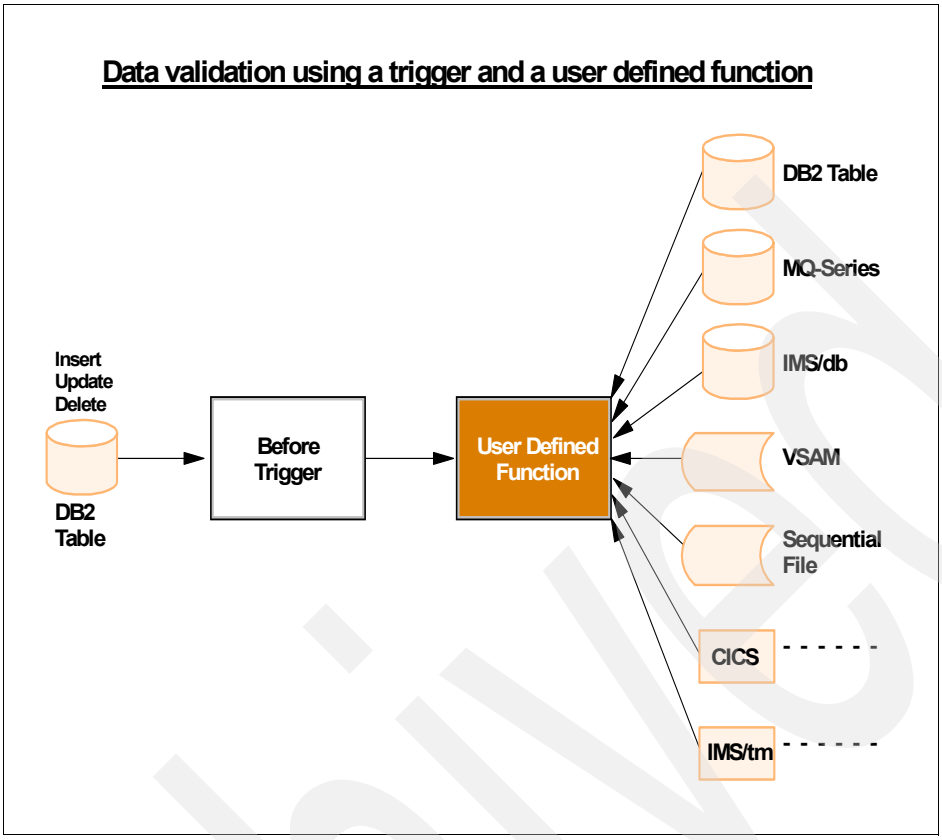

Figure 26-4 Data validation using a trigger and a user-defined function

Example 26-10 shows how a user-defined function can pass an output parameter back to the trigger. In this case, our sample user-defined function EMPAUDTU is passed three input parameters and sets the PRTNCD parameter and passes it back to the trigger.

Example 26-10 Setting parameters in a user-defined function

```
PROCEDURE DIVISION USING PEMPNO, POLDSALARY, PNEWSALARY,
                         PRTNCD.
....
**** ------------------------------------------------------
**** WE WILL VALIDATE THE RAISE PERCENT AND ACT AS FOLLOWS:
****   <   5% - NO ERROR, JUST PROCESS, PERHAPS LOG
****   >=  5% AND <10%  - SET PRTNCD TO 1
****   >= 10% AND <15%  - SET PRTNCD TO 2
****   >= 15% AND <20%  - SET PRTNCD TO 3
****   >= 20%           - SET PRTNCD TO 4
**** ------------------------------------------------------

           COMPUTE WS-RAISE-PCT
             = (WS-NEWSALARY - WS-OLDSALARY)* 100.00 /
                WS-OLDSALARY.

           DISPLAY 'RAISE = ' WS-RAISE-PCT.

           EVALUATE TRUE
               WHEN WS-RAISE-PCT < 5.00
                    MOVE SPACES TO PRTNCD
               WHEN WS-RAISE-PCT < 10.00
                    MOVE '1' TO PRTNCD
```

```
            WHEN WS-RAISE-PCT < 15.00
                 MOVE '2' TO PRTNCD
            WHEN WS-RAISE-PCT < 20.00
                 MOVE '3' TO PRTNCD
            WHEN OTHER
                 MOVE '4' TO PRTNCD
       END-EVALUATE.
```

Example 26-11 shows how you can use the result of a user-defined function to return different error messages to the calling application.

Example 26-11 Generating error messages in a trigger that invokes a UDF

```
CREATE TRIGGER DEVL7083.EMPTRIG2 NO CASCADE
BEFORE UPDATE OF SALARY ON DEVL7083.EMP
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
SELECT
  CASE
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '1' THEN COALESCE(RAISE_ERROR('75001','some message1'),' ')
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '2' THEN COALESCE(RAISE_ERROR('75002','some message2'),' ')
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '3' THEN COALESCE(RAISE_ERROR('75003','some message3'),' ')
      WHEN DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '4' THEN COALESCE(RAISE_ERROR('75004','some message4'),' ')
  END
FROM SYSIBM.SYSDUMMY1;
END#
```

The user-defined function DEVL7083.EMPAUDTU returns a value to the trigger, then the CASE statement within the trigger body determines whether the built-in function RAISE_ERROR is invoked to return a SQLSTATE value. The COALESCE function is used because the CREATE TRIGGER statement will receive a syntax error on the RAISE_ERROR function due to the possibility that the function will return a null value. The SQLSTATE value must conform to the following rules:

- Each character must be numeric (0 through 9) or uppercase alphabetic (A through Z).
- The SQLSTATE class (first two characters) cannot be 00, 01, or 02 because these are not error classes.
- If the SQLSTATE class starts with 0 through 6, or A through H, then the subclass (last three characters) must start with a letter in the range I through Z.
- If the SLQSTATE class starts with 7 through 9, or I through Z, then the subclass (last three characters) can be any of 0 through 9, or A through Z.

Figure 26-5 shows how a stored procedure can be used for data propagation.

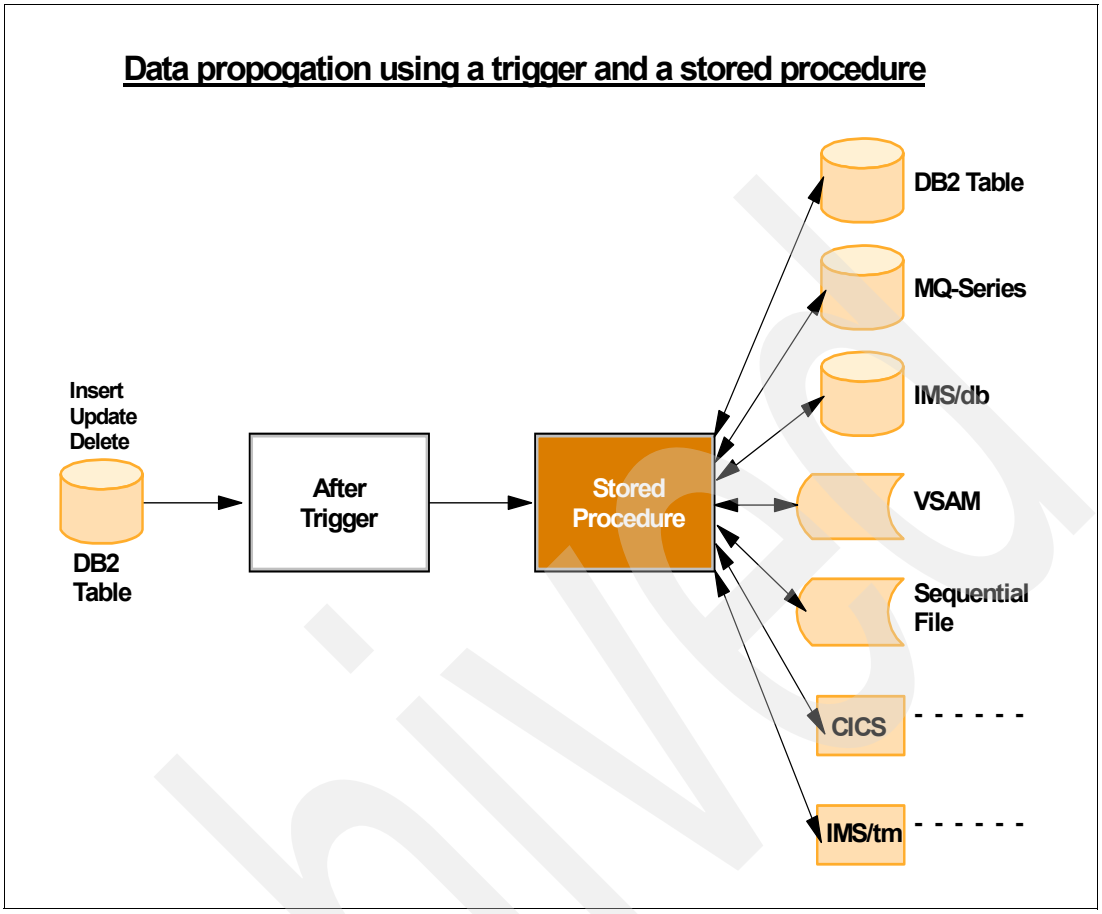

Figure 26-5 Data propagation using a trigger and a stored procedure

The stored procedure could take information available in either transition variables or a transition table and use that data to update another DB2 table, write to an MQ queue or propagate any of the other object types described in Figure 26-5.

26.5 Stored procedures calling user-defined functions

A stored procedure can call other programs, stored procedures, or user-defined functions. If the stored procedure calls other programs that contain SQL statements, each of those called programs must have a DB2 package. The owner of the package or plan that performs the call must have EXECUTE authority for the packages that are being called.

Invoking a user-defined function from a stored procedure is identical to how you invoke it from any other application. There are no special considerations that apply.

Invocation of a user-defined function or a stored procedure within one logical unit of work is considered as “nesting” (subject to the limit of 16); this is discussed in 10.3.1, “Nested stored procedures” on page 130.

26.6 user-defined functions calling stored procedures

Calling a stored procedure from a user-defined function is identical to how you call it from any other application. There are no special considerations that apply.

As before, invocation of a stored procedure from a user-defined function within one logical unit of work is considered as “nesting” (subject to the limit of 16); this is discussed in 10.3.1, “Nested stored procedures” on page 130.

Part 6

Cool tools for an easier life

In this part we discuss topics related to handling stored procedures across different platforms. Several tools are available for coding and debugging stored procedures. We concentrate on how to use these functions for deployment of stored procedures on z/OS, but most considerations apply to the other members of the DB2 family.

This part contains the following chapters:

27

The IBM Data Studio

In DB2 V7.2 for Linux, UNIX, and Windows, IBM introduced tooling support for stored procedures via the Stored Procedure Builder. IBM enhanced this tooling with the follow-on tool, Development Center, in DB2 V8.1 for Linux, UNIX, and Windows. With DB2 9 for Linux, UNIX, and Windows, the Developer Workbench was introduced. Developer Workbench is based on Eclipse technology. The stored procedure tooling for DB2 databases is also consistent with the tooling delivered in WebSphere Application Developer and Rational Application Developer. On October 30, 2007, IBM announced the IBM Data Studio, which builds upon the tooling support of Developer Workbench and other IBM tooling products as well.

As of the publication of this book, there are three “versions” of IBM Data Studio, shown in Table 27-1.

Table 27-1 Data Studio products

Product	PID
Data Studio V1.1.0 (free version)	Uses DB2 on LUW PID
Data Studio Developer V1.1.1(charged version)	5724-U15
Data Studio pureQuery runtime V1.1.1 (charged)	5724-U16

The IBM Data Studio is a comprehensive data management solution that empowers you to effectively design, develop, deploy and manage your data, databases and database applications throughout the entire application development life cycle utilizing a consistent and integrated user interface. Included in this tooling suite are the tools for developing and deploying DB2 for z/OS stored procedures. Unlike Development Center (DC), which was included in the DB2 V8.1 UDB Application Development Client (ADC) component, the IBM Data Studio is independent of any other product offering and does not require a DB2 client installed.

IBM Data Studio supports the entire family of DB2 servers, as well as Informix, using the DRDA architecture. It supports Versions 8 and 9 of DB2 for Linux, UNIX and Windows; Versions 7, 8 and 9 of DB2 for z/OS; Versions 5.3 and up of DB2 for iSeries; and Informix

Data Servers. The suite of servers and functions that IBM Data Studio supports are summarized in Figure 27-1. For more information about IBM Data Studio, visit:

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

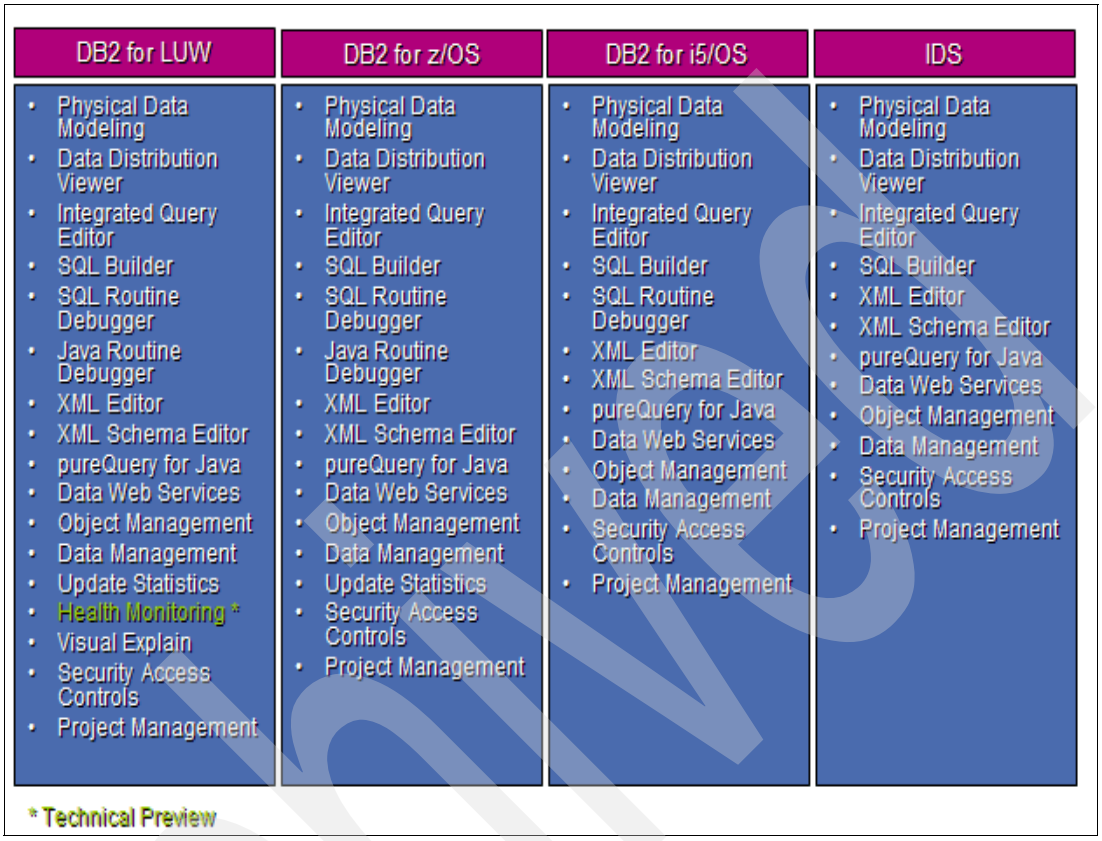

Figure 27-1 IBM Data Studio V1.1 Support features

While IBM Data Studio is a tooling suite that delivers features that go beyond stored procedure development, in this chapter we limit our discussions to the routine tooling feature for creating, developing and deploying SQL and Java stored procedures. Additional features in support of stored procedures are mentioned at the end of this chapter.

IBM Data Studio and Rational Application Developer both sit on top of the Eclipse Framework. Thus, both products have a similar “look and feel”. The setup for z/OS, which we describe in this chapter for creating SQL and Java stored procedures for IBM Data Studio, applies to Rational Application Developer as well. The same setup applies when using .NET to create SQL stored procedures on z/OS.

In 3.3, “Sample application components” on page 24, we discussed the sample SQL and Java stored procedures on DB9A, a DB2 for z/OS V9 server. We create similar stored procedures using IBM Data Studio in this section.

IBM strongly recommends using the IBM Data Server driver for JDBC and SQLJ, also known as the Universal JDBC driver, when connecting to the DB2 servers. While IBM Data Studio provides the ability to connect using the Legacy driver, the support for this is very limited. In this chapter, we assume that the IBM Data Server driver for JDBC and SQLJ is used throughout. For information on using the Legacy Driver with DB2 Development Center, refer to section 29.2.5 in the previous edition of this book.

This chapter contains the following:

- Eclipse workbench common terminology
- Prerequisites and setup steps
- Navigating through the IBM Data Studio workspace
- Developing stored procedures with IBM Data Studio
- Developing stored procedures
- Deploying a stored procedure
- Advanced IBM Data Studio topics

27.1 Eclipse workbench common terminology

In this section we introduce the basic Data Studio concepts and terminology.

Data Studio is based on the open and extensible framework of the Eclipse Workbench.

The Eclipse Workbench is an Integrated Development Environment (IDE). It is the major delivery of a consortium of companies, called *eclipse.org*, and is an open source development platform. This consortium was initiated by IBM. Eclipse is currently the most successful open source project judging by the number of contributing participants. Eclipse defines extensibility interfaces and integration points, so other projects can contribute, extend and/or consume "pieces" of the code, called *plugins*. Several specialized projects extend the basic IDE, such as Web Tools Project – WTP, Data Tools Project – DTP, etc.

The Eclipse Workbench, or Workbench for short, consists of:

- Workspace
- Resources
- Perspectives
- Views
- Editors
- Wizards

Workspace

When you open the workbench, you are asked to choose a workspace. All your resources and settings are saved in this workspace. Only one workspace is active at any given time. You may open a different workspace each time you open the workbench. You can also switch workspaces by clicking **File → Switch Workspace.**

Resources

A resource is a collective term for the projects, folders, and files that you created in a workspace. Typically, resources are viewed in a hierarchical format and can be opened for editing. There are three basic types of resources that exist in the Workbench:

Files

A file in Eclipse is comparable to files in the workstation. Each resource in Eclipse is associated with a file. Eclipse persists or saves all resources in the workspace as files in the file system.

Folders

Folders in the Eclipse workspace are comparable to directories in a file system. In the workspace, folders are containers for the various resources that can be created, viewed and/or manipulated by the tooling.

Projects

In Eclipse, development is contained in projects. Projects contain folders which in turn can contain either a set of objects or another folder. Projects have one or more nature associated with it, meaning that the contents and tasks that can be done on the objects within the project depend on the kind of project created.

A project is either open or closed. When a project is closed, it cannot be changed in the Workbench. The resources of a closed project will not appear in the Workbench, but the resources still reside on the local file system. When a project is open, the structure and contents of the project can be viewed and modified.

Perspectives

A perspective is a group of views and editors in the Workbench window. One or more perspectives can exist in a single Workbench window. Each perspective contains one or more views and editors. Each perspective may have a different set of views but all perspectives share the same set of editors.

Data perspective

The Data perspective is the primary perspective used by IBM Data Studio. The Data perspective provides a set of functions and specialized views for displaying, creating, deploying and managing database objects.

Debug perspective

The Debug perspective is the primary perspective used by IBM Data Studio when debugging an SQL or Java stored procedure. This perspective is used by other products, such as the Rational Developer V7 for System Z.

Other perspectives used by IBM Data Studio are the Java Perspective, the Team Perspective and the Resource Perspective.

Views

A view is a visual component within the Workbench that is used to display a hierarchy of resources in the Workbench, display properties of a resource, and perform tasks on the resource. Modifications made in a view are saved immediately. Only one instance of a particular type of view may exist within a Workbench window.

In the Data Perspective you can:

- Open or show a view.
- Move a view to a different area of the workspace.
- Reset views.
- Minimize or Maximize a view.

Several views can share an area of the workspace as in the case when multiple objects are opened in the Editor. The Output view also shows multiple types of output (e.g. Error Log, Problems, Data Output, etc.)

For most tasks, a database developer will use the Database Explorer, Data Project Explorer, and the Data Output views. More information about these views in IBM Data Studio is given in 27.3, “Navigating through the IBM Data Studio workspace” on page 667.

Editors

An editor is a visual component within the Workbench that is used to edit or browse a resource. Modifications made in the editor follow an open-save-close lifecycle model. An editor can be specialized for a function. Multiple instances of specialized editors may exist within a Workbench window.

Wizards

A wizard is a visual component within the Workbench that is used to step a user through a series of tasks related to a resource. The purpose of the wizard is to make a task easy for the user.

27.2 Prerequisites and setup steps

In this section we describe the prerequisites and setup steps for both the client and the z/OS Server.

- Client setup
- DB2 for z/OS setup
- Unicode support
- Setup for SQL and Java stored procedures
- IBM Data Studio Actual Costs setup
- IBM Data Studio and JDBC driver selection
- Java SDKs used by IBM Data Studio

27.2.1 Client setup

The client setup is made up of the following steps:

1. Review prerequisites
2. Install IBM Data Studio
3. Optionally, install DB2 Connect
4. Bind JDBC packages

1 - Review prerequisites

The following lists the minimum prerequisites:

- IBM Data Studio installation software or CD.

  Unlike IBM Development Center, the Developer Workbench and IBM Data Studio are separate installable products. Both are available as part of DB2 on LUW V9. IBM Data Studio is also available as part of the DB2 Accessories Suite for z/OS, V1.2 (5655-R14). The free version of IBM Data Studio is available as a download from the following Web site:

  `https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=swg-ids`

- Optionally, DB2 9 Connect or DB2 V8.1 Connect Enterprise Server Edition can be installed on the client workstation.

  A restricted use license of DB2 Connect Personal Edition for development purposes can be obtained in the DB2 for z/OS Management Clients Package. See the section on setup at “Workstation” on page 761 to obtain the FMIDs for the Management Clients Package.

- If you want to prototype your SQL or Java Stored procedures for DB2 on your client workstation, you need to install any edition DB2 V9.1 or DB2 V9.5 for LUW on your workstation.

2 - Install IBM Data Studio

After downloading and unzipping the IBM Data Studio, follow the steps below to install it. If you are using a DVD, replace the install drive character (e.g. “C:”) with the driver character for your DVD.

The downloaded image has the following layout:

1. Go to the <drive>:\windows\InstallerImage_win32\install.exe.
2. Double-click the install.exe file.
3. The IBM Installation Manager wizard is launched. Click the check boxes for *IBM Data Studio* and the *Version x.x.x*. Click **Next**.
4. Select the radio button for *“I accept the terms in the license agreement”*. Click **Next**.

5. In the Shared Resourced Directory, select **C:\Program Files\IBM\SDP70Shared**. Click **Next**.
6. In the Installation Directory text box, select **C:\Program Files\IBM\SDP70**. Click **Next**.
7. Do not extend the Eclipse IDE. Click Next.
8. The default setting is English. Select your language, then click **Next**.
9. A default list of features are preselected for you. Deselect or select features that you want installed. Click **Next**.
10. Verify that the disk space you have can accommodate the installation size. Click **Install**.
11. Check for Success, then click **Finish**.

Note: IBM Data Studio Developer V1.1.1 can share a package group with other compatible products that have been installed with IBM Installation Manager. So in Step 7 above, you can opt to extend the Eclipse Integrated Development Environment (IDE) you currently have installed in your system. The following products also include the Eclipse IDE in their installation. You can significantly reduce the storage requirements if you opt to "shell share" or share the base Eclipse plugins of the following products with IBM Data Studio.

The following products have been tested to successfully share the Eclipse package with IBM Data Studio Developer Version 1.1.1. Other products have not been tested and you should not attempt to install IBM Data Studio Developer Version 1.1.1 with them in a shared package group.

- Rational Data Architect Version 7.0.0.4
- Rational Application Developer Version 7.0.0.5
- Rational Software Architect Version 7.0.0.5

IBM Data Studio is installed into two default directories C:\Program Files\IBM\SDP70Shared and C:\Program Files\IBM\SDP70.

3 - Install DB2 Connect

This step is optional. If you are using IBM Data Studio for development purposes, then you do not need to have DB2 Connect installed.

IBM Data Studio ships with the IBM Data Server driver for JDBC and SQLJ[1], also called the Universal driver. These include the license jars, db2jcc.jar and db2jcc_license_cisuz.jar, needed to connect to DB2 for z/OS servers. However, these jars only work with IBM Data Studio. When you connect to DB2 for z/OS, the tooling uses these default license jars.

You can opt to use the license jars supplied by DB2 Connect. However, certain features and functions in IBM Data Studio require a specific level of db2jcc.jar. So, you need to verify if the version of the DB2 Connect license jars is equal to or greater than the version shipped with IBM Data Studio. To verify this, type the command shown in Example 27-1 on a Windows command prompt where classpath is the directory where db2jcc.jar is located.

Example 27-1 How to verify the JCC version

```
java -cp <classpath> com.ibm.db2.jcc.DB2Driver -version
```

4 - Bind JDBC packages

IBM Data Studio uses the IBM Universal Driver to connect to the server. The JDBC packages in this driver need to be bound to the server. The DB2Binder utility performs this task. This

[1] Also called, the IBM Universal Driver for JDBC and SQLJ

task needs to be done only once per server per collection ID. The DB2Binder utility can be executed from the server side or the client side. Example 27-2 shows the DB2Binder command executed from Windows.

Example 27-2 Connecting and binding DC

```
set CLASSPATH=C:\Program
Files\IBM\SDP70Shared\plugins\com.ibm.datatools.db2_1.0.100.v200709182330\driver\db2jcc.jar
;C:\Program
Files\IBM\SDP70Shared\plugins\com.ibm.datatools.db2_1.0.100.v200709182330\driver\db2jcc_lic
ense_cisuz.jarc;%$CLASSPATH%
java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A -user paolor5
-password pup4sale -collection DEVL7083
```

In the example, `-url` points to the domain:port//location of the DB2 for z/OS server that you want to connect to. The `user` is the TSO logon ID and `password` is the TSO logon ID password. The bind may need to be repeated after applying a FixPak to IBM Data Studio. If the DB2Binder is not run, and you do not re-execute the bind, you receive `-805` at the workstation when trying to use IBM Data Studio.

Note: Both the location and collection IDs should be in uppercase when submitting the DB2Binder command from the client. To continue typing a long line to the next line, type a backslash (\), followed by a space, then continue typing on the next line.

27.2.2 DB2 for z/OS setup

Next, we now show the steps for the DB2 for z/OS[2] setup. The DB2 for z/OS V9 APARS listed in Table 27-2 need to be applied to the system before using IBM Data Studio

Table 27-2 APARs for IBM Data Studio

Language	Component supported	DB2 for z/OS Version	APAR
Java	SQLJ		PQ95544 (support for long package names)
Java and SQL	Routine Tooling		**PK52533** (SQLCode113 is issued when calling a procedure with “&” in schema) **PK45372** (Incorrect error SQLCode and SQLState when creating SQLPL procedures with VALIDATE(RUN) option enabled **PK50369** (Provide a Visual Explain stored procedure for DB2 Data Studio 1.1), install with V9 Visual Explain PTF
SQL	DB2 V8 NFM and DSNTPSMP 1.21 or higher		**PK49647** (Exploit current schema for sql procedure builds by DSNTPSMP)
SQL	Unified Debugger		**PK41138** (Unified Debugger V9 FP2 on DB2 for z/OS V8) **PK41833** (Unable to activate stored procedure debugger if DB2 is configured with CMTSTAT=INACTIVE)

[2] IBM Data Studio supports connections to DB2 for z/OS V7, V8 and V9. This book focuses on the setup for DB2 for z/OS V9. See *DB2 for z/OS Stored Procedures, Through the Call and Beyond,* SG24-7083 for specific APARs for V7 and V8.

The minimum prerequisites are listed next.

- SQL and Java stored procedures:
  - DB2 for z/OS, minimum Version 8
  - LE
  - WLM
  - RRS
  - REXX language
  - Unicode support
- External SQL stored procedures
  - C compiler
- Java stored procedures
  - JDBC driver
  - SDK 1.4.2

IBM Data Studio interacts with the DB2 for z/OS subsystems using several DB2-supplied stored procedures. These stored procedures are defined using different customization jobs included in <hlq>.SDSNSAMP.

- The following customization jobs are required for both SQL and Java stored procedures:
  - DSNTIJSD
  - DSNTIJRX
  - DSNTIJTM
  - DSNTIJMS
  - DSNTEJ6W
  - DSNTIJSG
  - DSNTIJCC
- An Explain-like stored procedure can be set up that is optional on the SQL Statement page with both the SQL and Java wizards used by IBM Data Studio.
  - SYSPROC.DSNWSPM (Actual Costs) setup is currently a manual job submission.

IBM Data Studio authorization setup

This section describes general authorities and privileges needed for IBM Data Studio tasks when there are no secondary authorization IDs being used. Table 27-3 summarizes them.

Table 27-3 General authorities and privileges for all platforms using IBM Data Studio

Task	Authorities and privileges
Access target databases	CONNECT

Task	Authorities and privileges
Register stored procedures with a database server	CREATE PROCEDURE Also requires one of the following privileges: ► SYSADM or DBADM ► CREATEIN for the schema if the schema name of the stored procedure refers to an existing schema. ► IMPLICIT_SCHEMA authority on the database if the implicit or explicit schema name of the stored procedure does not exist. IMPLICIT_SCHEMA allows you to implicitly create an object with a CREATE statement and specifying a schema name that does not already exist. SYSIBM becomes the owner of the implicitly created schema and PUBLIC is given the privilege to create objects in this schema. ► CREATE IN privilege on desired collection id
Retrieve rows from a table or view.	SELECT
Create a view on a table	SELECT
Run the EXPORT utility	SELECT
Insert an entry in a table or view, and run the IMPORT utility.	UPDATE
Change an entry in a table, a view, or one or more specific columns in a table or view.	UPDATE
Delete rows from a table or view.	DELETE
To use the IBM Data Studio Unified Debugger	DEBUGSESSION
Test a stored procedure	SYSADM or DBADM or EXECUTE or CONTROL for the package associated with the stored procedure (for SQL stored procedures or Java stored procedures with embedded SQL)
Drop a stored procedure	You must have ownership of the stored procedure and at least one of the following: DELETE privilege DROPIN privilege for the schema or all schemas SYSADM or SYSCTRL authority
Update a stored procedure	You must have ownership of the stored procedure and at least one of the following: UPDATE privilege ALTERIN privilege for the schema or all schemas SYSADM or SYSCTRL authority

The IBM Data Studio accesses a number of DB2 system catalog tables on z/OS. Table 27-4 on page 653 lists the privileges required to view the objects in the Database Explorer. The user connecting to DB2 for z/OS must hold privileges listed in Table 27-4 and Table 27-5 on page 653 for SQL stored procedures, and Table 27-6 for Java stored procedures. The privileges can be held by any authorization ID of the process, either the primary authorization ID or any secondary authorization ID.

Table 27-4 Privileges required to view the objects in the Database Explorer in IBM Data Studio

Additional required privileges: ► EXECUTE privilege on DSNTPSMP SELECT privilege on: ► SYSIBM.SYSCOLAUTH ► SYSIBM.SYSCOLUMNS ► SYSIBM.SYSDATABASE ► SYSIBM.SYSDBAUTH ► SYSIBM.SYSINDEXES ► SYSIBM.SYSJAROBJECTS ► SYSIBM.SYSPACKAGE ► SYSIBM.SYSPACKAUTH ► SYSIBM.SYSPACKDEP ► SYSIBM.SYSPARMS ► SYSIBM.SYSPLAN ► SYSIBM.SYSPLANAUTH ► SYSIBM.SYSRESAUTH ► SYSIBM.SYSROUTINEAUTH ► SYSIBM.SYSROUTINES ► SYSIBM.SYSSCHEMAAUTH ► SYSIBM.SYSSYNONYMS ► SYSIBM.SYSTABAUTH ► SYSIBM.SYSTABLES ► SYSIBM.SYSUSERAUTH ► SYSIBM.SYSVIEWS

For DB2 for z/OS V8 and DB2 for z/OS V9, the IBM Data Studio accesses the following catalog and non-catalog tables when creating external SQL stored procedures.

Table 27-5 DB2 system catalog tables accessed when creating SQL stored procedures

SELECT privilege on: ► SYSIBM.SYSDUMMY1 ► SYSIBM.SYSROUTINES ► SYSIBM.SYSPARMS SELECT, INSERT, UPDATE and DELETE privilege on: ► SYSIBM.SYSROUTINES_SRC ► SYSIBM.SYSROUTINES_OPTS ► SYSIBM.SYSPSM ► SYSIBM.SYSPSMOPTS ALL on the global temporary table ► SYSIBM.SYSPSMOUT

For DB2 for z/OS V8 and V9, the IBM Data Studio accesses the following catalog tables when creating JAVA stored procedures.

Table 27-6 DB2 system catalog tables accessed when creating Java stored procedures

SELECT privilege on: ► SYSIBM.SYSROUTINES ► SYSIBM.SYSDUMMY1 ► SYSIBM.SYSPARMS ► SYSIBM.SYSJARCONTENTS ► SYSIBM.SYSJAROBJECTS ► SYSIBM.SYSJAVAOPTS

27.2.3 Unicode support

IBM Data Studio users creating SQL and Java Stored Procedures experience incorrect codepage translation when Unicode Conversion Services (UCS) is not set up. For more information, see *Support for Unicode: Using Conversion Services*, SC33-7050 and the Web site:

```
http://www.ibm.com/servers/s390/os390/bkserv/latest/v2r10unicode.html
```

To determine if UCS is active, issue `'D UNI,ALL'` from an SDSF screen. If the support is installed, you receive output with the actual CCSID entries that have been defined. If UCS is not installed, the following message is returned:

```
CUN2029S CONVERSION ENVIRONMENT IS NOT AVAILABLE
```

The installation of Unicode Conversion Services requires:

- Updating SYS1.PARMLIB member IEASYSxx with UNI=xx
- Adding SYS1.PARMLIB member CUNUNIxx
- Defining Conversion Table with CCSID entries in
  - SYS1.PARMLIB (CUNIMGxx)
- IPLing the system
- De-activating or activating the conversion table

Without Unicode Conversion Services set up, you can initially create, view, and modify a Java stored procedure. However, restoring the source from the database of a previously created Java stored procedure returns the source as a single line with red blocks interspersed, which represent line feeds that have not been translated correctly. The IBM Data Studio support for SQL stored procedures handles the code conversion, and UCS is not required.

27.2.4 Setup for SQL and Java stored procedures

IBM Data Studio uses DB2-supplied stored procedures to build both SQL and Java stored procedures customized by the following jobs:

- DSNTIJSD
- DSNTIJRX
- DSNTIJTM
- DSNTIJMS
- DSNTEJ6W
- DSNTIJSG
- DSNTIJCC

DSNTIJSD

This job installs the debugger server routines. All DB2 for z/OS servers must have the complete set of debugger server routines installed to enable interaction with IBM Data Studio. Depending on your installation, the server routines installed by this job can be used with:

- The IBM Data Studio Unified Debugger clients. SYSPROC.DBG_%MANAGER server routines are installed in the server
- The SQL Debugger clients. SYSPROC.DBG_% server routines are used in the server.

DSNTIJRX

This customization job sets up REXX support, which is used by both DSNTPSMP (used for SQL stored procedures) and DSNTBIND (used for Java SQLJ stored procedures).

DSNTIJTM

This customization job sets up DSNTWR, the external module used by the DB2-supplied stored procedure WLM_REFRESH, which is used when creating SQL and Java stored procedures.

DSNTIJMS

This customization job sets up SQL Wizard support. SQL Wizard is used when creating SQL Statements within the New Stored procedure wizard in IBM Data Studio.

DSNTEJ6W

This customization job creates and populates the RACF resource profile to support WLM_REFRESH in resource class DSNR. It also prepares and executes a sample caller of WLM_REFRESH. Stored procedures that execute in a WLM AE may stay resident whether a resident run option was specified or not, until the WLM application environment (AE) terminates. To ensure that the latest changes execute during the next invocation of the stored procedure, the WLM AE needs to be refreshed. This can be done from the MVS, SDSF console using the following command:

```
V WLM,APPLENV=DB9AWLM,REFRESH
```

where DB9AWLM is the environment name we want to refresh.

Table 27-7 lists the WLM commands.

Table 27-7 WLM commands entered from SDSF

D WLM,APPLENV=DB9AWLM	Displays the environment status
V WLM,APPLENV=DB9AWLM,QUIESCE	Stops the environment
V WLM,APPLENV=DB9AWLM,RESUME	Resumes the environment
V WLM,APPLENV=DB9AWLM,REFRESH	Refreshes the environment

IBM Data Studio automatically performs this refresh operation using the WLM_REFRESH stored procedure. Customization job <hlq>,SDSNSAMP(DSNTIJSG) defines the procedure, and grants the authorization, while <hlq>.SDSNSAMP(DSNTIJTM) binds the package.

The WLM_REFRESH stored procedure requires RACF permissions using an authorization ID that has MVS command authority. This allows IBM Data Studio to perform the refresh on behalf of each developer that uses it, but only one ID has to be granted MVS command authority. <hlq>.SDSNSAMP(DSNTEJ6W) includes the RACF defines and a sample calling program.

The RACF class DSNR needs to be activated first. This is done using the RACF panels described in Table 27-8.

Table 27-8 Activate Class DSNR

RACF panel	**Option to select**
RACF Services Option Menu	5
RACF System Security Options Menu	3
RACF Set Class Options Menu, panel 1	Enter YES in To CHANGE options for SPECIFIC CLASSES field
RACF Set Class Options Menu, panel 2	Enter DSNR in CLASS field and YES in ACTIVE field

DSNTIJSG

This customization job sets up numerous DB2-supplied stored procedures used when creating SQL and Java stored procedures including:

- DSNTPSMP used for SQL support
- DSNTJSPP used for Java support
- DSNTBIND used for Java SQLJ support
- WLM_REFRESH used by IBM Data Studio
- Miscellaneous stored procedures

Setup specific to SQL stored procedures

The following setup is needed for IBM Data Studio to create external SQL stored procedures. The same setup applies when creating SQL stored procedures using the Rational Application Developer and .NET products:

1. Configure DSNTPSMP.
2. Optionally, define the data set for the //CFGTPSMP statement.

Customization job <hlq>.SDSNSAMP(DSNTIJSG) includes the registration for DSNTPSMP and the GRANT authorization for execution. The WLM AE needs to be configured in this setup job. Make sure to specify a WLM AE with NUMTCB=1. Create the proc that runs in this application environment using <hlq>.SDSNSAMP(DSN8WLMP).

When DSNTPSMP executes, it creates a compiled SQL load module using the C compiler in the data set referenced by //SQLLMOD in its WLM AE. This same data set needs to be included in STEPLIB in the WLM proc where the user's SQL stored procedure created by DSNTPSMP will run.

Configure //CFGTPSMP (optional step)

This is an optional DD statement that can be included in the WLM proc that executes DSNTPSMP. This is a configuration file that externalizes some settings for DSNTPSMP. It is expected that additional options will be added to this data set in future releases.

The definition of the configuration file we used on DB9A is listed in Example 27-3.

Example 27-3 Sample CFGTPSMP configuration data set

```
;-THE CONFIGURATION KEYWORDS AND VALUES ARE AS FOLLOWS:
;-
;-                  .-CBCDRVR--.
;-C_COMPILER--=--+----------+----------------------------|
;-               |-CCNDRVR--|
;-               |-CBC320PP-|
;-               '-EDCDC120-'
;-THE NAME OF THE C COMPILER TO USE. ADJUSTMENT OR ADDITIONAL
;-CONFIGURATION OF THE WLM ENVIRONMENT IS USUALLY REQUIRED
;-WHEN CHANGING THE C COMPILER.
;-
VALIDATE_BIND = DEFAULT
;- DEFAULT, PERMIT, ENFORCE
;-SPECIFIES INSTALLATION CONTROL FOR ALL BUILDS OVER THE
;-USAGE OF THE BIND PACKAGE OPTION VALIDATE(BIND). CHANGING
;-THE DEFAULT MAY PROVIDE A PERFORMANCE IMPROVEMENT.
ISOLATION_DEFAULT = CS
;-    CS  OR RR
;-SPECIFIES INSTALLATION CONTROL OVER THE DEFAULT VALUE FOR
;-THE BIND PACKAGE OPTION ISOLATION.  CHANGING THE DEFAULT
;-MAY PROVIDE A PERFORMANCE IMPROVEMENT.
;
```

```
CURRENTDATA_DEFAULT = YES
;-SPECIFIES INSTALLATION CONTROL OVER THE DEFAULT VALUE FOR
;-THE BIND PACKAGE OPTION CURRENTDATA.  CHANGING THE DEFAULT
;-MAY PROVIDE A PERFORMANCE IMPROVEMENT.
;
DSNTPSMP_TRACELEVEL= LOW
;- OFF, LOW, MEDIUM, HIGH
;-CONTROLS THE LEVEL OF DSNTPSMP TRACE DATA WRITTEN OUT TO
;-THE DD:SYSTSPRT DATASET IN THE WLM ADDRESS SPACE. SETTING
;-THE VALUE TO OFF WILL MINIMIZE, NOT ELIMINATE, LOG RECORDS
;-WRITTEN TO DD:SYSTSPRT IN THE WLM-SPAS.
;
```

Setup specific to Java stored procedures

The following setup is needed for IBM Data Studio to create Java stored procedures. The same setup applies when creating Java stored procedures using the Rational Application Developer product:

- Install JDBC drivers
- Create the JAVAENV data set
- Verifying the DB2 and Java setup on z/OS

Install JDBC drivers

The DB2 JDBC driver needs to be set up in your environment, which is done through SMP/E. The *DB2 Program Directory*, GI10-8216, describes the installation for this in the Receive Sample job DSNRECV3 for ODBC/JDBC/SQLJ.

Create the JAVAENV data set

See 13.3.7, “Setting up the JAVAENV data set for Java stored procedure execution” on page 188.

Verifying the DB2 and Java setup on z/OS

Before using IBM Data Studio, you can verify your Java setup on z/OS by running a simple Java application. This example includes steps for creating DDL, creating the procedure in the DB2 Catalog, and creating a stored procedure and a calling program for the stored procedure. To create a Java stored procedure without the tooling requires a .profile to be set up, and an example exists for creating this as well.

Customize DSNTIJSG for WLM_REFRESH

Customization needed for WLM_REFRESH includes specifying the WLM AE. The proc that runs in this WLM AE requires all APF-authorized data sets.

Creating multiple versions of DSNTPSMP

Multiple versions of DSNTPSMP for external SQL stored procedures are needed if there are different resource requirements for the same stored procedure needed on the same DB2 system during the stored procedure build process. That is there may be a test version and a production version where the WLM proc has different data sets needed in either STEPLIB for external SQL stored procedures.

When multiple versions of either of these DB2-supplied stored procedures are required, first register a new copy of DSNTPSMP with a new schema (SYSPROC is the default). In that registration, specify a new WLM proc where this copy of DSNTPSMP will execute and complete other similar steps as described in “Setup specific to SQL stored procedures” on page 656 for SQL stored procedures.

Selecting a different version of DSNTPSMP

SQL stored procedures can specify a different schema value, and optionally a different build utility than the DB2-supplied DSNTPSMP REXX stored procedure. You specify the default build utility schema that will be used by clicking **Window → Preferences → Data → Stored Procedures and User-defined Functions → SQL - External → Deploy Options.** Figure 27-2 shows an example for setting the build utility, DSNTPSMP, with a different schema value from the SYSPROC default.

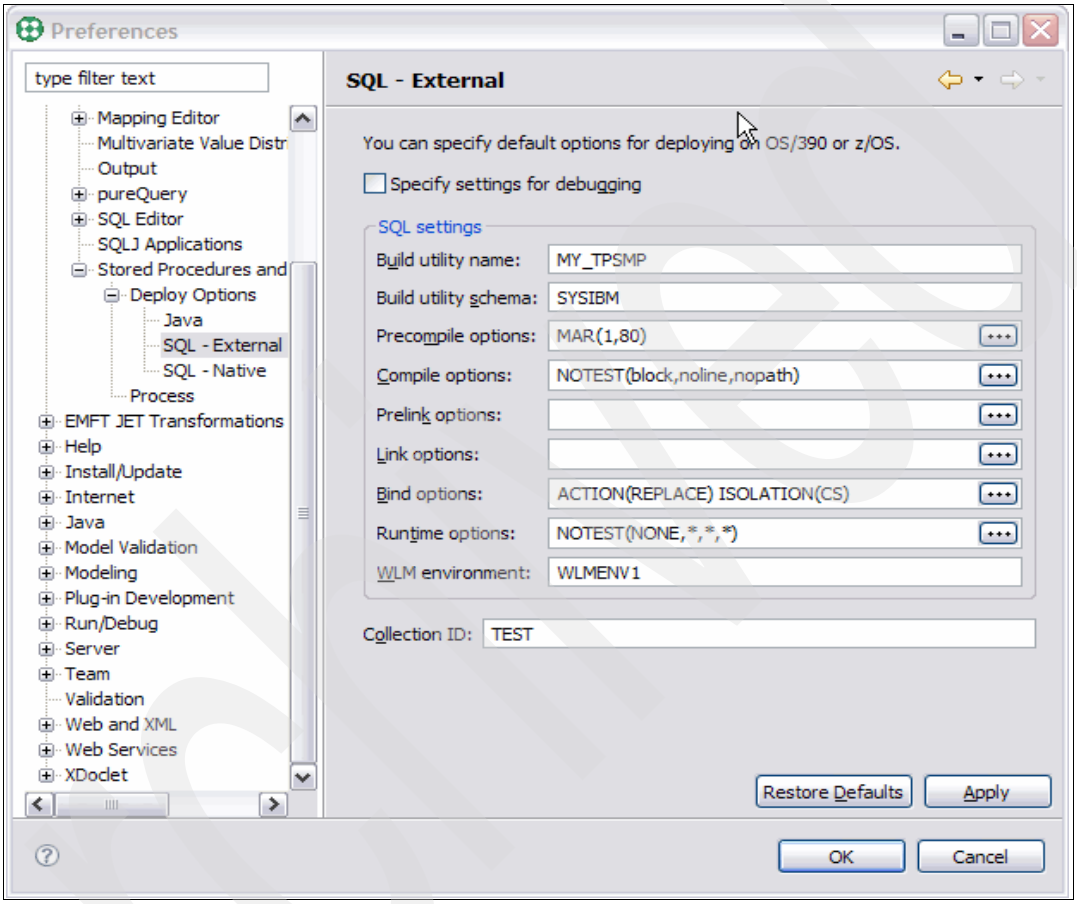

Figure 27-2 DSNTPSMP setting with different schema

When deploying a new SQL or Java stored procedure using the Deploy wizard, a different build schema and utility (external SQL only) can be selected from the **Routine options** page → **Deploy Options** tab → **Build utility** field as shown in the example in Figure 27-3.

DSNTIJCC

This job installs the DB2-supplied stored procedures that are called to perform database administration tasks from the client. IBM Data Studio uses these stored procedures for deploying using binaries external SQL stored procedures. See the discussion in “24.1.2, “Setting up DB2-supplied stored procedures” on page 502”.

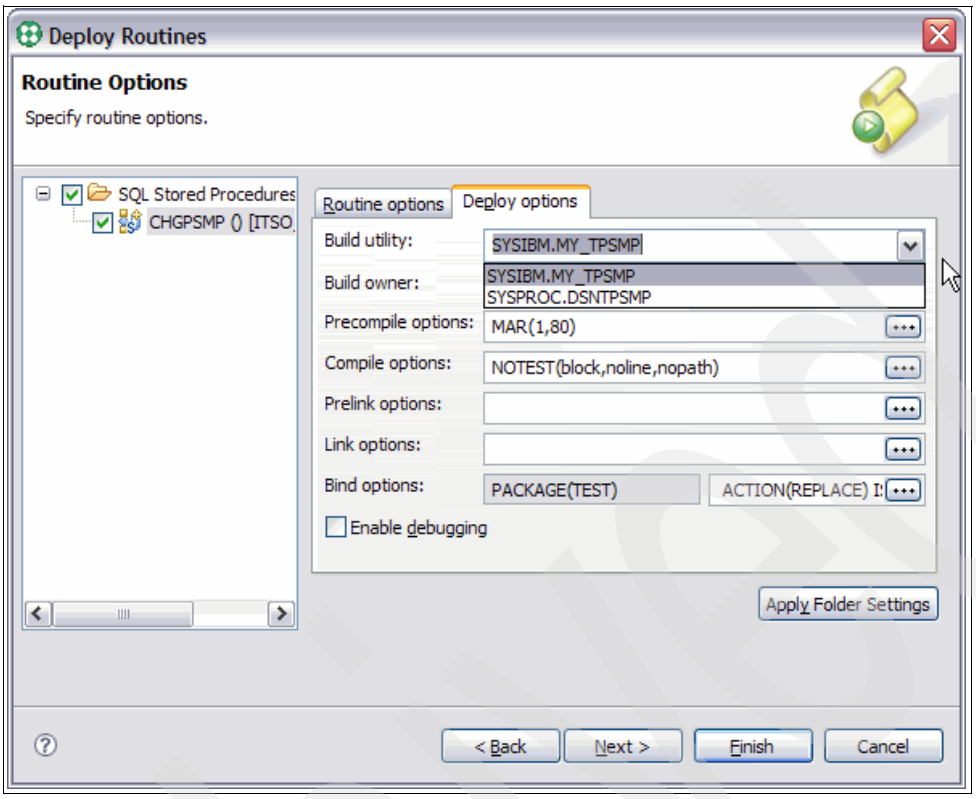

Figure 27-3 Multiple versions of schema

27.2.5 IBM Data Studio Actual Costs setup

SYSPROC.DSNWSPM is a stored procedure optionally used on the SQL Statement page when initially creating either a Java or SQL stored procedure for z/OS. This stored procedure measures the following areas of a specific SQL statement in your stored procedure:

- CPU time
- Latch/lock wait time
- Getpages
- Read I/Os
- Write I/Os

The setup steps are as follows:

Define DSNWSPM as a stored procedure

The CREATE PROCEDURE definition is described in Example 27-4. Since DB2 for z/OS V8 requires WLM-managed SPAS, we defined this stored procedure to be WLM-managed and specified the same environment, DB8AWLM2, used by other DB2-supplied language C and Assembler stored procedures.

Example 27-4 Register the procedure

```
CREATE PROCEDURE SYSPROC.DSNWSPM
(INOUT PA CHAR(8) FOR BIT DATA,
INOUT PB CHAR(8) FOR BIT DATA,
OUT P1 CHAR(16),
OUT P2 CHAR(16),
OUT P3 INTEGER,
OUT P4 INTEGER,
```

```
OUT P5 INTEGER,
OUT P6 INTEGER,
OUT P7 INTEGER)
PROGRAM TYPE MAIN
EXTERNAL NAME DSNWSPM
COLLID DSNWSPM
LANGUAGE ASSEMBLE
RUN OPTIONS 'TRAP(ON),TERMTHDAC(UADUMP)'
PARAMETER STYLE GENERAL
WLM ENVIRONMENT DB8AWLM2
COMMIT ON RETURN NO
```

Bind the DSNWSPM package

After the procedure is registered in the DB2 catalog, we bind the package using the command described in Example 27-5.

Example 27-5 Bind the package

```
BIND PACKAGE(DSNWSPM) CURRENTDATA(NO) -MEMBER(DSNWSPM) ACT(REP) ISO(CS) VAL(BIND)
BIND PLAN (DSNWSPM) PKLIST(DSNWSPM.DSNWSPM) ISO(CS) ACTION(REPLACE)
```

Set up accounting information

The final setup step for actual costs is to ensure that the DB2 accounting trace is running. If not, issue this command to start it:

```
-START TRACE(ACCTG) CLASS(1,2,3)
```

27.2.6 IBM Data Studio and JDBC driver selection

IBM Data Studio uses the JDBC driver specified in the Connection wizard in the following three areas:

- The server connection
- The generated stored procedure (only Java SQLJ requires the Driver significance[3], which is included in the *.ser file)
- The runtime environment for Java stored procedures, both SQLJ and JDBC, which are determined by the WLM SPAS //JAVAENV DD statement

The default JDBC driver that IBM Data Studio uses is the IBM Data Server Driver for JDBC and SQLJ.

Build process generates Java stored procedure

The build process performed by IBM Data Studio generates the connection in the Java stored procedure using the default syntax, “jdbc:default:connection”. This eliminates any JDBC Driver significance in the stored procedure source. Instead, the JDBC Driver significance is included in the *.ser file for Java SQLJ stored procedures during the customization process. No Driver significance is included in generated Java JDBC stored procedures. The *.ser file is created during customization for Java SQLJ stored procedures and is performed by IBM Data Studio using DB2SQLJCUSTOMIZE.

Runtime determines JDBC driver

The JDBC driver used for runtime for both Java SQLJ and Java JDBC stored procedures is determined by the //JAVAENV DD statement in the WLM SPAS where the stored procedure

[3] The Driver significance pertains to the set of information regarding the JDBC driver being used.

executes. The JCC_HOME environment variable in this file specifies the location in UNIX System Services (USS) where the Universal JDBC driver is installed.

The default directory structure for JCC_HOME is /usr/lpp/db2/db2910/jcc. For more details on the //JAVAENV DD statement, see 13.3.7, “Setting up the JAVAENV data set for Java stored procedure execution” on page 188.

See the section “Installing the IBM DB2 Driver for JDBC and SQLJ” under “Installation and Migration” in the following URL for the DB2 for z/OS V9 Information Center for more information on the IBM Data Server Driver for JDBC and SQLJ.

http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp

27.2.7 Java SDKs used by IBM Data Studio

In this section we describe your options for the Java Software Developer Kit (JDK) selection when using the DB2 IBM Data Studio to create Java stored procedures on z/OS platforms.

Overview

Both JDK 1.4.2 and JDK 1.5 are supported when creating Java stored procedures on DB2 for z/OS V9. DB2 for z/OS V8 only supports the JDK 1.4.2. The two areas that reference a JDK are:

- The build time of the stored procedure by IBM Data Studio, which includes compiling the source
- Optionally, at runtime of the stored procedure in the Work Load Manager (WLM) Stored Procedure Address Space (SPAS)

Java methods used during build time must be available at runtime in the Java Runtime Environment (JRE™), otherwise an execution error indicating a mismatch occurs in the WLM SPAS. For instance, using a Java stored procedure that includes a JDK 1.5 method fails if it executes in a WLM SPAS referencing a Java 1.4.1 JRE.

IBM Data Studio ships with a JDK 1.5 library. This is the default JDK used when developing Java stored procedures and applications in IBM Data Studio. In a typical install, the JDK is found in C:\Program Files\IBM\SDP70\jdk.

However, the user can opt to compile with the 1.4 level by setting the JDK level in three places:

- In the workspace Preferences
- In the Project’s Properties
- In the stored procedure’s Deploy options

Overriding the default JDK

Overriding the default JDK used by the IBM Data Studio client is done through the workspace Preferences.

- Click **Window** → **Preferences** → **Stored Procedures and User Defined Functions** → **Deploy Options.** See Figure 27-4 on page 662.
- Click **Browse**. Point the File Browser to the directory where the overriding JDK is located.

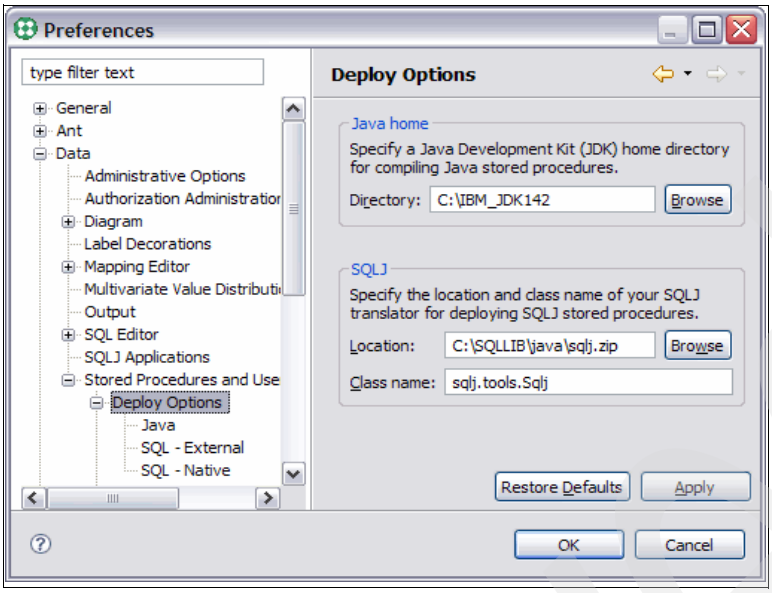

Figure 27-4 Overriding the default JDK

Runtime of the Java stored procedure

The SDK used at runtime is determined by the //JAVAENV DD statement in the WLM SPAS where the stored procedure executes. Specifically, the JAVA_HOME environment variable included in this DD statement determines the SDK that is selected.

The base product of DB2 for z/OS V8 includes support for both the SDK 1.3.1 and SDK 1.4.1. There is no support for SDK 1.3.0 or SDK 1.4.0 in DB2 V8.

When using SDK 1.4.1, the XPLINK(ON) parameter is required in the //JAVAENV DD statement. If the XPLINK(ON) parameter is included when specifying an SDK 1.3.1, the following information message is written to the WLM SPAS:

```
CEE3611I The run-time option XPLINK= was an invalid run-time option or is not supported
in this release of Language Environment.
```

When the XPLINK(ON) parameter is not included in a //JAVAENV DD statement that specifies SDK 1.4.1, the WLM SPAS does not initialize and the following error message is included in the WLM SPAS joblog:

```
+DSNX961I  DSNX9WLJ ATTEMPT TO PERFORM JNI FUNCTION CreateJavaVM  421
 FAILED FOR STORED PROCEDURE . .  SSN= DB8A     PROC= DB8AJAV1 ASID=
 008E  CLASS=  METHOD=  ERROR INFO= DSNX9WLS ESTAE ENTERED
```

If JSPDEBUG is turned on in the same //JAVAENV DD statement, information like the following is also included indicating that a call was made from a NOXPLINK-compiled application to an XPLINK-compiled exported function in DLL libjvm.so, and the XPLINK(ON) runtime option was not specified:

```
CEE3501S The module libjvm.so was not found.
         From entry point initjvm at compile unit offset +000014A0 at entry off
```

Runtime //JAVAENV DD statement examples

The following examples apply to Java stored procedures using the Legacy JDBC driver.

Example 27-6 is for SDK 1.3.1.

Example 27-6 Legacy JDBC Driver - SDK 1.3.1

```
ENVAR("DB2_HOME=/usr/lpp/db2/db2710",
"JAVA_HOME=/usr/lpp/java/IBM/J1.3",
"DB2SQLJPROPERTIES=/u/DB7PU/db2sqljjdbc.properties"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

Example 27-7 is for SDK 1.4.1.

Example 27-7 Legacy JDBC Driver - SDK 1.4.1

```
XPLINK(ON),
ENVAR("DB2_HOME=/usr/lpp/db2/db2710",
"JAVA_HOME=/usr/lpp/java/IBM/J1.4",
"DB2SQLJPROPERTIES=/u/DB7AU/db2sqljjdbc.properties"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

The following examples apply to Java stored procedures using the Universal JDBC driver.

Example 27-8 is for SDK 1.3.1.

Example 27-8 Universal JDBC driver - SDK 1.3.1

```
ENVAR("JCC_HOME=/usr/lpp/db2/db2810/jcc",
"JAVA_HOME=/usr/lpp/java/IBM/J1.3"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

Example 27-9 is for SDK 1.4.1.

Example 27-9 Universal JDBC driver - SDK 1.4.1

```
XPLINK(ON),
ENVAR("JCC_HOME=/usr/lpp/db2/db2810/jcc",
"JAVA_HOME=/usr/lpp/java/IBM/J1.4"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

27.2.8 Overview of routine development with IBM Data Studio

When the IBM Data Studio is launched, it asks for a directory to be used for the tooling's workspace. A default directory is provided, but the user can change this to another directory. IBM Data Studio then checks if there is a DB2 database directory in the client's workstation. If there is one, it creates Connection objects for each of the DB2 aliases in the DB2 database directory. Figure 27-5 shows the basic UI flow to get you started to using IBM Data Studio.

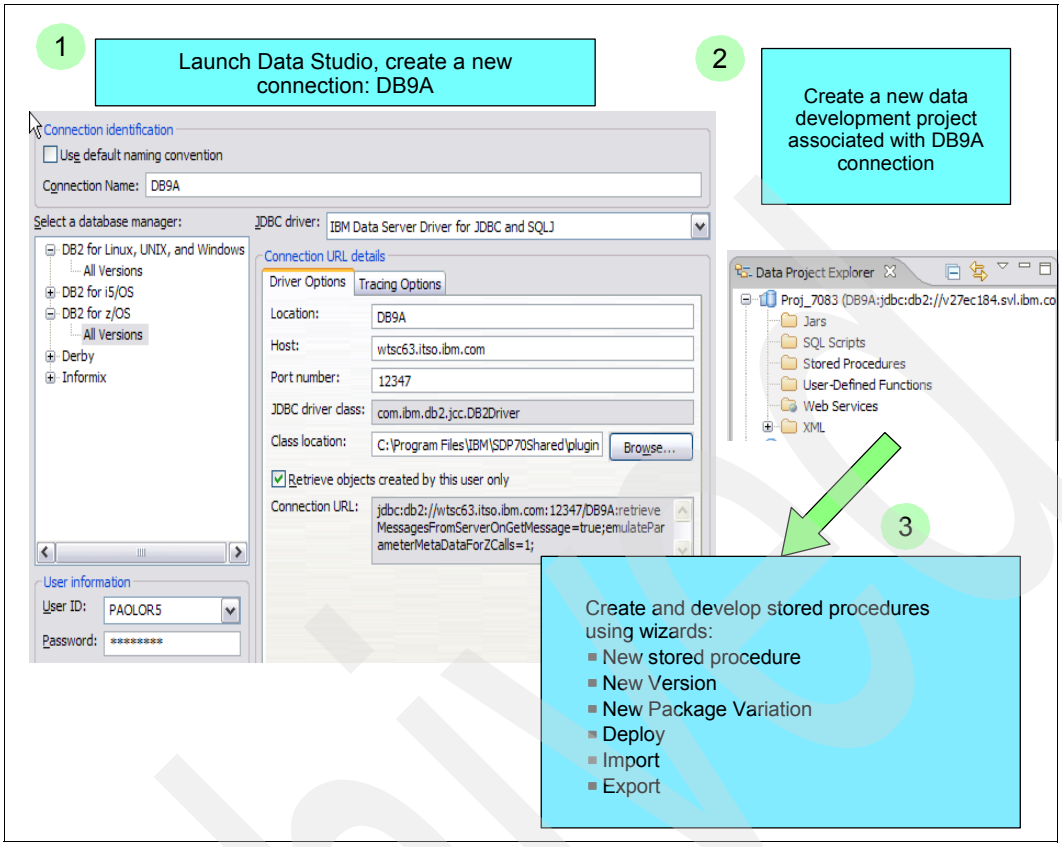

Figure 27-5 Starting IBM Data Studio

Like Development Center, IBM Data Studio creates external SQL and Java stored procedures on z/OS using multiple DB2-supplied stored procedures. The main DB2-supplied stored procedures that perform the processing on z/OS for IBM Data Studio are:

- DSNTPSMP for SQL stored procedures
- SQLJ.DB2_INSTALLJAR
- SQLJ.DB2_REPLACEJAR
- SQLJ.DB2_UPDATEJARINFO

When connected to DB2 for z/OS V9, IBM Data Studio can create both external SQL stored procedures as well as Native stored procedures which no longer require DSNTPSMP. The tooling identifies the type of SQL stored procedure by appending “(external)” or “(native”) to the stored procedure name displayed in the Data Project Explorer. Also, starting in DB2 9 for z/OS, DSNTJSPP is no longer used for creating Java stored procedures.

External SQL stored procedures built with DSNTPSMP

DSNTPSMP is a REXX DB2-supplied stored procedure that builds External SQL stored procedures on DB2 for OS/390 V7, DB2 for z/OS V8, and DB2 for z/OS V9. Multiple *build* functions are supported by this stored procedure. Table 27-9 shows the functions that are supported.

Table 27-9 DSNTPSMP supported functions

Type	Name	Function
Basic	BUILD	Creates a new SQL procedure only
	ALTER	Updates (most) stored procedure options

Type	Name	Function
	REBIND	Perform Bind Package again (not REBIND command)
	DESTROY	Remove an existing SQL procedure
	REBUILD	Builds an SQL procedure. Destroys existing one first. Best for changing parameter-declarations
Modify	ALTER_REBIND	Perform ALTER of collection ID
	ALTER_REBUILD	Perform procedure-body updates
SQL Debugger Enabling (z/OS V8 only)	BUILD_DEBUG	Basic function plus Debugger “hooks”
	REBUILD_DEBUG	Basic function plus Debugger support
	ALTER_REBUILD_DEBUG	Hybrid function plus Debugger support
Identification	QUERYLEVEL	Returns Interface and Service level of DSNTPSMP

The following steps are performed by IBM Data Studio to create external SQL stored procedures on DB2 V7, V8 and V9:

1. Launch IBM Data Studio, and select a workspace.
2. Create a connection or Reconnect to an existing connection to a DB2 server.
3. Create a Data Development Project and set the Target connection to the above database server.
4. Create a new SQL stored procedure using the New Stored Procedure wizard. This wizard has an embedded SQL Wizard to help develop SQL statements. Alternatively, existing SQL statements can be imported into the SQL stored procedure. See 27.4.4, “Creating SQL statements to use in your stored procedure” on page 685 for details on using the SQL Wizard.
5. The New Stored Procedure wizard allows the user to deploy the stored procedure at the end of the wizard, or to simply display the generated DDL and configuration properties in the Editor. Assume the automatic deploy is selected.
6. Now DSNTPSMP is called.
7. DSNTPSMP performs the following steps to create the SQL stored procedure on z/OS:
   a. SQL precompile
   b. C precompile
   c. C compile and prelink
   d. Link
   e. Bind package
   f. Register procedure in the DB2 catalog
   g. Save options

Figure 27-6 describes how the IBM Data Studio creates SQL stored procedures on z/OS.

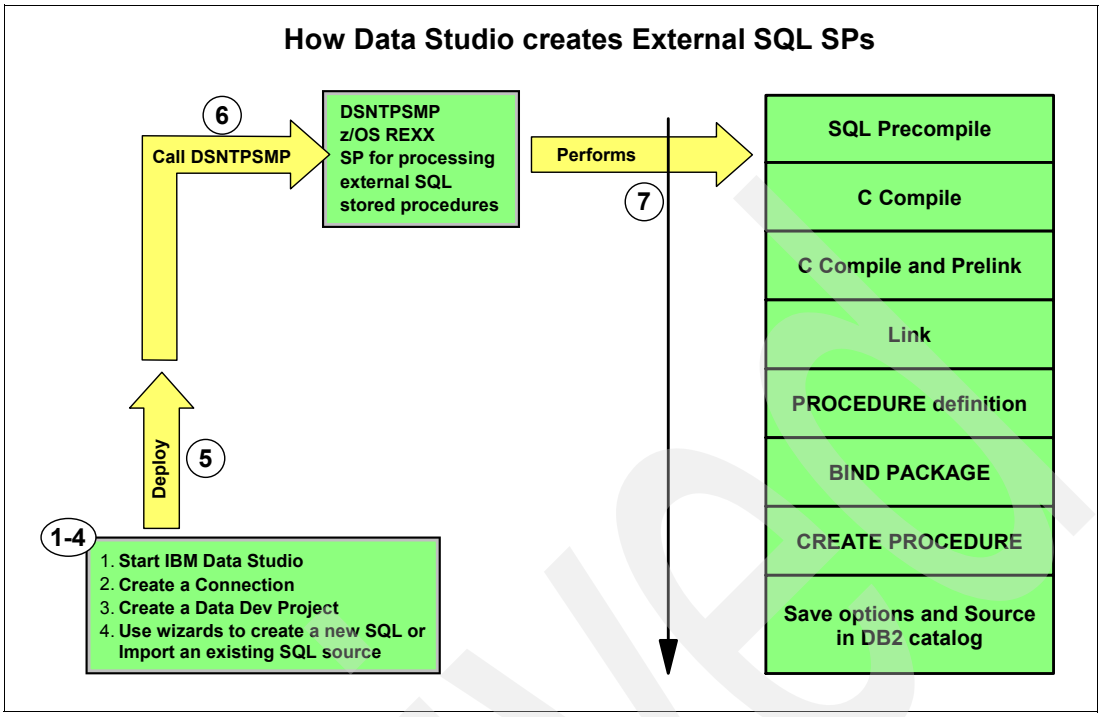

Figure 27-6 How IBM Data Studio creates SQL stored procedures

Java stored procedures built on the client

IBM Data Studio builds Java stored procedures on DB2 for OS/390 V7, DB2 for z/OS V8, and DB2 for z/OS V9, when connected to the server with the IBM Universal Driver.

1. As in SQL stored procedures, use the New Stored Procedure wizard to create a Java stored procedure.
2. Deploy the stored procedure.
3. For Java stored procedures using dynamic (JDBC) SQL, Data Studio issues **`javac`** to compile the stored procedure.

   For Java stored procedures using static (SQLJ) SQL, IBM Data Studio issues **`sqlj`** to translate and compile the stored procedure.

   Data Studio issues **`db2sqljcustomize`** which does the following:

   – Updates the .Ser file created in Step 2
   – Optionally binds the stored procedure

   Then Data Studio issues a **`jar`** command to jar the .SER, .class, and optionally the .java files.
4. Data Studio calls SQLJ.DB2_INSTALL_JAR to install the jar in the server; or SQLJ.DB2_REPLACE_JAR to replace the jar when the stored procedure already exists in the server.
5. Data Studio issues the CREATE PROCEDURE DDL to register the stored procedure into the catalog.
6. Data Studio calls SQLJ.DB2_UPDATEJARINFO to copy the Java source into the catalog.

Figure 27-7 describes how the IBM Data Studio creates Java stored procedures on z/OS.

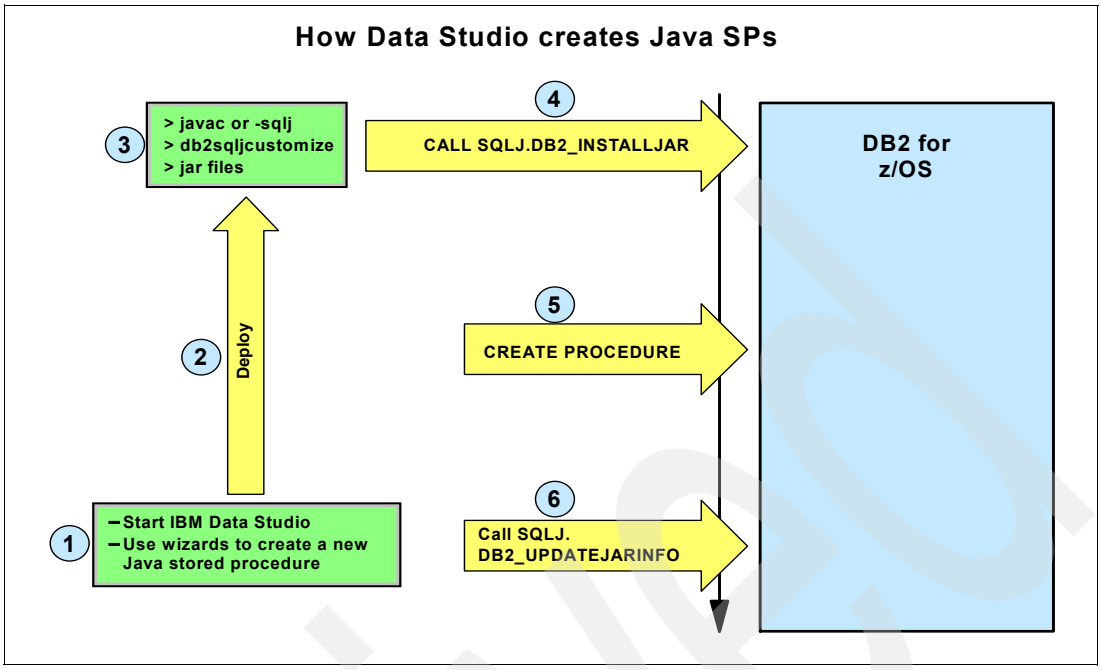

Figure 27-7 How IBM Data Studio creates Java stored procedures

27.3 Navigating through the IBM Data Studio workspace

This section applies to all platforms supported by IBM Data Studio. IBM Data Studio uses the Data Perspective, which contains the following five views that can be rearranged or closed. Additional views can also be opened.

- Database Explorer view
- Data Project Explorer view
- Editor view
- Data Output view
- Outline view

27.3.1 Database Explorer view

The Database Explorer displays the connections to database servers that were previously created in the workspace as well as those in the client DB2's database directory, if a DB2 for LUW is installed. Each connection shows a hierarchical view of the database and the objects within them. The cataloged stored procedures and user-defined functions are listed under a schema folder in this view, as shown in Figure 27-8.

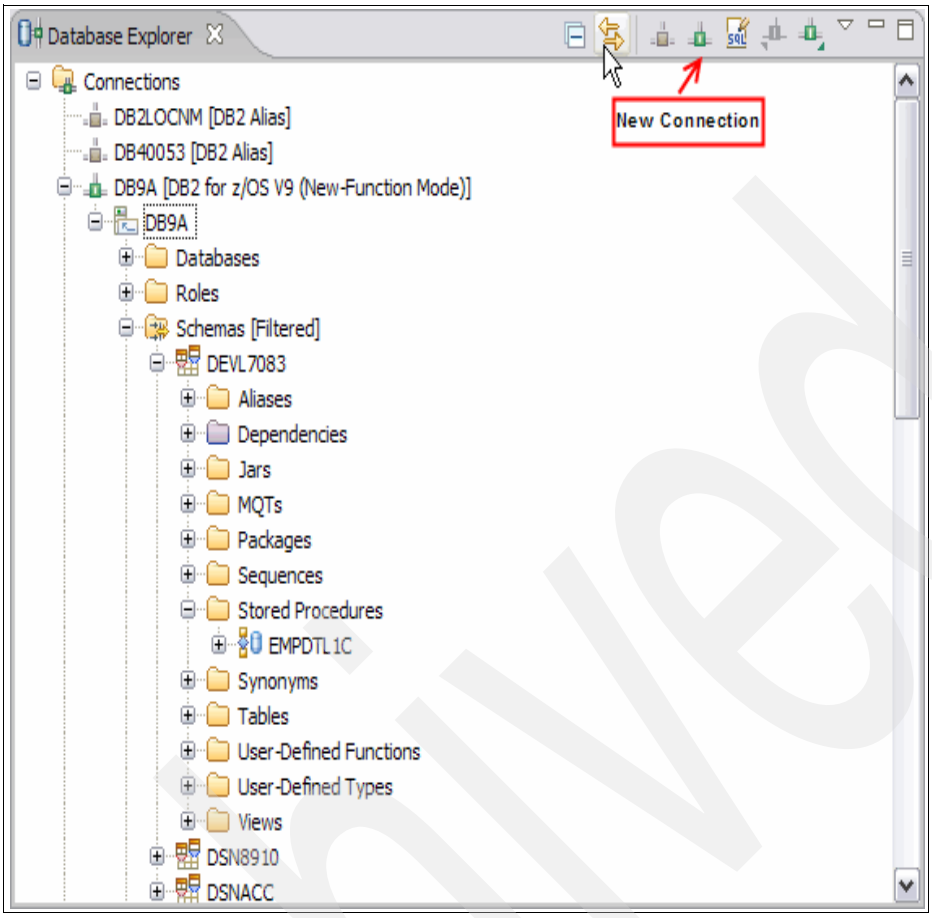

Figure 27-8 Database Explorer

In addition to viewing the stored procedures and UDFs that are on the server, the Database Explorer allows you to view and work with other database objects such as tables, triggers, views, etc.

Database Explorer menu bar

The Database Explorer contains a toolbar at the top of the view for doing the following tasks:

- Disconnect a currently selected connection
- Create a New connection
- Launch the SQL Editor for creating SQL ad-hoc statements against a connection
- Export Connections
- Import Connections

In 27.4, “Developing stored procedures with IBM Data Studio” on page 680, we discuss the Connection Wizard and the SQL Editor in detail.

Context menu actions

Each object in the Database Explorer has a context menu that shows the actions that can be performed on this object. To view the actions on a folder or object, right-click the folder. In this book, we examine the context menu actions available for stored procedures only. To activate the context menu for stored procedures, expand the database connection → **Schemas** → **`your schema`** → **Stored Procedures.** Right-click on this to view the following context menu actions:

1. **Filter -** This action allows you to filter what is displayed in the Stored Procedures folder in two ways, as shown in Figure 27-9 on page 669. Right-click the Stored Procedure folder → **Filter** to launch the Filter dialog.
   – Using an Expression: pull-down list allows the user to create clause-like expressions such as LIKE 'DEV%'.
   – Select from a list of stored procedure names.

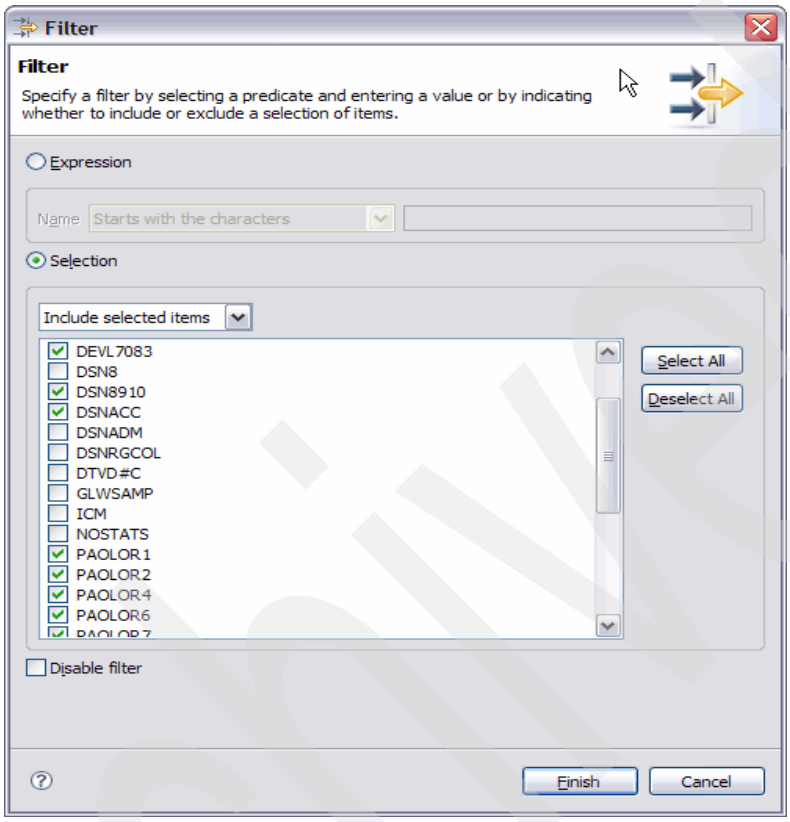

Figure 27-9 Filtering option

2. **New** - This action creates a new stored procedure using the New Stored Procedure wizard or the SQL Editor.
   – Using the New Stored Procedure wizard: Right-click the Stored Procedures folder → **New** → **With Routine Editor.** This launches the New Stored Procedure wizard and guides the user to creating a new SQL or Java stored procedure. At the end of the wizard, you are asked to give the name of a project that will contain this new object. Section 27.4, "Developing stored procedures with IBM Data Studio" on page 680 gives more details on the New Stored Procedure wizard.
   – Using the SQL Editor: Right-click the Stored Procedures folder → **New** → **With SQL Editor** creates a blank editor in the Editor view. You can type your CREATE PROCEDURE statement there, <add xref to SQL Editor>.
3. **Refresh** - This action reads the latest information for the specific folder from the server catalog. Right-click the Stored Procedures folder → **Refresh** to refresh the list of deployed or cataloged stored procedures.
4. **Deploy:** This action launches the Deploy wizard. Right-click the Stored Procedures folder → **Deploy** allows you to redeploy one, some, or all the stored procedures listed in this folder. Right-click a specific stored procedure → **Deploy** allows you to redeploy only the selected stored procedure. Figure 27-10 on page 670 shows the first page of the Deploy Wizard. Details on the Deploy Wizard are discussed in 27.6, "Deploying a stored procedure" on page 703.

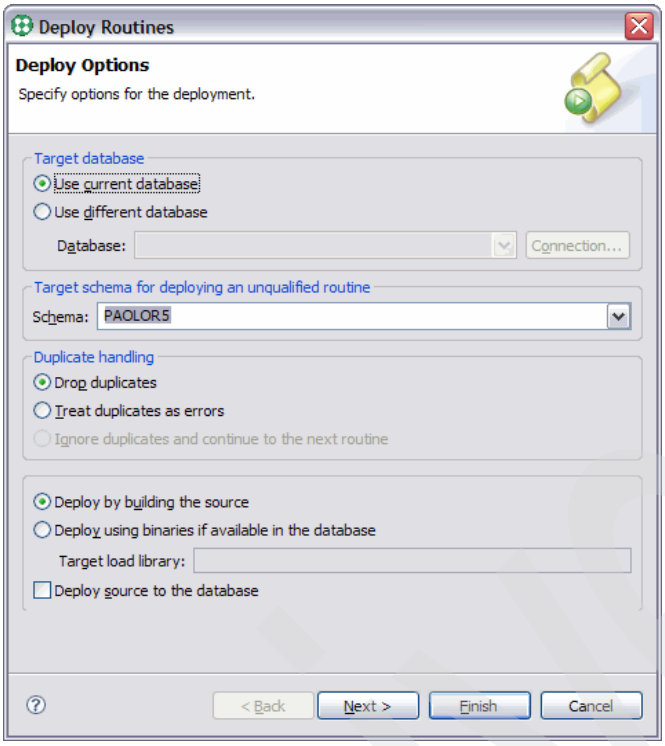

Figure 27-10 Deploy Wizard

The context menu on a specific stored procedure shows additional actions that can be taken on the stored procedure. Right-click the stored procedure to activate the following actions:

a. **New version** - This action is only available when the selected stored procedure is a Native SQL stored procedure. This action launches the New Version wizard. This wizard creates a new version of the selected stored procedure and optionally deploys it.

b. **Open -** This action launches either the Routine Editor or the SQL Editor on the Editor view. You are asked to specify a Data Development Project to contain the stored procedure. IBM Data Studio assumes that you are opening the stored procedure for editing. If you want to simply view the properties of this stored procedure, click the Properties tab of the Output View. We discuss this in more detail in 27.3.3, “Output view” on page 673.

c. **Drop -** This action issues a DROP PROCEDURE against the selected stored procedure. A confirmation dialog is displayed before the action is sent to the server.

> **Note:** This action drops all versions of a Native SQL Stored procedure. To drop a specific version, expand the selected stored procedure → **Versions** → select the specific version and activate the Drop action from this version.

d. **Generate DDL -** This action launches the Generate DDL Wizard. In the wizard, you can:

- Generate the CREATE PROCEDURE DDL.
- Optionally generate the associated DROP statement before the CREATE statement.
- Optionally generate any associated COMMENT ON and LABEL ON statements.

- Optionally generate any GRANT statements based on the current privileges held on this stored procedure.
- Execute the generated DDL, or save and edit the generated script. You are asked to specify an existing project to contain the generated script. IBM Data Studio looks at the list of existing projects and defaults the project to one that is using the current server or one that is a “best fit” (that is, same operating system, same version).

Note: For Java stored procedures, Generate DDL does not create a clone of the Java source associated with the Java stored procedure.

Figure 27-11 shows the DDL generated for one of our sample stored procedures.

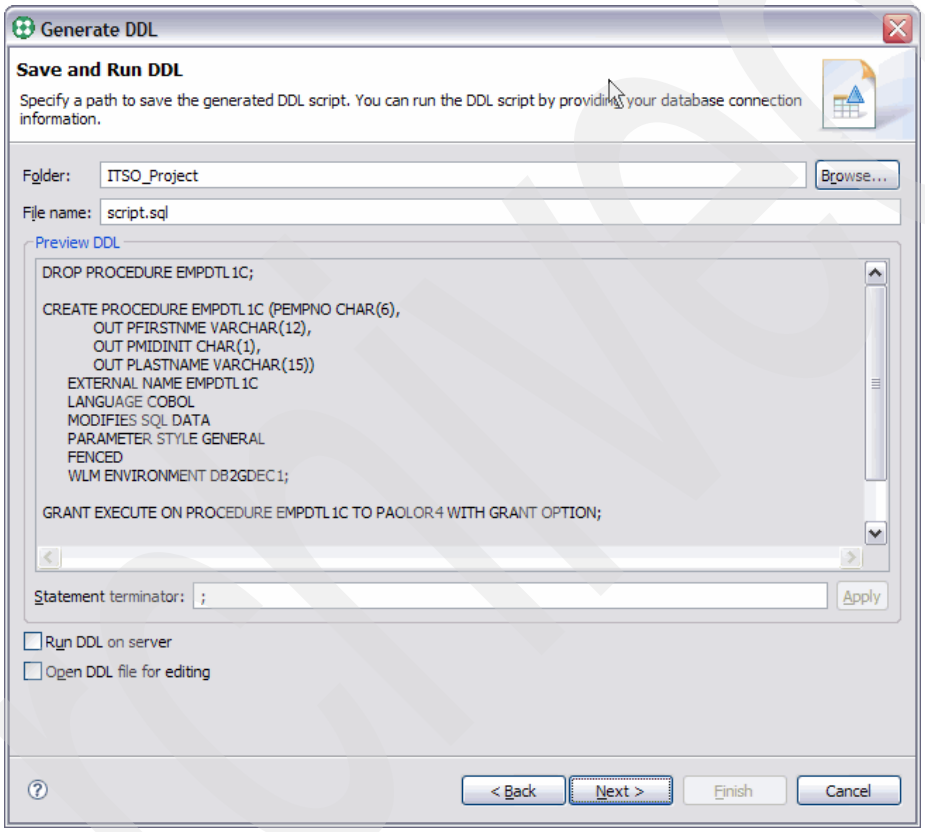

Figure 27-11 Generated DDL for a stored procedure

e. **Analyze Impact** - This action examines the SYSROUTINESDEP and SYSPACKAGEDEP catalogs and lists the objects that have dependencies on this stored procedure.

f. **Run and Run Settings** - These two actions execute the selected stored procedure. Run Settings allows you to preset parameters, and execute SQL statements before and after calling the stored procedure. We discuss this more in 27.4, “Developing stored procedures with IBM Data Studio” on page 680.

g. **Deploy -** This is the same action as in the stored procedure executed against a specific stored procedure.

h. **Generate pureQuery Code -** This action allows you to generate the pureQuery code to call this stored procedure. You need to have an existing Java project and an existing Java program where the pureQuery code is embedded. PureQuery is beyond the

scope of this book. However, see 27.7, “Advanced IBM Data Studio topics” on page 710 for an overview of IBM Data Studio’s pureQuery support.

27.3.2 Data Project Explorer view

The Data Project Explorer view is the main development view for managing your projects. The main project type managed by the Data Project Explorer is the Data Development Project. In this view, you can:

- Manage multiple projects
- Target a specific connection to a project
- Under each project, create and manage objects such as stored procedures, SQL scripts, User-Defined Functions, Jars, XML objects, and Web Services
- Copy and Paste objects from one project to another
- Share a project between teams

Each Data Development Project contains the database objects that you can work on, in an object tree structure. Figure 27-12 on page 672 shows the contents of a Data Development Project.

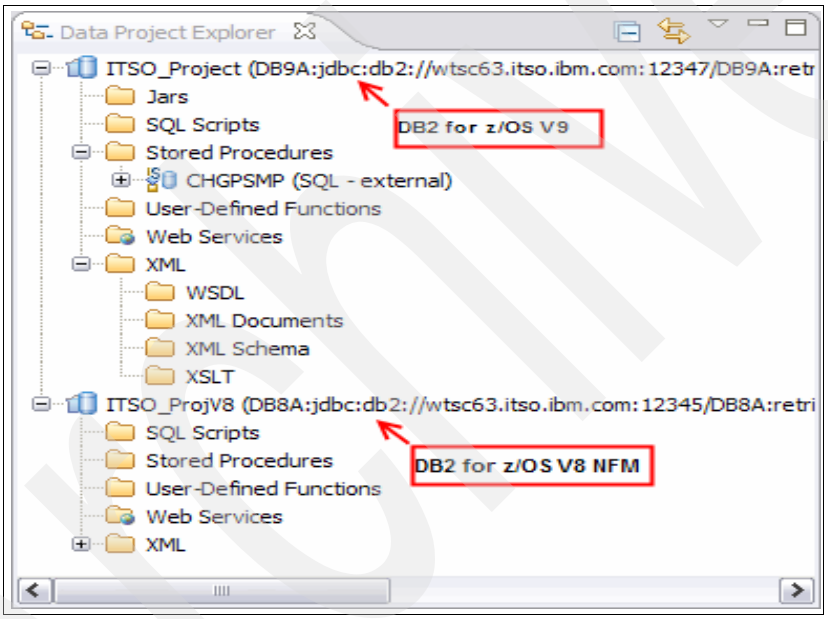

Figure 27-12 Data Project Explorer

Note: Multiple Jars, XML and Web Services are only supported with DB2 for z/OS V9. These folders may exist in projects associated with non-DB2 for z/OS V9 servers for strategic purposes.

Project properties

A set of Project properties associated with the Data Development Project can be used to set default values when creating objects within the project. Right-click the project name → **Properties** to launch the Properties Dialog.

- The Connection page displays the connection properties of the target connection. In 27.4.2, “Creating a connection” on page 682 we discuss how to set the connection properties.

- The Development page allows you to set the Current Schema, as shown in Figure 27-13 on page 673. A discussion of the Current Schema versus Current SQLID is in 7.5.7, “Resolution of unqualified stored procedure names at create time” on page 79.

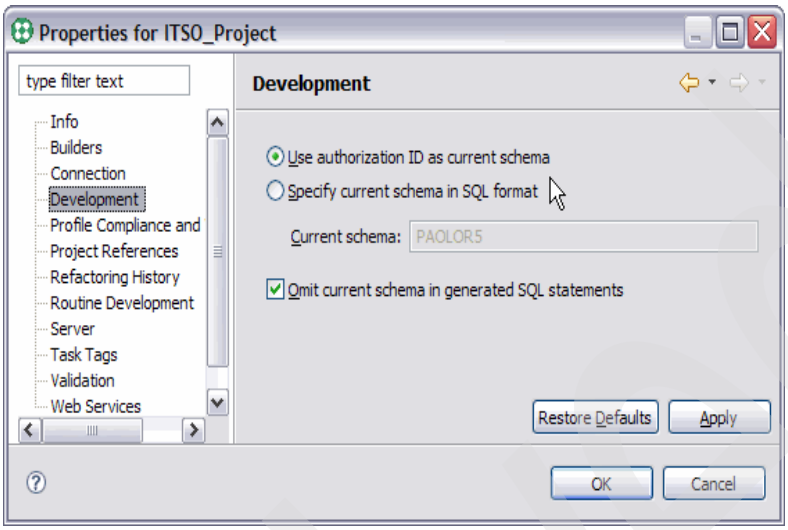

Figure 27-13 Setting current schema

- The Routine Development page allows you to set the JDK level to be used for compiling the Java stored procedures you create. Additionally, you can set the package owner and build owner for SQL and SQLJ stored procedures in this page, as shown in Figure 27-14. Both the package and build owner can be set to secondary authorization IDs.

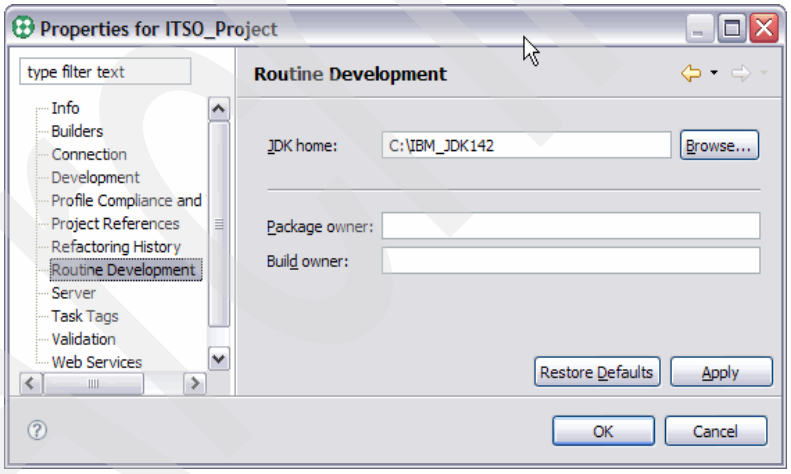

Figure 27-14 Setting package and build owner

27.3.3 Output view

The Output view is used for displaying messages and execution results. Any output-related view in Eclipse is added as a tab to the Output view. We examine two tabs in this view.

Properties view

In the Database Explorer, when a specific object is selected, the properties of the object are retrieved and displayed in the Output view’s Properties tab, as shown in Figure 27-15 on page 674.

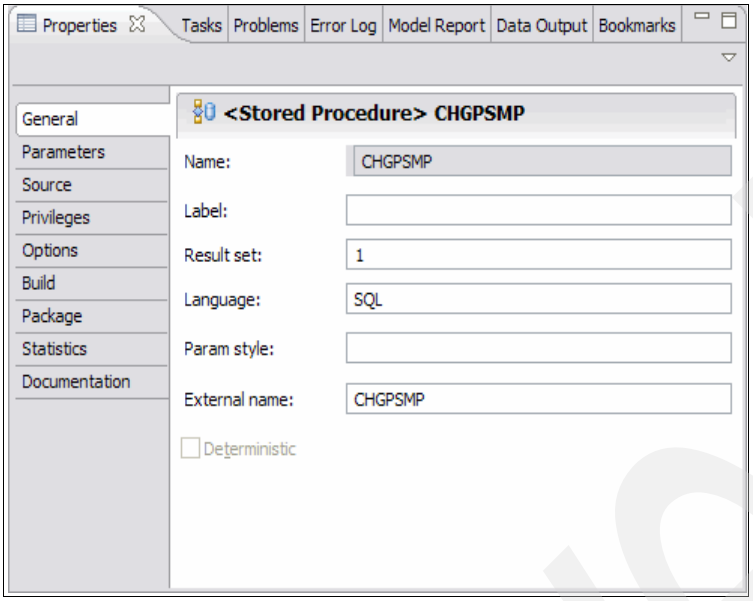

Figure 27-15 Property Browser for stored procedure

The Properties page, in turn, contains multiple tabs which group the stored procedure's attributes into:

- General - Name, Label (on), Result sets returned, Language, Parameter style, external name, Deterministic/Non Deterministic
- Parameters - parameter type (IN, OUT, INOUT), parameter name and parameter data type
- SQL or Java source
- Privileges - Grantee, Grantee type, privilege, Grantor, with grant option
- Procedure options - Specific name, Package ID, data access type, Collection id, ASUTIME, External Security, Stay Resident, Program Type and Commit on Return
- Build options - WLM Environment, Build Utility, Build Owner, Precompile, Compile, Prelink, Link, and Bind options
- Documentation - The text supplied in the *Comment on* statement for this stored procedure is displayed here

The fields in the Property Browser are READ-ONLY. To modify an SQL or Java stored procedure, you need to "Open" the procedure with the Routine Editor, then redeploy it.

Also, the Package and Statistics tabs are not used. To view the package information related to a specific stored procedure, expand the Packages folder for this schema and select the associated package ID from the list. The Properties tab is refreshed with the package information.

The Data Output tab

The Data Output tab of the Output View is used to report execution status and messages for SQL statements and stored procedures. An example of the Data Output is shown in Figure 27-16 on page 675.

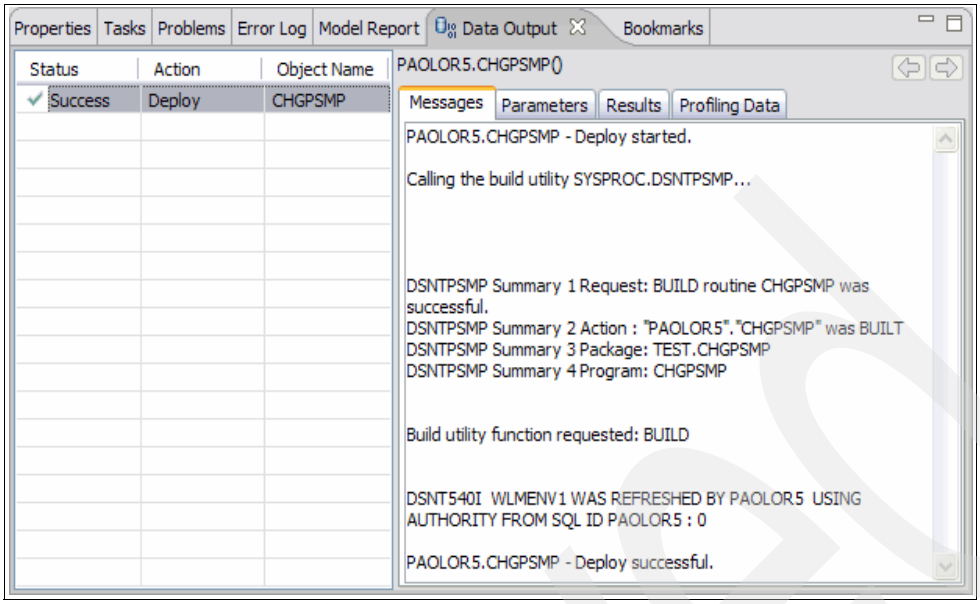

Figure 27-16 Data Output view

The Data Output tab contains two panes: the Status History and the Current Output.

The Status History pane is in turn divided into three columns:

- The Status column shows the current status of each action, such as error or success.
- The Action column shows the type of action, such as Deploy, Run, Export, Import, etc.
- The Object Name column shows the name of the object for each action.

When a specific action in the Status History pane is selected, the Current Output pane is populated with the output details. You can select one of four tabs to view different kinds of output:

- The Messages page of the output section displays the detailed status of each action.
- The Parameters page displays the name and values of the parameters after an object is run.
- The Results page displays the results of the action. If an object returns multiple result sets, you can scroll through the result sets by clicking the arrow at the top right corner of this page.
- Profiling data is only available when the server is DB2 for Linux, UNIX, and Windows.

You can remove an item from the action list by right-clicking the action and clicking **Delete** or **Delete All.**

27.3.4 Editor View

IBM Data Studio uses this view to present different editors to the user. The three main editors that the tooling supports are:

- Routine Editor
- Enhanced SQL Editor
- Java Editor

All three editors share the Editor view. Any object being viewed shows up as a tabbed page in the Editor view.

The Routine Editor

This editor is launched in several ways:

- From the Data Project Explorer, double-click a stored procedure.
- From the Data Project Explorer, right-click **Open** on a specific stored procedure.
- From the Database Explorer, right-click **Open** → **Using Routine Editor** against a specific stored procedure.
- At the end of the New Stored Procedure or New Version wizards, the Routine Editor is refreshed with the generated stored procedure DDL.

Use this editor for viewing and changing the source code and configuration options of a stored procedure you are working on in the Data Project Explorer. This editor is also launched when you want to open an existing stored procedure from the Database Explorer. Figure 27-17 and Figure 27-18 show the contents of the Routine Editor *Source* and Configuration tabs.

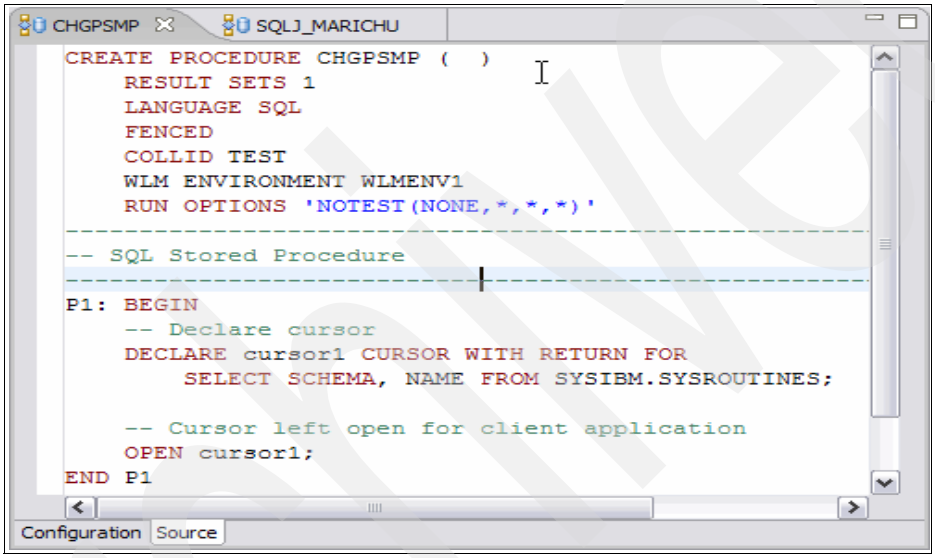

Figure 27-17 Routine Editor - source tab

The Routine Editor's Source tab is a rich editor that supports cut, copy, paste, find and replace, menu and keyboard shortcuts, and syntax highlighting. You can change the default "look and feel" of this page in the Preferences. See 27.4.1, "Starting the IBM Data Studio for the first time" on page 681.

The Routine Editor's Configuration tab shows the properties of the stored procedure grouped similarly as in the Properties tab when a stored procedure is selected from the Database Explorer.

To edit the source code of an SQL stored procedure:

1. In the Data Project Explorer, right-click the routine that you want to modify, and click **Open**. The source code of the routine displays in the Routine Editor view.
2. Edit the source code. You can:
   - Change or add SQL statements directly in the editor.
   - Click **Ctrl-Space** to launch Content Assist.
3. To save your changes, you can:
   - Click **File** → **Save Object** or **File** → **Save All**, or
   - Click the **Save** icon (floppy disk), or type `Ctrl-S`.

After saving your changes, IBM Data Studio replaces the persisted resource corresponding to this stored procedure in the workspace. However, to replace the object in the server, you need to deploy or redeploy the stored procedure. Depending on your deploy options, IBM Data Studio either drops the old routine from the database and creates a routine that reflects the changes you made, or it alters it. Changes to the source rarely cause the procedure to be dropped. Where possible, changes to the source code of SQL stored procedures result in an ALTER command rather than a DROP command.

Finally, the Routine Editor is also used by the Debug Perspective for debugging SQL routines.

Java Editor

The Java Editor is launched from both the Data Perspective and the Java Perspective.

- From the Data Perspective, Data Project Explorer, each Java stored procedure has a Java Source folder. Open this folder and double-click the **.java** or **.sqlj** file in this folder.

  From the Data Project Explorer:

  - Right-click **Open** on the stored procedure. The stored procedure DDL is displayed in the Routine Editor view.
  - Click the **Configuration** tab
  - Click the **.java** or **.sqlj** link at the top of the page (see the box in Figure 27-18).

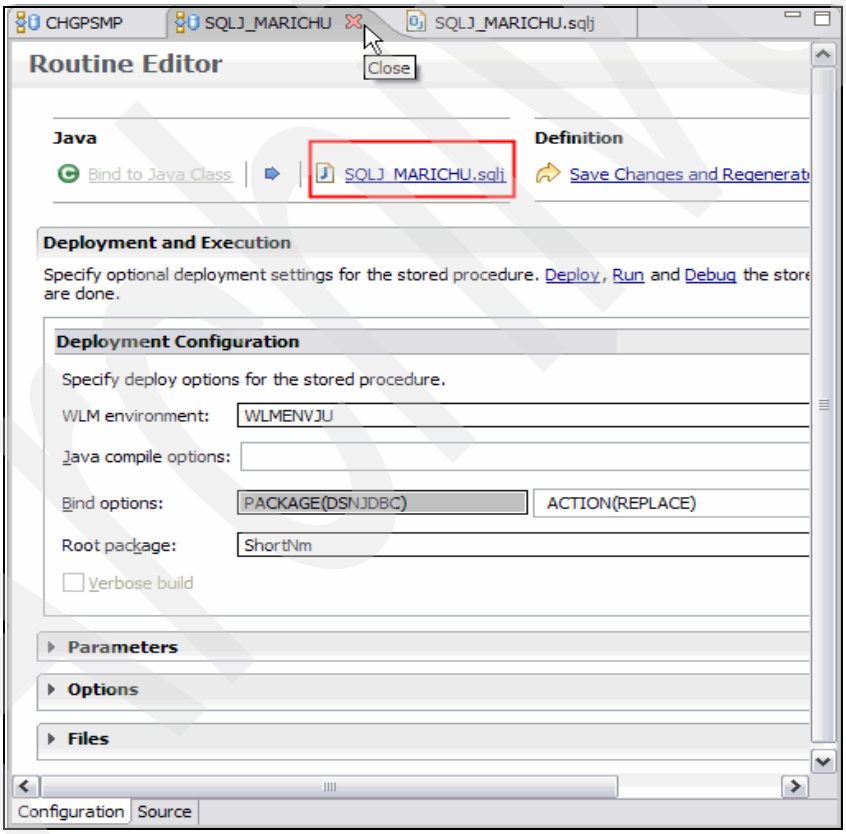

Figure 27-18 Routine Editor - Configuration tab

- From the Java Perspective, Project Explorer, open the package or the default package to the .java or .sqlj file. Double-click this file.
- From the Java Perspective, at the end of the New Java Class wizard, the generated Java class is displayed in the Editor area.

The Java Editor includes features such as the following:

- Syntax highlighting and checking
- Content/code assist
- Code formatting
- Import assistance
- Quick fix

Java routines built by the IBM Data Studio conform to the SQLJ Routines specification. Java objects are defined in the catalog table with LANGUAGE JAVA and PARAMETER STYLE JAVA. Java objects must follow these rules:

- The method that is mapped to the object must be defined as a public static void method.
- The object must receive input parameters as host variables.
- Output and InOut parameters must be set up as single element arrays.

When editing your source in the Java editor, your changes are dynamically compiled and errors reported immediately. When you add arguments to the .java or .sqlj main method, then save the changes, they are reflected as input parameters in the Parameters section of the Configuration tab.

As in SQL stored procedures, changes to the source code of Java stored procedures only change the object in the workspace. To replace the object in the server, redeploy the Java stored procedure.

To close any object, open it in the Editor, click **File** → **Close Object** or **File** → **Close All,** or click the **X** next to the procedure name in the Routine Editor.

Enhanced SQL Editor

This editor is launched when creating or editing SQL statements. We discuss this more in “Creating SQL statements to use in your stored procedure” on page 685.

Export wizard

Use the Export wizard to export routines from your current project to the file system for later deployment. IBM Data Studio supports exporting an entire project or just the stored procedures. You may want to export the entire project to the file system, which can then be imported into another workspace. In this book, we discuss exporting stored procedures only.

You can export a specific stored procedure or several stored procedures at a time. To export routines using the Export wizard:

1. In the Data Project Explorer, select the **Stored Procedures** folder → right-click **Export**.
2. In the Selection page, click the checkboxes for the stored procedures you want to export. You can also click **Select All** to select all stored procedures in this folder. Click **Next**.
3. In the Target and Options page, type the filename and directory where the exported script will be sent. You can optionally click **Browse** to launch the File browser.
4. Click **Next** or **Finish**. The wizard exports the selected routines to the filename and directory that you specified.

Import wizard

Use the Import wizard to import routines to your project. To open the Import wizard:

1. In the Data Project Explorer, select the **Stored Procedures** folder. Right-click **Import.** The Import wizard is launched.
2. In the Import wizard’s Source page (see Figure 27-19), select the location of the object or file that you want to import. You can import from the File System or from another Project in this workspace.

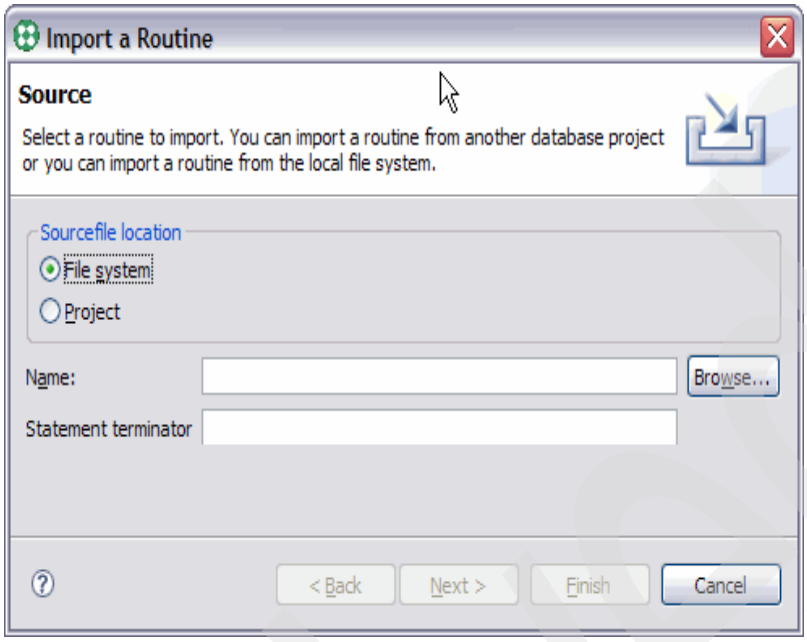

Figure 27-19 Import Wizard

3. Click **Browse** to select the directory or project that contains the stored procedure. A File browser is launched. Click **OK** after selecting the stored procedure.
4. If you are importing an SQL stored procedure, you can optionally set the Statement terminator used in the imported file. Click **Next**.
5. The next page shows the “discovered” entry points for this stored procedure. You can verify whether the imported stored procedure is correct. Click **Next**.

> **Note:** If there are multiple CREATE PROCEDURE statements in the imported file, only the first CREATE PROCEDURE statement is processed and imported by IBM Data Studio.

6. The next page shows the parameters of the imported stored procedure. You can change the parameter data types of imported Java stored procedures. You cannot change the parameters of imported SQL stored procedures. Click **Next**.
7. The next page of the Import wizard allows you to specify import options. You can opt to replace stored procedures with the same name and parameter signature that already exist in the project. Click **Next** or **Finish**.

Deploy wizard

Use the Deployment wizard to deploy routines to a target database. The target database must be compatible with the database for which the object was created.

The wizard consists of four steps. First, you select the target database and enter your user ID and password. Next, you select the routines that you want to deploy. Then, you specify deployment and error handling options. Finally, a summary shows the deployment options that you specified in the wizard.

To deploy routines to a target database using the Deployment wizard, open the Deployment wizard:

1. In the Data Project Explorer, select a project → **Stored Procedures** → select a stored procedure.
2. Right-click this stored procedure and click **Deploy**.
3. Complete the necessary steps of the wizard.
4. Click **Finish**. The wizard deploys the routines to the target database.

Menu and Task Bar

The IBM Data Studio menu bar includes several selections. They are shown in Figure 27-20.

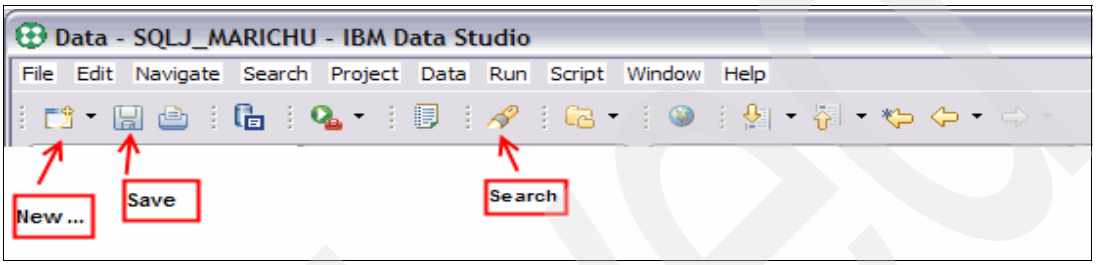

Figure 27-20 Menu and Task Bar

The following are the main selections used in the tooling:

- File - Use this menu to save or close objects that you are currently editing. You can also create new objects from this menu. Eclipse also allows you to switch to a different workspace from this menu.
- Edit - Use this menu to work with the object that you are currently editing.
- Project - Use this menu to open or close a project, and to edit the project's properties. See "Project properties" on page 672 for details on what the properties are.
- Window - Use this menu to open additional IBM Data Studio views. This is also the menu item used to launch the Preferences dialog where you can set various default items. See "Configuring preferences" on page 681 for more details on setting preferences.
- Help - Use this menu to display online help, product information, and to open the Information Center.

27.4 Developing stored procedures with IBM Data Studio

This section describes the steps when developing stored procedures with IBM Data Studio. Here we will discuss:

1. Starting the IBM Data Studio for the first time
   - Configuring preferences
2. Creating a connection
3. Creating a Data Development Project
4. Creating SQL statements to use in your stored procedure
   - Using the SQL Builder
   - Using the Enhanced SQL Editor

In the next section, 27.5, "Developing stored procedures" on page 688, we build on what we've done here and continue on to creating, building, and executing our stored procedure.

27.4.1 Starting the IBM Data Studio for the first time

We start IBM Data Studio by selecting **Start → Programs → IBM Software Development Platform → IBM Data Studio → IBM Data Studio.**

The first time IBM Data Studio is started, a default workspace directory, named workspace, is created for you in C:\Documents and Settings\Administrator\IBM\rationalsdp7.0. You can click **Browse** to launch the File dialog and point to a different directory. The workspace is the main container for all the resources on which you will work.

In our case study, we set the workspace to `C:/SG247083/workspace`, as shown in Figure 27-21.

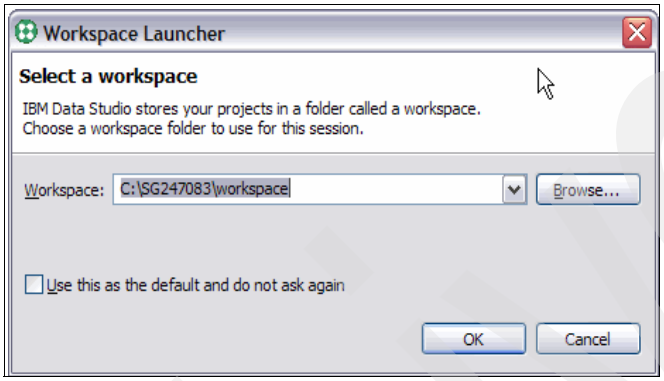

Figure 27-21 Select a workspace

The next window that appears is the IBM Data Studio Welcome window. To close the Welcome window, click the **X** in the Welcome window title bar.

The new window presented is the Data Perspective, which is the default perspective when the IBM Data Studio is launched. You can change the perspective by selecting the title bar **Window → Open Perspective,** and select from the list presented.

Configuring preferences

Each workspace has a set of preferences that is stored in an Eclipse resource. To view the preferences specific to stored procedures, click **Window → Preferences → Data.** Click the following folders to set specific preferences for:

- **Output**
  - Maximum number of rows retrieved and maximum number of bytes to display for character and binary data.
- **Stored Procedures and User-Defined Functions → Deploy Options**
  - The JDK level used when generating and compiling Java stored procedures
  - The location of the SQLJ translator used for translating SQLJ stored procedures
  - Deploy options for Java, External SQL and Native SQL stored procedures
- **Stored Procedures and User-Defined Functions → Process**
  - Autocommit setting
  - Save files after build
  - Set Tracing on
- **Run/Debug → DB2 Stored Procedure Debugger**
  - Session manager information for the Unified Debugger. See 28.2, “The Unified Debugger” on page 738.

We will leave the default values for all of the above preferences for now.

27.4.2 Creating a connection

The New Connection wizard is launched when you click the New Connection icon in the Database Explorer, as shown in Figure 27-8 on page 668. The Connection wizard can also be launched by selecting the **Connections folder** → then right-click **New**. The wizard is also embedded in the Deploy and New Project wizards. The first page of the wizard is shown in Figure 27-22.

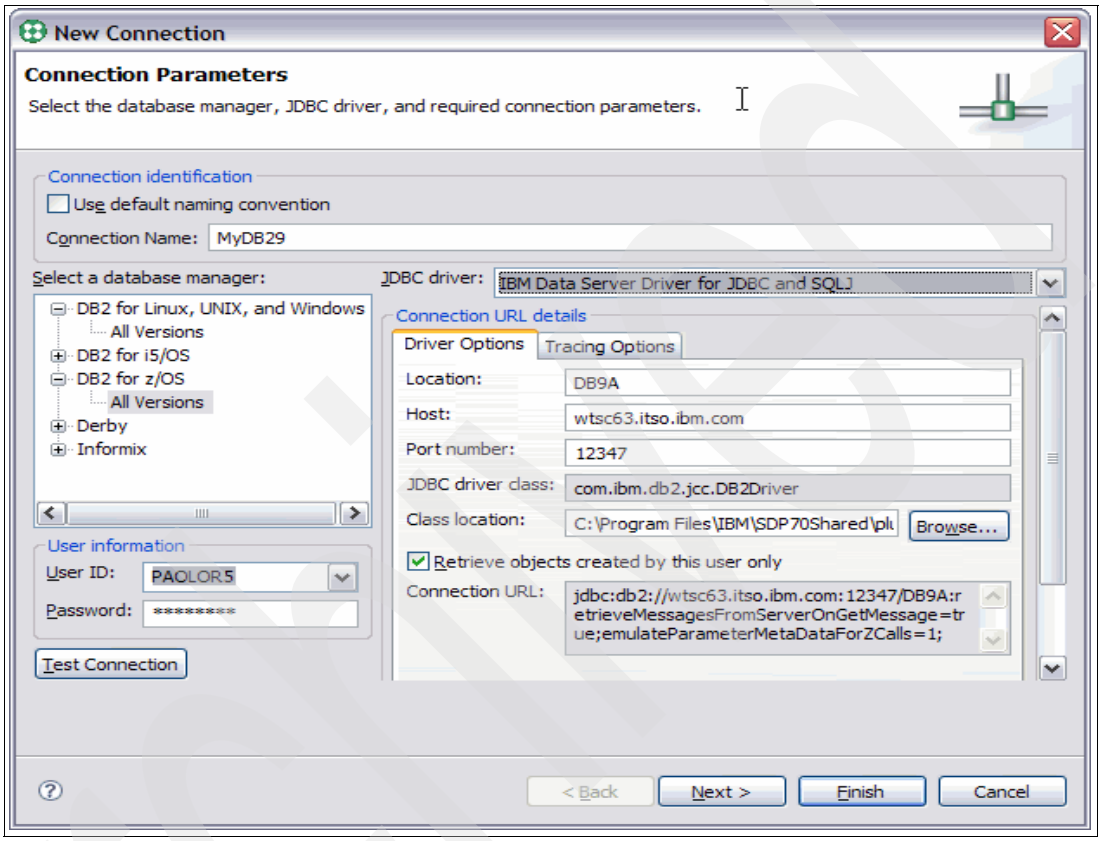

Figure 27-22 New Connection wizard

For this exercise, we connect to the DB9A subsystem. The DDF information for this subsystem is shown in Figure 27-23.

```
-DB9A DIS DDF
DSNL080I  -DB9A DSNLTDDF DISPLAY DDF REPORT FOLLOWS: 458
DSNL081I STATUS=STARTD
DSNL082I LOCATION           LUNAME            GENERICLU
DSNL083I DB9A               USIBMSC.SCPDB9A   -NONE
DSNL084I TCPPORT=12347 SECPORT=12349 RESPORT=12348 IPNAME=-NONE
DSNL085I IPADDR=::9.12.6.70
DSNL086I SQL    DOMAIN=wtsc63.itso.ibm.com
DSNL086I RESYNC DOMAIN=wtsc63.itso.ibm.com
DSNL099I DSNLTDDF DISPLAY DDF REPORT COMPLETE
```

Figure 27-23 Display DDF output

The first page of the wizard contains two tabs, the Driver Options and the Tracing Options.

Driver options

The driver options set the properties of the JDBC driver you wish to use to connect to the server. IBM Data Studio is delivered with a version of the IBM Data Server driver for JDBC and SQLJ[4] in <Install directory>/plugins/com.ibm.datatools.db2_<some version>\driver. The driver and license files are db2jcc.jar and db2jcc_license_cisuz.jar.

The fields in the first page of the wizard allow you to enter the following information:

- **Connection name** - default is the location name.
- **JDBC driver -** default is IBM Universal Driver. IBM strongly recommends using this driver. This driver uses the Server authentication. From the pull-down list, you can change the driver to:
  - IBM Data Server driver for JDBC and SQLJ with Kerberos security, or
  - IBM Data Server driver for JDBC and SQLJ using LDAP, or
  - Other

  If you choose **Other**, you will need to provide the JDBC driver class name, class location, and Connection URL.
- **Location -** enter the DB2 for z/OS location ID.
- **Host -** enter the domain or FTP address of your DB2 for z/OS server.
- **Port number -** enter the port number of your DB2 for z/OS server.
- **JDBC driver class -** when using the IBM Universal driver, this is pre-filled with the value `com.ibm.db2.jcc.DB2Driver`.
- **Class location -** when using the IBM Universal driver, this is pre-filled with the location of the license jar files installed with your IBM Data Studio.
- **Retrieve objects created by this user only -** check this box if you want to work only with objects that you created.
- **Connection URL -** IBM Data Studio composes this as you enter values for the location, host, and port number. It additionally adds some default JDBC properties.
- **User ID -** enter your DB2 for z/OS login authorization ID.
- **Password -** enter the password associated with the above user ID.

The Test Connection button allows you to test the connection using the fields you entered.

Tracing options

The JDBC tracing options can be set in the connection URL by selecting the trace levels in this tab of the New Connection wizard. For more information on tracing levels and other problem determination tools for IBM Data Studio, check:

`http://www.ibm.com/developerworks/db2/library/techarticle/dm-0706scanlon/`

The second page of the wizard allows you to filter the schemas that will be loaded into the connection. IBM Data Studio performs on-demand loading. The objects are loaded from the catalog only when the object folder is opened (double-clicking the folder).

Filter connection

The next page of the New Connection wizard allows you to filter the schemas that will be loaded into your connection. You can filter using a Where clause-like expression or select from a list. Figure 27-9 on page 669 shows the filter page.

Click **Finish** to complete creating the connection.

[4] Also known as the IBM Universal driver.

Editing the connection

After you create the connection, you can modify the connection properties by selecting the connection and right-clicking **Edit connection**.

When you upgrade Data Studio to a new release, or when migrating from Developer Workbench to Data Studio, the connection properties from your workspace may need to be updated to use the Universal license jars in the new release. A typical error that you may get when connecting is:

```
Connection has failed. The following error was reported:
----------------------------------------------------------
java.io.FileNotFoundException:
C:\Program\IBM\SDP70Shared\plugins\com.ibm.datatools.db2_1.0.100.v200707172230\driver\db
2jcc.jar

Do you want to work offline?
----------------------------------------------------------
```

Edit the connection properties and point the JDBC driver class location to the new install directory's driver folder.

27.4.3 Creating a Data Development Project

Click **File** → **New** → **Data Development Project** to launch the New Data Development Project wizard, shown in Figure 27-24. The wizard contains three pages for setting the following:

- **Project Name** - Type `Project_7083` for the project name.
- **Current schema setting** - You can select to use the server's login ID as the current schema or set the current schema to another value. You can also opt to omit the schema from generated SQL statements. SQL statements that you create using the Editor are not affected by the Current schema testing. We will use the default.

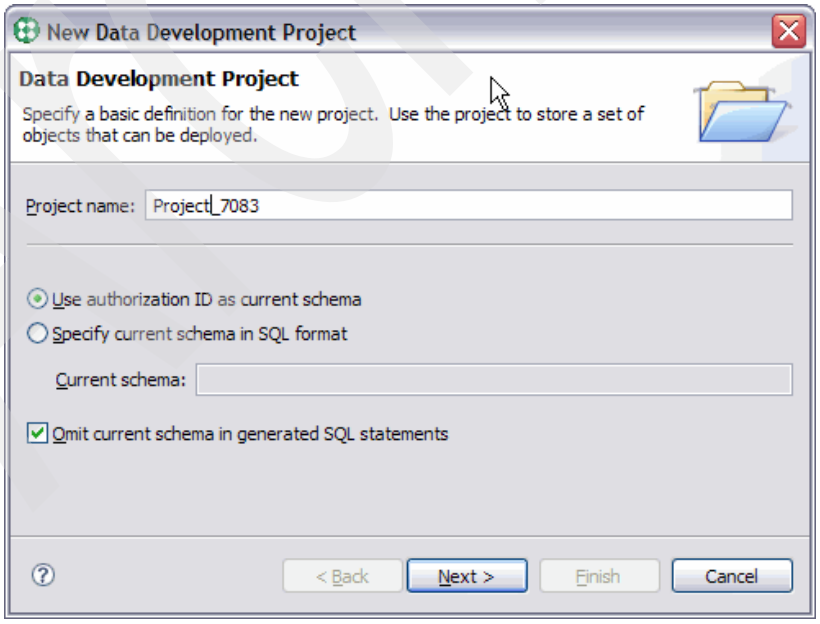

Figure 27-24 New Data Development Project

- **Target connection -** The wizard's second page lists all the active connections available. The user can opt to create a new connection. The New Connection wizard will be launched.
- **Package Owner -** Enter a valid primary or secondary authorization ID as the package owner. If blank, the current login ID is used.
- **Build Owner -** Enter a valid primary or secondary authorization ID as the owner of the stored procedure created. This is the value in the OWNER column in SYSROUTINES. If blank, the current ID is used.

Clicking **Finish** in the wizard creates a data development project. Figure 27-12 on page 672 shows you the folders contained in a data development project.

27.4.4 Creating SQL statements to use in your stored procedure

Before we develop our stored procedures, we can optionally develop our SQL statements first, then import them into the stored procedure. We will see later that we can optionally create a stored procedure directly from an SQL script. An SQL script can contain one or more SQL statements.

IBM Data Studio provides you with three tools for developing SQL statements:

- SQL Builder - This is a graphical builder used for creating SELECT, INSERT, UPDATE, DELETE, Full SELECT and WITH statements.
- Enhanced SQL Editor - This is a rich text editor that can handle both DML and DDL statements. It has colorization and content assist capabilities.
- SQL Wizard - This is embedded in the New SQL Procedure wizard and is similar to SQL Assist in Development Center.

Using the SQL Builder

Right-click the SQL Scripts folder → **New** → **SQL** or **XQuery** to launch the New SQL Statement wizard. Figure 27-25 is the dialog that allows you to select whether to create the SQL statement with either the SQL Editor or SQL Builder. Select the latter.

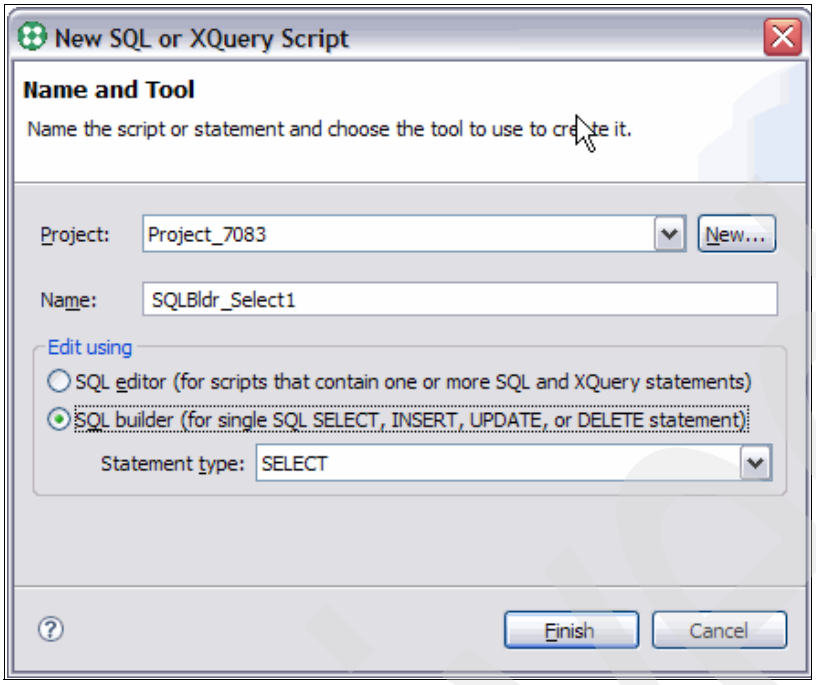

Figure 27-25 New SQL or XQuery Script

The SQL Builder populates the Editor view with three panes: the SQL statement view, as is shown in Figure 27-26.

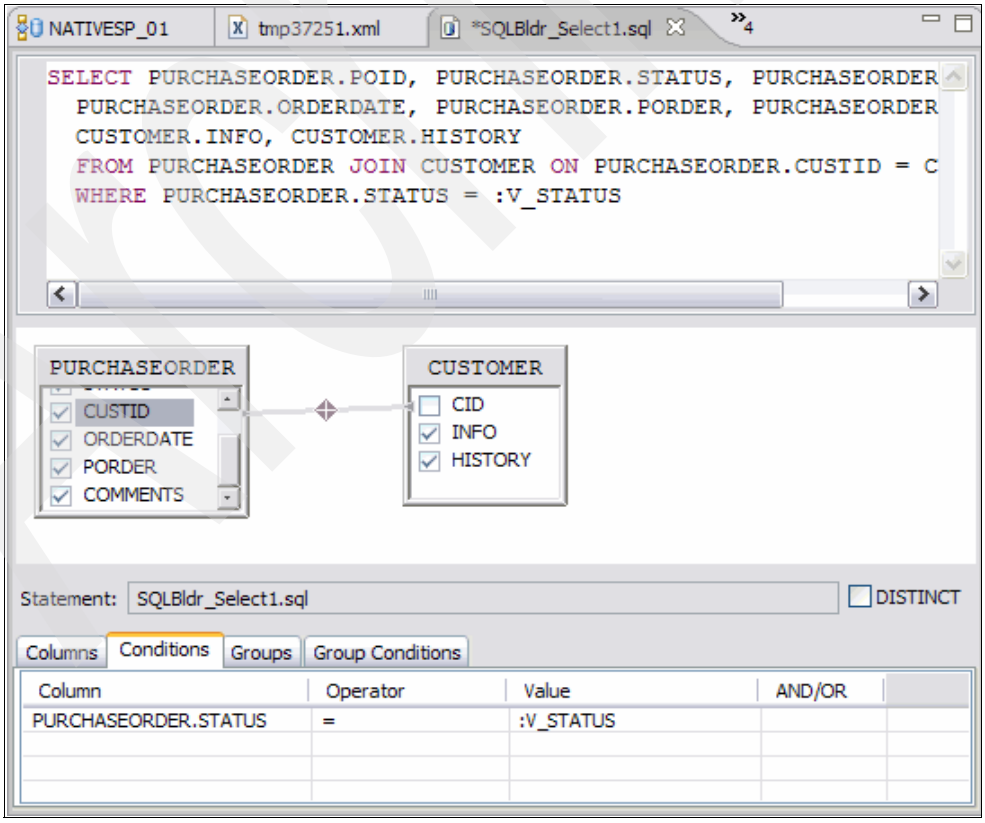

Figure 27-26 SQL Builder

To build the illustrated statement in the SQL Builder, do the following:

- Right-click the graphical view (middle pane) → **Add table**. Expand the **DEVL7083** schema and select the table **PURCHASEORDER**. Add the table CUSTOMER in the same manner. This adds the selected tables to the FROM clause of the statement.
- Still in the graphical view, click **CUSTID** in table PURCHASEORDER and drag to CID in table CUSTOMER. This creates an inner join between the two tables. You can right-click the join line and select **Specify the join type** to select another type of join, such as a left outer join.
- Click the check boxes for POID, STATUS, CUSTID, ORDERDATE, PORDER and COMMENTS in table PURCHASEORDER. This adds those columns to the SELECT clause of the statement.
- Click the check boxes for INFO and HISTORY in table CUSTOMER.
- Click the **Conditions** tab. Click the first cell, and a pull-down arrow is displayed. Scroll down the list and select **PURCHASEORDER.STATUS**.
- Click the Value column and type `:V-STATUS`. This creates a variable in the statement. When you run the query, you will be prompted to provide a value for this variable.
- Click **Save**.

Using the Enhanced SQL Editor

In the New SQL or XQuery Statement dialog shown in Figure 27-25 on page 686, you can opt to use the Enhanced SQL Editor by selecting the SQL Editor button. The Editor will be started with a blank page in which you can enter any SQL statement.

The Enhanced SQL Editor has the following features:

- Multiple statement support

  You can type multiple statements in an .sql file and then run them.

- Variable statement terminator

  The statement terminator is actually a statement separator. By default, the SQL editor uses a semicolon (;). You can specify a different statement terminator for the statements that you create in the Enhanced SQL Editor. You don't need to specify a statement terminator for the last (or only) statement in your script.

- Syntax highlighting

  To aid you in differentiating the elements in an SQL statement, syntax highlighting renders different kinds of elements in the text in unique colors.

- Content assist

  Content assist is an editing tool that provides you with helpful information as you type an SQL or XQuery statement. For example, after you type the dot that follows a schema qualifier in an SQL statement, content assist supplies a list of the tables in the schema.

  As you develop your statement, you can press Ctrl + Space at any time to see a list of available choices in the content assist window. You can filter the list of choices by typing a character, and only the choices beginning with that character will be shown.

- Query parsing and validation

  As you type, the parser checks the syntax of both SQL and XQuery expressions, and provides a visual indication of any errors that it detects in the query.

In IBM Data Studio, you can also use or create code templates. This allows you to skip typing parts of an SQL statement and tab into input fields within the templates. An example of a template is shown in Figure 27-27 on page 688.

To use this template and create the SQL statement in the Enhanced SQL Editor, we followed the following steps:

- From the SQL Scripts folder, right-click → **New** → **SQL** or **XQuery** script.
- Type `SE_Select1` for the statement name. Click **Edit using the SQL Editor button**.
- A blank editor page is created. Press Ctrl + Space to activate Content Assist.
- Type `S` to see template choices that start with S. Select the **SELECT - SELECT statement with two columns template**.
- In the editor the cursor is placed in col1. Type `PURCHASEORDER.POID.`
- Tab to col2. Type `CUSTOMER.INFO`.
- Tab to table1. Type `PURCHASEORDER`.
- Tab to table2. Type `CUSTOMER`.
- Press Esc to return to the normal editing mode.

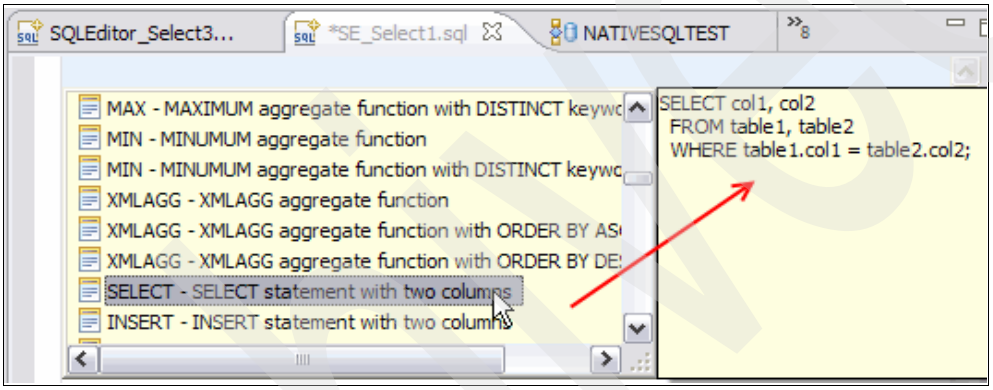

Figure 27-27 SELECT template with two columns

You can create your own template in the Preferences of IBM Data Studio. In the menu bar, click **Window** → **Preferences** → **Data** → **SQL Editor** → **Templates**.

Whether you use the SQL Builder or the SQL Editor, your created scripts will be saved in the SQL Scripts folder.

Creating a stored procedure from a script

You can create a stored procedure straight from an SQL script in the SQL Scripts folder. Select the SQL script → right-click **Generate stored procedure**. The New Stored Procedure wizard is launched.

27.5 Developing stored procedures

In this section and the next, we discuss creating, deploying, and executing a stored procedure.

You can create a stored procedure in two ways:

- Creating a stored procedure from scratch using the wizards
- Importing a previously written stored procedure from another project or file system

So, in this section, we discuss:

1. Creating a new native stored procedure using the wizard
   - Use the new stored procedure wizard
   - Use the new version wizard
   - Use the Import wizard
   - Create a stored procedure from an SQL script
2. Creating an external SQL stored procedure from the wizard
   - Creating an SQL stored procedure on DB2 for z/OS V8 for debugging
3. Creating a Java stored procedure from the wizard
   - JDBC
   - SQLJ
4. Importing an SQL stored procedure
5. Importing a Java stored procedure
   - JDBC
   - SQLJ
6. Deploying a stored procedure
   - Deploy to the current or a different server
   - Duplicate and error handling options
   - Deploying nested or dependent stored procedures
   - Setting the JDK level for Java stored procedures
   - Setting the bind options
   - Setting SQLJ options
   - Enabling debug
7. Executing a stored procedure
   - Run the Settings dialog
   - Output view status and messages
   - Result sets

27.5.1 Creating a new native stored procedure using the wizard

On the Stored Procedures folder, right-click **New.** The New Stored Procedure wizard is launched. Six pages for SQL, and seven pages for Java stored procedures, make up the wizard. Figure 27-28 on page 690 shows the first page of the wizard.

The pages of the wizard allow you to specify properties that are either common or specific to a new Native SQL, External SQL, dynamic Java, or static Java stored procedure. In this section, we start with creating a new native stored procedure. We will investigate the differences for external SQL and Java stored procedures in later sections.

Name and language

- **Project -** This is set to the project from which the New Stored Procedure was launched.
- **Name -** Type `NativeSQLTest` as the name for this new stored procedure.

> **Note:** IBM Data Studio accepts upper and lower case schema.procname. However, when the SQL or Java stored procedure is built, both schema and procname are converted to uppercase in the DB2 catalog on z/OS. To enter lowercase, enclose the name in quotation marks, for example “MyProc”.

- **Version -** This field is initially set to Version1. If the language is changed to SQL- External or Java, this field is disabled.

  Note: In DB2 for z/OS, the default version for native SQL procedures is V1.

- **Language -** This field is initially set to SQL - Native. Click the pull-down list to change to SQL - External or Java. When the language is set to Java, the Java options box is enabled. We will discuss Java options in 27.5.3, “Creating a Java stored procedure from the wizard” on page 696.

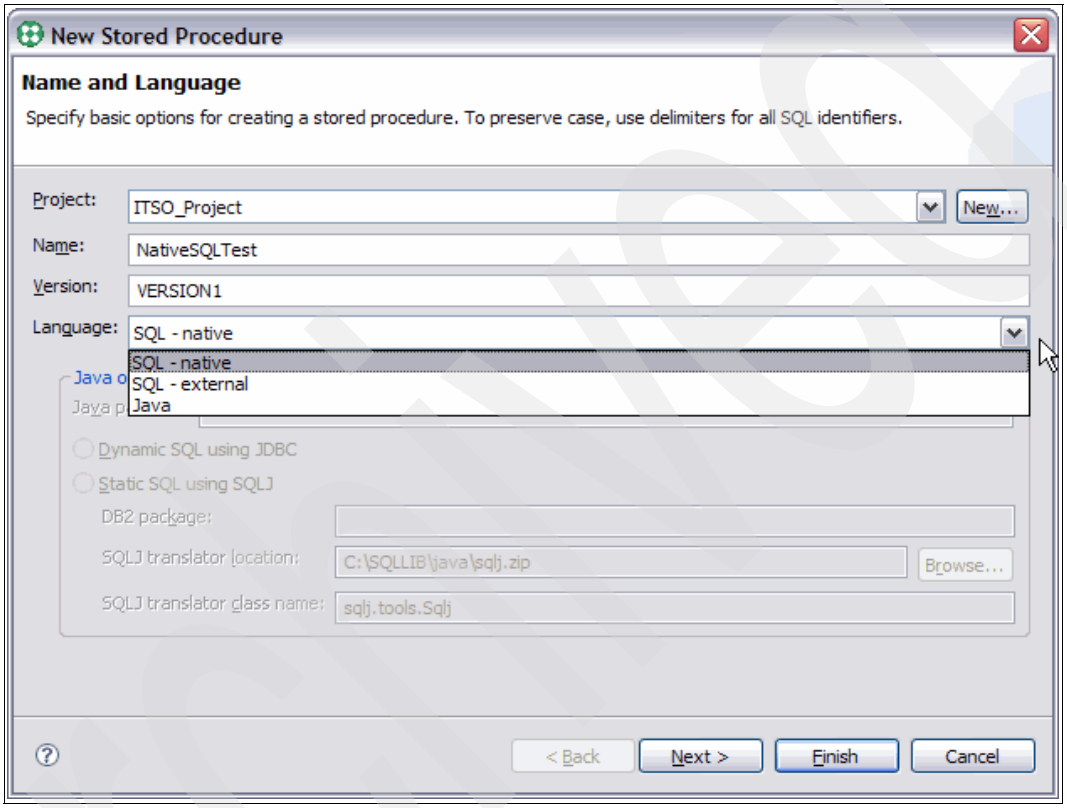

Figure 27-28 New Stored Procedure wizard

- Click **Next**.

SQL statements

A default SQL statement for testing setup is included in the SQL statement on the SQL Statements page. The statement is `SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES` (see Figure 27-29 on page 691). We will use the default settings on this page. However, we discuss each of the UI elements below.

In the Statements list box, you can:

- **Add** additional SQL statements to be processed in the stored procedure. Each statement is labeled StatementN where N is a sequence number.
- **Remove** SQL statements from the Statements list.
- **Import** a previously created SQL statement from the SQL Scripts folder of this project.
- **Show All** statements that will participate in this stored procedure.

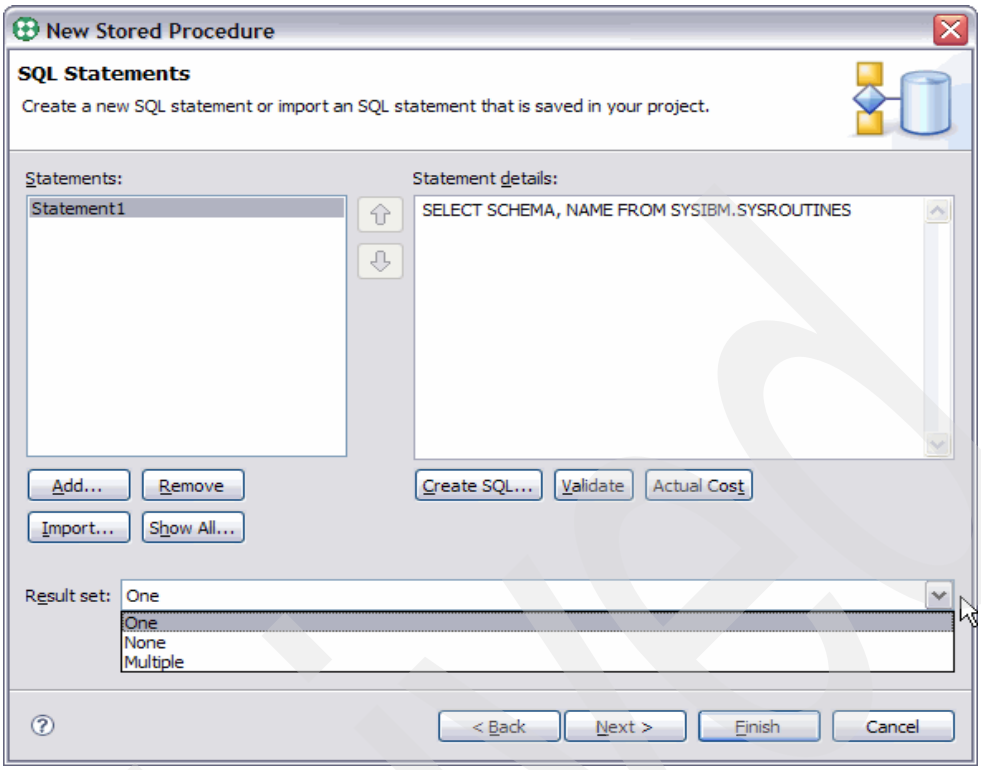

Figure 27-29 SQL Statements page

- **Result set** allows you to specify whether the stored procedure returns One, Multiple, or no (None) result sets.

> **Note:** If you have multiple queries and specify only One result set, the tooling will generate a Case statement with each statement being a case. An input variable, whichQuery, is also generated to determine which case/query will be executed.

- **Create SQL** will launch the SQL wizard. This is discussed further in 27.4.4, “Creating SQL statements to use in your stored procedure” on page 685.
- **Validate** will cause the current statement in the Statement details text box to be parsed and validated for syntax errors.
- **Actual Cost** will call the DB2-supplied stored procedure DSNWPSM and return a cost value for executing the statement.

> **Note:** You need to set up this support in DB2 for z/OS beforehand. See 27.2.5, “IBM Data Studio Actual Costs setup” on page 659.

- **Visual Explain** (V8 only) - This launches the Visual Explain for DB2 for z/OS V8 tool. A graphical display of the access path is shown in a separate dialog. This product is a separate install. The first time you click **Visual Explain**, you will be asked if you want to download this product from the Web.

Parameters

In this page, you can:

- Specify the following errors:
  - SQL Exception

- SQL Message
- SQLSTATE and SQL Message
- SQLCODE and SQL Message
- SQLSTATE, SQLCODE and SQL Message

- Add, remove and reorder Parameters. The Add Parameter dialog shown in Figure 27-30 allows you to specify the parameter properties.

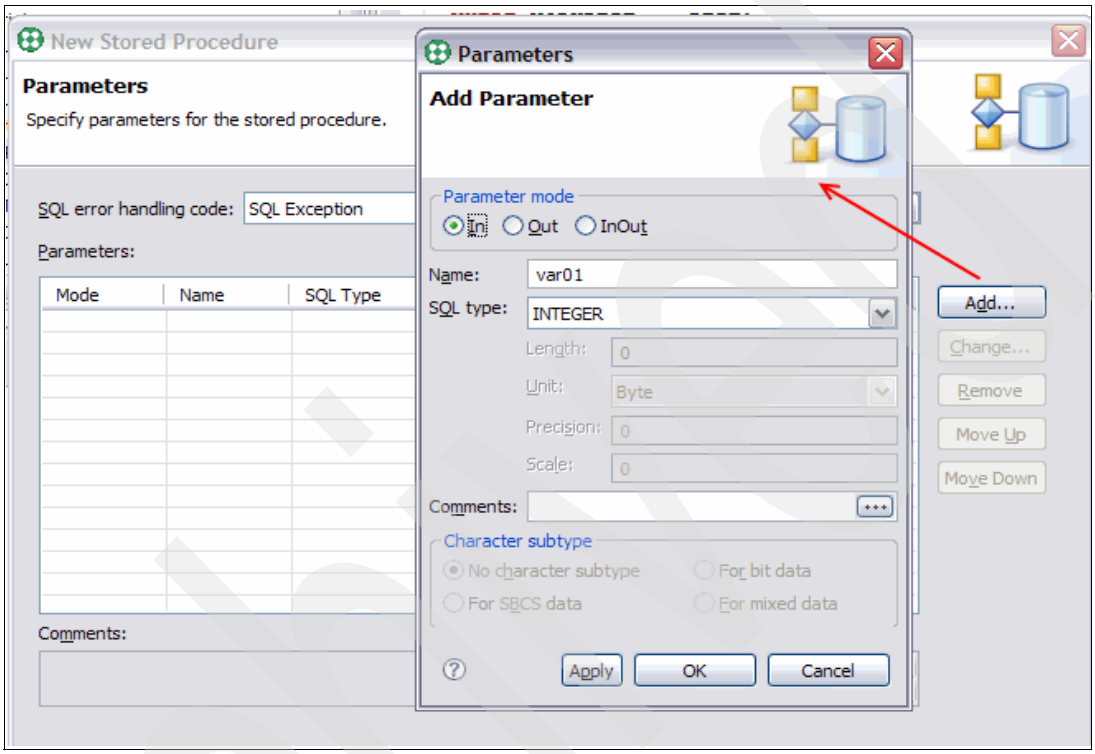

Figure 27-30 Parameters page and Add Parameter dialog

Note: DB2 for z/OS V9 does not support passing parameters of XML data types.

We don't have any parameters, so we skip this page. Click **Next.**

Deploy options

In this page, we specify the options for deploying and/or debugging the stored procedure.

- **Deploy On Finish** - Leave this unchecked. We want to deploy the stored procedure after we examine the generated stored procedure code.
- **Current Schema -** This is enabled and initialized with either the login authorization ID or the Project properties' current schema setting. See 27.7.5, "Behavior when setting the Current Schema project property" on page 717 for more information on this.
- **Enable Debugging -** This checkbox triggers building the stored procedure for debug. Check this to debug this stored procedure later on. For native SQL stored procedures, the keywords ALLOW DEBUG MODE and WLM ENVIRONMENT are generated in the CREATE PROCEDURE DDL. The default WLM used is whatever is specified in the Preferences (see "Configuring preferences" on page 681).
- **Advanced** - This launches the z/OS Options dialog. For native SQL stored procedures, this is shown in Figure 27-31 on page 693.

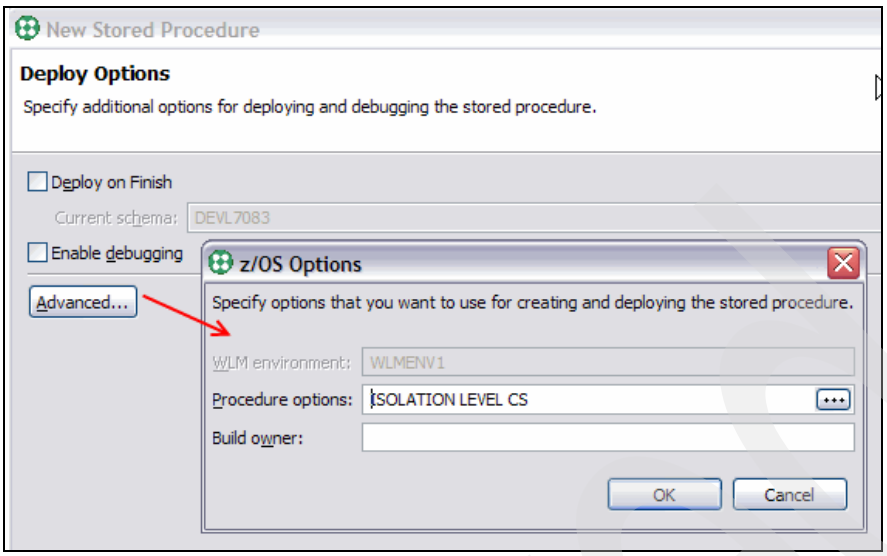

Figure 27-31 Deploy options page and z/OS Options dialog

Since native SQL stored procedures do not need a WLM environment unless it is enabled for debug, the Advanced button in the Deploy Options page has the WLM Environment field disabled.

Note: Although native SQL stored procedures do not require you to specify a WLM environment, you still need to specify a default WLM Environment in your DSNZPARM.

- **Procedure Options -** You can optionally update or add procedure options from the Advanced dialog.

Code fragments

Like Development Center (DC), IBM Data Studio gives you the ability to import code fragments into your stored procedure. In DC, you could only import one file at a time. IBM Data Studio allows you to import multiple code fragments. The code is concatenated before inserting into the appropriate area of the stored procedure. In 27.5.3, “Creating a Java stored procedure from the wizard” on page 696, we will see how to insert code fragments into a Java stored procedure.

Summary

This page (shown in Figure 27-32) summarizes our SQL stored procedure. Optionally, we can view the SQL procedure definition by clicking **Show SQL**.

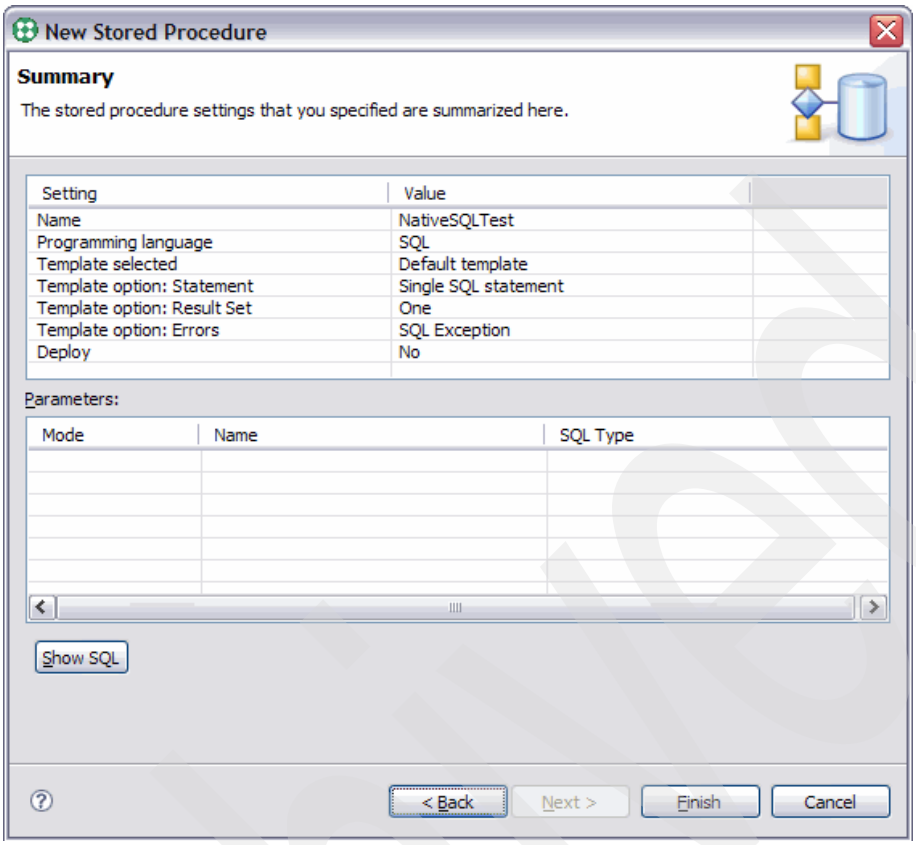

Figure 27-32 Native SQL stored procedure summary info including procedure definition

27.5.2 Creating an external SQL stored procedure from the wizard

We created a native stored procedure from the wizard in 27.5.1, “Creating a new native stored procedure using the wizard” on page 689. The steps to create a new external SQL stored procedure from the wizard are similar. Here we will highlight the differences.

In Data Project Explorer, expand Project_7083, right-click the **Stored Procedures** folder and then select **New** → **Stored Procedure**. This launches the New Stored Procedure wizard.

- In the Name and Language page (see Figure 27-28 on page 690), do the following:
  - Type `EXTSQLSP`.
  - Select **SQL - External** as Language.
  - Note that Version is grayed out.
  - Click **Next.**
- Click **Next** two more times until you get to the Deploy Options page.
- Click **Advanced**. The z/OS Options dialog is launched. This dialog contains two tabs (see Figure 27-33 on page 695 and Figure 27-34 on page 696). In our case study, we kept the default values in this dialog, except for the WLM environment.

  Since this dialog contains additional fields when compared to a native SQL stored procedure, we examine it further.

  In the Stored Procedure Options tab, we do the following:

  - *Runtime options* - Leave the default value `NOTEST(NONE,*,*,*)`.
  - *WLM Environment* - This is required for external SQL stored procedures. Type `DB9AWLM`.
  - *ASU time limit* - Leave the default as 0.

– *Stay Resident* - Do not check this checkbox. Checking this checkbox generates the keyword STAY RESIDENT YES. The default is STAY RESIDENT NO.
– *External Security* - The radio button `DB2` is pre-selected and is the default.

In the Deploy Options tab, we use all default settings. If you like, you can specify the following:

– *Build Utility* - This defaults to the value in the Preferences page.
– *Build Owner* - Specify the authorization ID that will be the owner / definer of this external SQL stored procedure. See 27.7.6, “Package owner and Build owner” on page 717 for more information on this field.
– *Precompile options* - This defaults to `MAR(1,80)`.
– *Compile options* - This defaults to `NOTEST(block,noline,nopath)`.
– *Prelink options* - Specify any prelink options, default is blank.
– *Link options* -Specify any prelink options, default is blank.
– *Bind options* - The bind options are specified in two parts. The `PACKAGE` area is read-only and defaults to `PACKAGE(collid)` where collid is the Collection ID specified in the Deploy Options preference page.

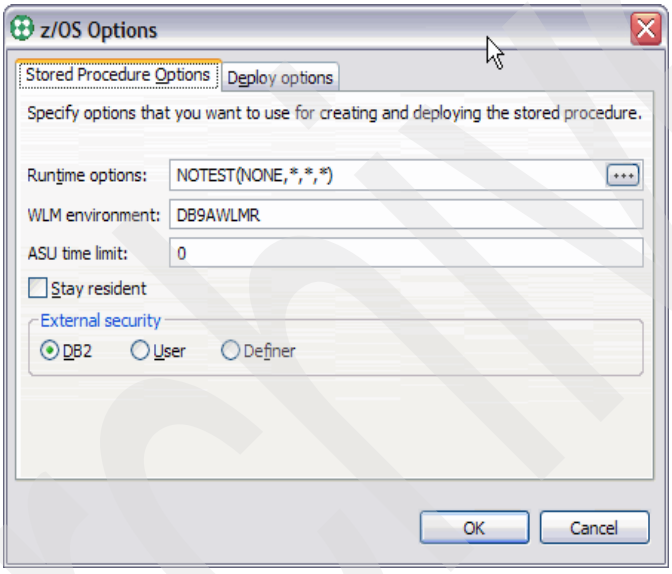

Figure 27-33 Stored Procedure options

- Click **OK** to close the z/OS Options dialog.
- Click **Finish** to complete creating the external SQL stored procedure.

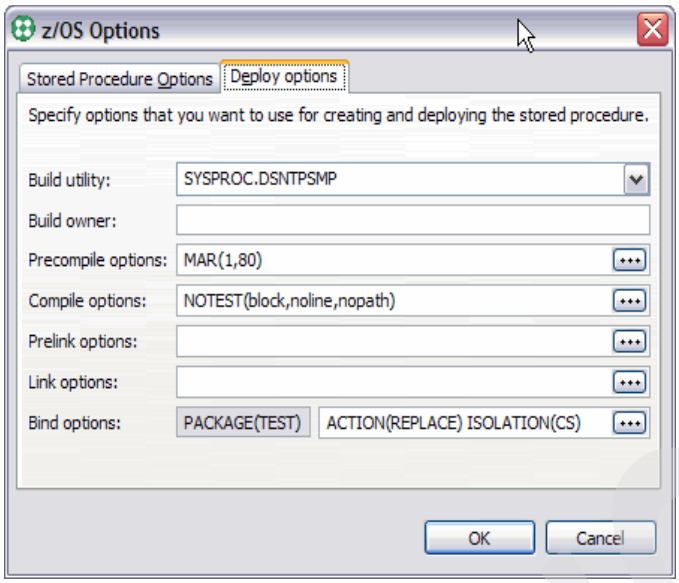

Figure 27-34 External SQL deploy options

Import the source of a stored procedure from a file

You can use the Import wizard to copy the SQL stored procedure from the file system or another project to the current project. See our example of the Import wizard for an SQL stored procedure in 27.5.4, “Importing an SQL stored procedure” on page 701.

27.5.3 Creating a Java stored procedure from the wizard

Creating a Java stored procedure from the New Stored Procedure wizard has similar pages to those used when creating SQL stored procedures. In Data Project Explorer, expand Project_7083, right-click the **Stored Procedures** folder and then select **New** → **Stored Procedure**. This launches the New Stored Procedure wizard.

The wizard will guide you through the following pages to create the Java stored procedure.

Name and Language

- Type `DEVL7083.JAVATEST` for the Name.
- Select **Java** for Language.
- In the Java Options box area, type `COM.IBM.TEST` for Java package.
- Select **Static SQL using SQLJ**.
- If DB2 for LUW is installed in the client, IBM Data Studio will set the SQLJ translator location and SQLJ Translator class name to the translator supplied by DB2. Otherwise, you can click **Browse** to specify the SQLJ translator class location in your file system.
- The completed page is shown in Figure 27-35 on page 697. Click **Next.**

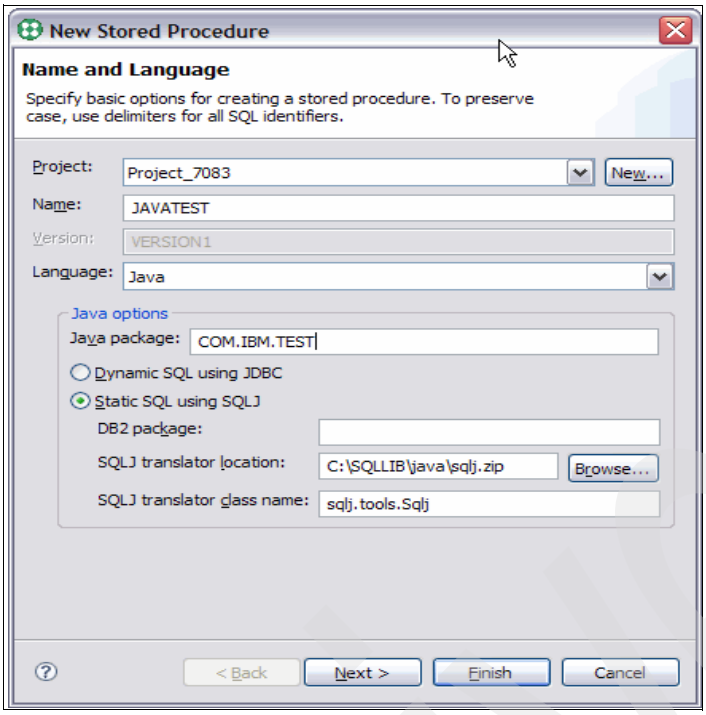

Figure 27-35 New Java stored procedure (SQLJ)

SQL statements

A default SQL Statement for testing is generated in the SQL statement details text box. The statement is `SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES`. See “SQL statements” on page 690 for more information about this page.

Use the defaults for this page. Click **Next.**

Parameters

In the SQL Error Handling code: select the `SQLState, SQLCode and SQLMessage` from the pull-down list. This generates three output parameters, as shown in Figure 27-36.

Click **Next.**

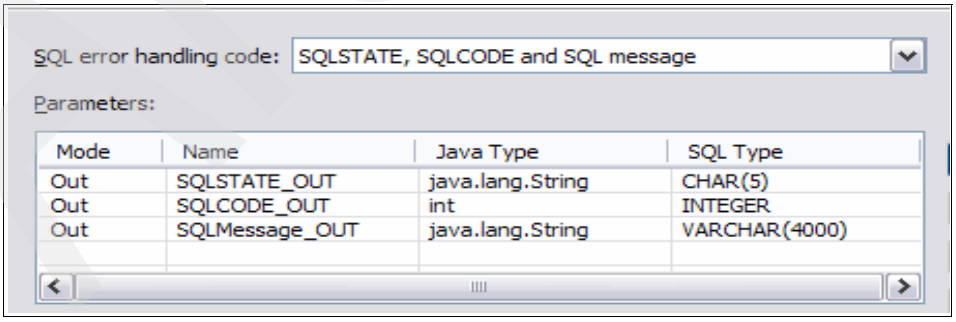

Figure 27-36 Error Handling code

Deploy options

The Java stored procedure deploy options differ slightly from SQL stored procedures, as shown in Figure 27-37 on page 698.

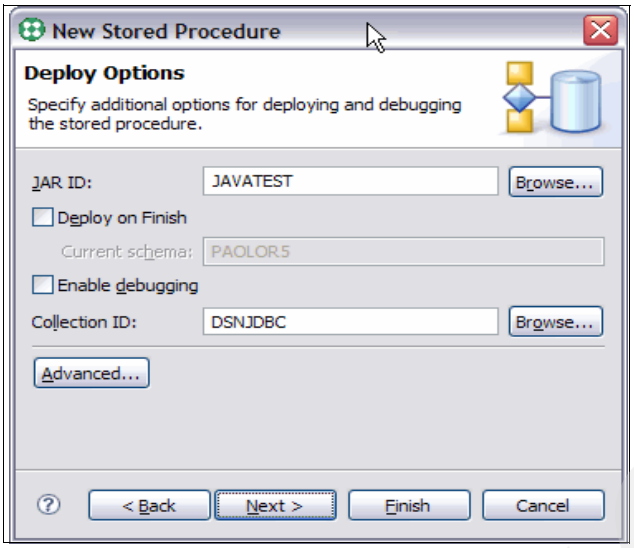

Figure 27-37 Java stored procedure deploy options

- The Jar ID is automatically filled in with the name of the stored procedure. We override it by typing `DEVL7083.JAVATEST` in our schemaname.procname.
- We can automatically deploy this stored procedure after clicking **Finish**. We will deploy later, so leave the Deploy on Finish button unchecked.
- Check **Enable Debugging** to enable debugging on this stored procedure.
- The Collection ID that is specified must include the JDBC drivers on z/OS. The JDBC drivers are bound into DSNJDBC. For JDBC stored procedures, this defaults to `DSNJDBC`. For SQLJ stored procedures, we can change this to a collection ID that is bound to the JDBC drivers. We will use the default value, `DSNJDBC`.
- Click **Advanced**. This displays the Z/OS Options dialog.
- Since we updated our default WLM environment for executing Java stored procedures to point to our DB9A WLM AE, `DB9AWLMJ`, we do not have to make changes to the Stored Procedure Options tab.
- Click the **Deploy Options** tab.
- Check the Enable Debugging checkbox. The `-g` compile option is automatically added.
- IBM Data Studio generates a default root package name. Change the default value to `SQLJTST`. DB2 will generate four packages for the stored procedure, where the package name is the root package + 1, 2, 3, and 4. See Figure 27-38.

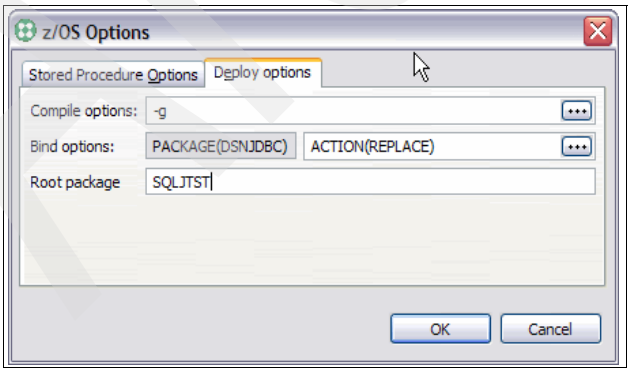

Figure 27-38 SQLJ stored procedure root package and compile options

Note: In DB2 for z/OS V8 and V9, you can generate single packages by specifying the ISOLATION LEVEL in the Bind Options. In DB2 for z/OS V9, you can also specify a package name greater than 7 characters.

Java Path

You can specify additional jar files that your stored procedure will use. We will discuss this in more detail in 27.7, “Advanced IBM Data Studio topics” on page 710.

Code fragments

To include the sample code fragments, launch the file browser for each of the code fragments shown in Figure 27-39.

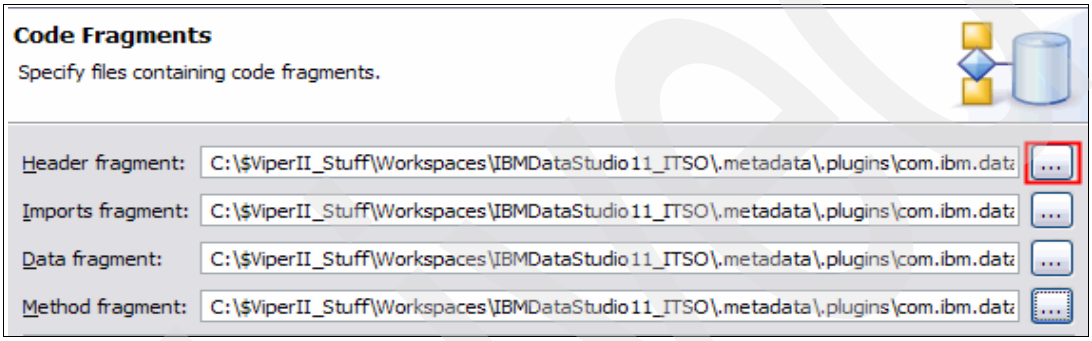

Figure 27-39 Specifying code fragments

- Header fragments - Include any file with additional header statements you want included in this stored procedure. This file is placed before the `package` statement.
- Import fragments - Include any file with additional import statements you want included in this stored procedure. This file is placed after any generated import statements.
- Data fragments - Include any file with data definitions you want included in this stored procedure. This file is placed after the `Public class` statement
- Method fragments - Include any file with additional methods you want included in this stored procedure. This file is included at the end of the generated code.

Note: The multi-file code fragments support is in Fix Pack 1 of IBM Data Studio.

Summary

This page summarizes our Java stored procedure. The generated code for this example is shown in Example 27-10.

Example 27-10 Example of generated SQLJ code using fragments

```
/**
 * SQLJ Stored Procedure JAVATEST
 * @param SQLSTATE_OUT
 * @param SQLCODE_OUT
 * @param SQLMessage_OUT
 */
/**
 * Header fragment inserted from SP_JAVA_HDR.FRAGMENT
 */
package marichu;
```

```
import java.sql.*; // JDBC classes
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql context SPContext;
/**
 * Imports fragment inserted from SP_JAVA_IMPORT.FRAGMENT
 */

#sql iterator JAVATEST_Cursor1 ( java.lang.String, java.lang.String );

public class JAVATEST {
/**
* Member data fragment inserted from SP_JAVA_MBR.FRAGMENT
*/

public static void jAVATEST(java.lang.String[] SQLSTATE_OUT,
   int[] SQLCODE_OUT, java.lang.String[] SQLMessage_OUT,
   ResultSet[] rs1) throws Exception {
   JAVATEST_Cursor1 cursor1 = null;
   SPContext ctx = null;
   try {
      ctx = new SPContext("jdbc:default:connection", false);
      #sql [ctx] cursor1 =
      {
         SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES
      };
      rs1[0] = cursor1.getResultSet();

      // Set return parameters
      SQLSTATE_OUT[0] = SQLSTATE_OUT[0];
      SQLCODE_OUT[0] = SQLCODE_OUT[0];
      SQLMessage_OUT[0] = SQLMessage_OUT[0];
   } catch (SQLException e) {
   // Set return parameter

   // Close open resources
   try {
      if (cursor1 != null)
         cursor1.close();
         if (ctx != null)
         ctx.close();
   } catch (SQLException e2) { /* Ignore */
   }
   }
}

   /**
    * Methods fragment inserted from SP_JAVA_MTHD.FRAGMENT
    */
}
```

Import the source of an existing stored procedure

You can use the Import wizard to copy the Java source file from the file system or another project to the current project. See our example of the Import wizard for a Java stored procedure in 27.5.5, “Importing a Java stored procedure” on page 703.

27.5.4 Importing an SQL stored procedure

Importing an SQL stored procedure is similar to importing a Java stored procedure. See 27.5.5, “Importing a Java stored procedure” on page 703 for details on using this wizard.

When importing an SQL stored procedure from the file system to a project that is targeting a DB2 V9 for z/OS, if the imported DDL does not contain the `FENCED` or `EXTERNAL` keyword, IBM Data Studio constructs the imported SQL stored procedure as a Native SQL stored procedure.

Right-click the **Stored Procedures** folder and then select **Import** to launch the Import wizard.

- Source
  - In the Sourcefile Location area (see Figure 27-40 on page 701), click **Browse** to the right of the Name field to locate the source EMPDTLSS.ddl that we had previously saved on our workstation.
  - Click **Open** to import this file.
  - Click **Next.**

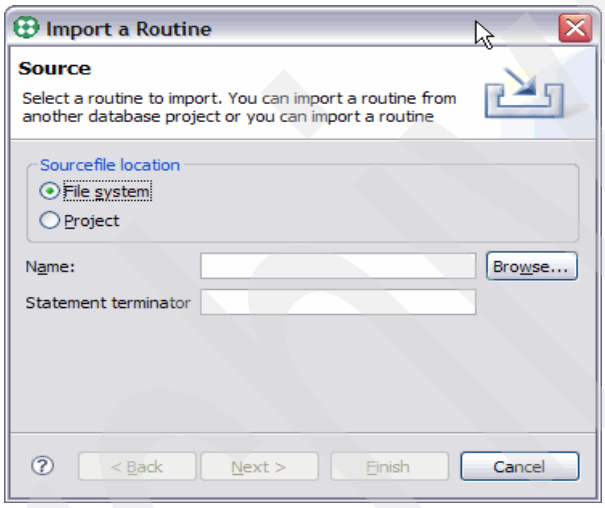

Figure 27-40 Import Wizard, Source page

- Entry Points
  - The EMPDTLSS entry point is the CREATE PROCEDURE itself. See Figure 27-41 on page 702.
  - Click **Next**.

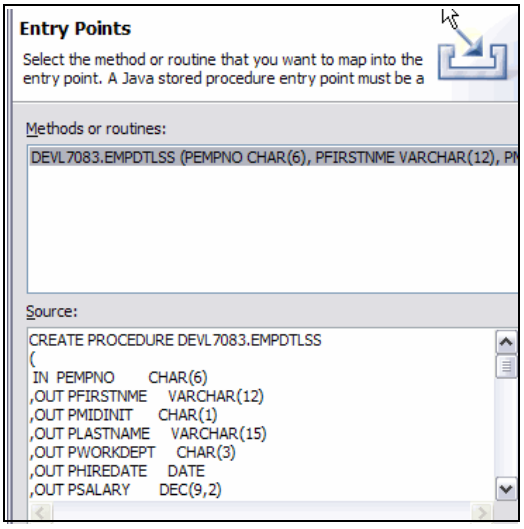

Figure 27-41 Import Wizard, Entry points

- Parameters
  - On this page, one input parameter and seven output parameters are listed, as shown in Figure 27-42.
  - Click **Next.**

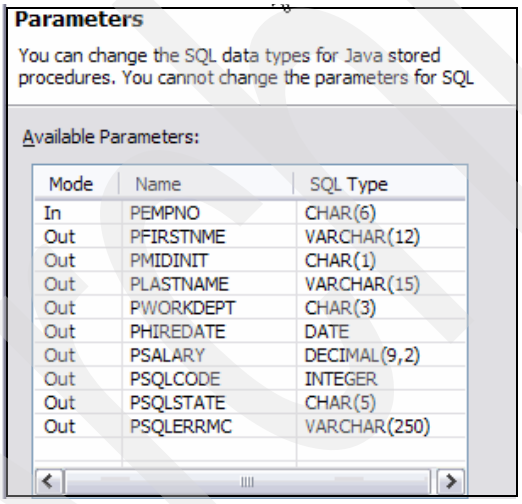

Figure 27-42 Import Wizard, Parameters page

- Stored Procedure Name
  - The stored procedure name, EMPDTLSS, shows on this page.
  - Click **Next.**
- Options
  - The Collection ID is set to `DEVL7083`.
  - Leave unchecked the radio buttons to Replace duplicate routines in the Project, Deploy and Enable Debugging.
  - Click Advanced. This launches the z/OS Options page.
  - Type `DB9AWLMR` for the WLM environment name and click **OK**.
  - See DS_IWOptions.

- Summary
  - The summary page summarizes the above settings. Click **Show SQL** to view the DDL.
  - Click **Finish.**

The imported stored procedure is added to the Stored Procedures folder.

27.5.5 Importing a Java stored procedure

We imported the source for EmpDtlsJ into IBM Data Studio from 13.10.1, “Sample Java stored procedure code: EmpDtlsJ using JDBC” on page 211. We also imported the SQLJ stored procedure, using similar steps.

Right-click the Stored Procedures folder and then select **Import** to launch the Import wizard.

The Import wizard includes the following four pages for importing a stored procedure. It parses the imported source to complete the pages when possible:

- Source

  On this page, you have the option to import from an existing project or the file system. We will import from the file system. Click **Browse** at the right of the Name field to locate the source EmpDtlsJ.java that we had previously saved on our workstation.

- Entry Points

  On the IBM Data Studio page, any methods or routines that are included in this stored procedure are listed by the wizard to be used to select the main entry point. The EmpDtlsJ stored procedure has one method only. IBM Data Studio selects that static method.

- Parameters

  On this page, one input parameter and seven output parameters are listed.

- Stored Procedure Name

  The stored procedure name, GetEmpDtls, shows on this page.

- Options

  On the Options page, type `DEVL7083.EMPDTLSJ` for the JAR ID, and `DEVL7083` for the Collection ID. The collection ID we choose needs to include the DSNJDBCx drivers. Leave unchecked the radio buttons to Replace duplicate routines in the Project, Deploy and Enable Debugging. Next, we select **Advanced** on this page, which launches the z/OS Options page. Type `DB9AWLMJ` for the WLM environment name and click **OK**.

- Summary

  The summary page summarizes the above settings. Optionally, we can view the DDL for the procedure definition for the DB2 catalog by clicking **Show SQL**.

  Click **Finish** to generate the artifacts for the imported Java stored procedure. The Java stored procedure DDL is displayed in the Editor view. Note that, when the server is DB2 for z/OS, the stored procedure name, GetEmplDtls, is changed to all uppercase.

27.6 Deploying a stored procedure

27.6.1 The Deploy wizard

As we have seen, we can always deploy the stored procedure under development to the current server from the New Stored Procedure wizard. However, you may want to edit the stored procedure prior to deploying. Here we examine the Deploy wizard.

You can launch the Deploy wizard from the context menu of the following:

- Database Explorer → *Stored Procedures* folder → **Deploy**
- Database Explorer → a specific stored procedure → **Deploy**
- Data Project Explorer → Stored Procedures folder → **Deploy**
- Data Project Explorer → a specific stored procedure → **Deploy**
- Routine Editor → *Source* page, right-click → **Deploy**
- Routine Editor → *Configurations* page → **Deploy** (see Figure 27-43)

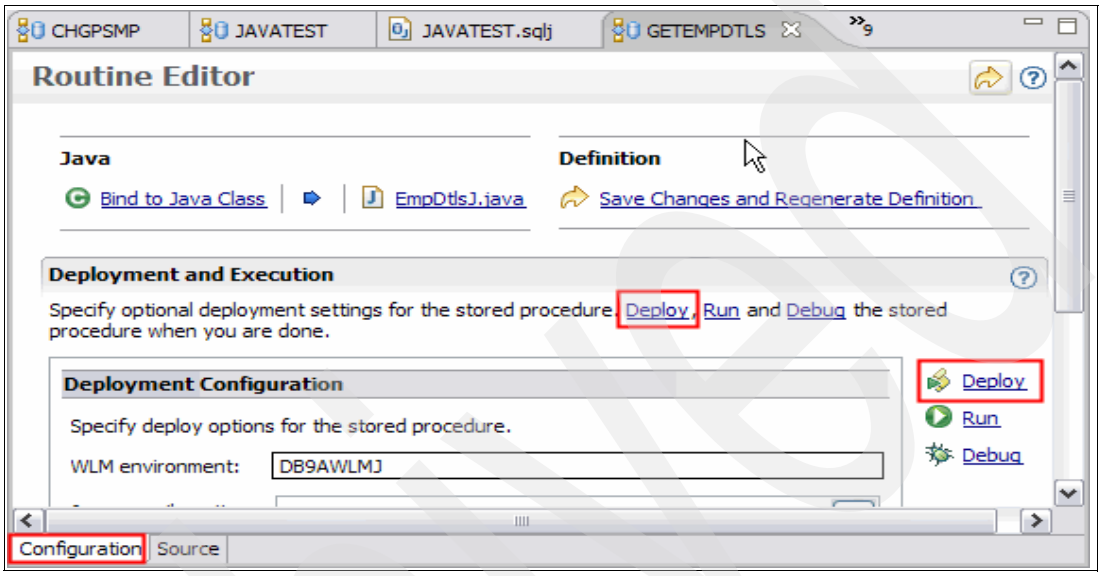

Figure 27-43 Routine Editor, Configuration page

So, let's deploy the stored procedure, GetEmplDtls, that we imported in 27.5.5, "Importing a Java stored procedure" on page 703. Right-click **GetEmpDtls** and then select **Deploy.** The Deploy wizard is launched, as shown in Figure 27-44 on page 705.

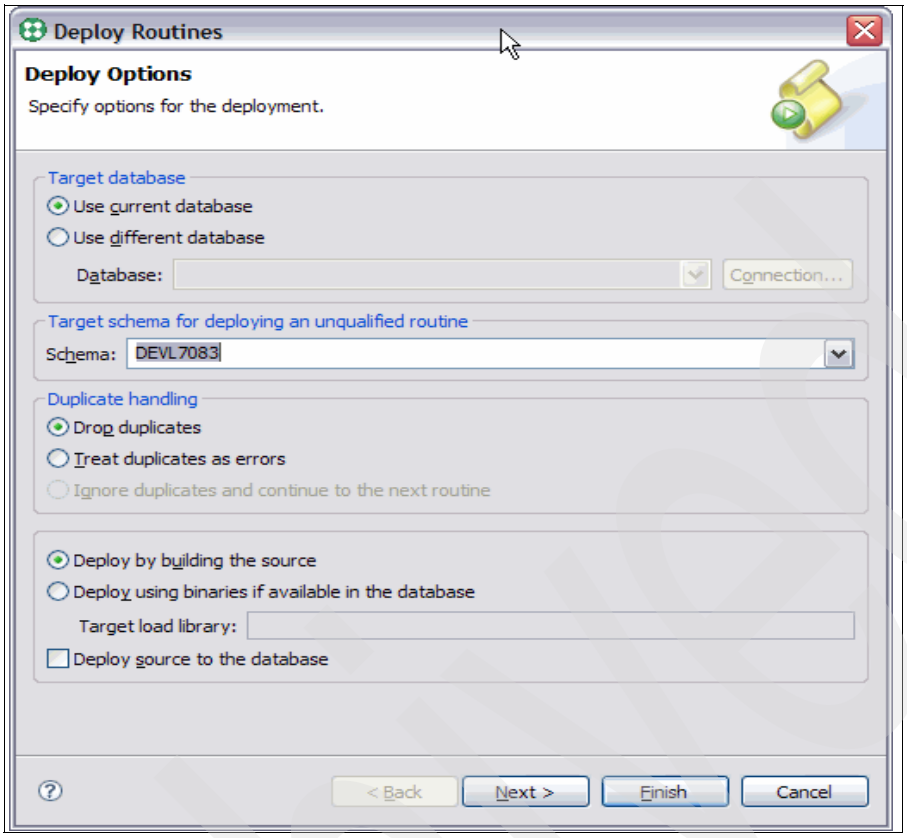

Figure 27-44 Deploy wizard, Deploy Options

Deploy options

- Select **Use current database as the Target connection**.
- Since this stored procedure is unqualified, we can specify the schema for this routine. Select **DEVL7083** from the pull-down list. Default is the login ID.
- Use defaults for all other options. Deploy options are discussed in the 27.6.3, “Duplicate and error handling options” on page 706.

Routine options

- For Java stored procedures, you will have three tabs for this page. In the Routine Options tab, the default WLM environment we set in the Preferences, DB9AWLMJ, is used.
- In the Deploy Options tab, you can do the following:
  - Click the Enable debugging checkbox to enable debug for this deploy. The compile option `-g` is automatically generated. See Figure 27-38 on page 698.
  - Use a different JDK level for the client by clicking **Browse** for the JDK home and pointing the file browser to the directory of your JDK.
  - Specify the JRE level at the server[5]. See Figure 27-45 on page 706 for an illustration of how to specify the JDK and JRE levels.
  - Add additional bind options in the space to the right of the PACKAGE field. Click the ellipsis to display a text box for typing in your bind options.
  - Display all messages generated during the deploy process by clicking **Verbose build**.
  - In the Java Path tab, you can add jars from other projects to resolve references in your stored procedure.

[5] You need to supply the exact version number of the server JRE.

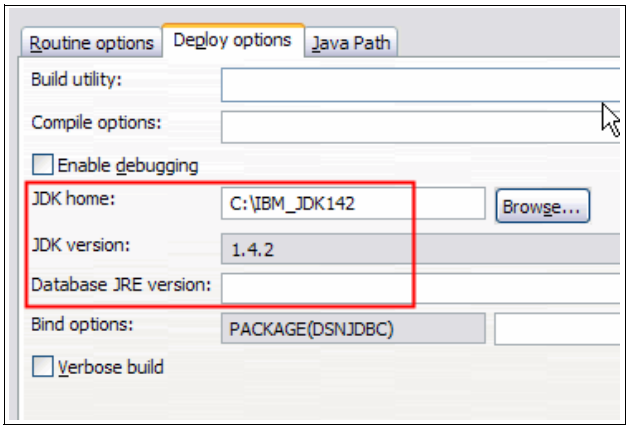

Figure 27-45 Deploy Wizard, changing the JDK and JRE

Summary

This page summarizes the options you specified in the wizard.

Click **Finish** to close the Deploy wizard. The Data Output view will be refreshed with the status and progress of the deploy.

27.6.2 Deploying to a different server

IBM Data Studio supports deploying to an “unlike” DB2 servers, that is to a DB2 server that is running in another type of operating system. Table 27-10 lists the source and target server combinations supported.

Table 27-10 Deploy source and target server combinations

Source Servers	Target Servers		
	LUW V8.2, V9	**z/OS V8**	**z/OS V9**
LUW v8.2, v9	SQL and Java	Java	Native SQL and Java
Derby	Java	N/A	N/A
iSeries V5.4	N/A	N/A	N/A
z/OS V8	N/A	External SQL and Java	External SQL and Java
z/OS V9	N/A	N/A	External SQL, Native SQL and Java

27.6.3 Duplicate and error handling options

Specify how you want to handle duplicates and errors in the tooling.

- Duplicate handling
  - Drop duplicates
  - Treat duplicates as errors
  - Ignore duplicates and continue to the next routine[6]
- Error handling
  - Stop and roll back on errors

– Stop on errors
– Ignore errors and continue to the next routine[7]

27.6.4 Deploying nested or dependent stored procedures

You can call stored procedures from within your stored procedures. If the nested stored procedure is in the same project, you can opt to all or some of the nested stored procedures starting with the innermost nested stored procedure to the outermost one. Figure 27-46 shows the calling hierarchy.

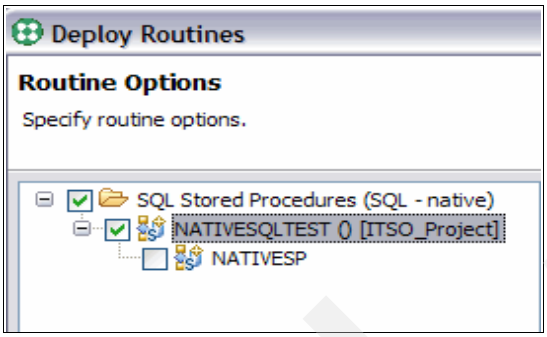

Figure 27-46 Deploying nested stored procedures

27.6.5 Setting the JDK level for Java stored procedures

IBM Data Studio V1.1 ships with the Java Development Kit Version 1.5. DB2 for z/OS V8 typically supports JDK 1.3 and JDK 1.4. Apply PTF for DB2 V8, APAR PK09213 to allow a client Java SP built with JDK 1.5 to execute with a 1.5 JVM in DB2 for z/OS V8. There are setup requirements when applying APAR PK09213. Read the PTF.

In our case study, we compiled our Java stored procedure with JDK 1.5. If we deployed this stored procedure to a DB2 for z/OS V8 server that supports a lower level version of the JDK (for example, JDK 1.4.2), then executing this Java stored procedure compiled will fail with the following error message:

```
17.38.36 STC00108  DSNX961I  DSNX9WLJ ATTEMPT TO PERFORM OPERATION
     -  FindClass
     - FAILED FOR ROUTINE . .  SSN= V91A     PROC= V91AWMJU ASID= 003A
     - CLASS=  METHOD=  ERROR INFO= java.lang.NoClassDefFoundError:
     - com/ibm/db2/jcc/DB2Driver
```

You can change the JDK level by changing the location of the JDK Home setting in the client. You can set this for all projects, for a project or for a specific stored procedure.

- Workspace scope: Go to **Window** → **Preferences** → **Data** → **Stored Procedures & User-Defined Functions** → **Deploy options**.
- Project scope: Right-click the project, then select **Properties** → **Routine Development.**
- Stored procedure scope: In the **Deploy Options** → **Advanced** page of the New Stored Procedure wizard, or in the **Routine Options** → **Deploy Options** tab of the Deploy wizard.

[6] When launched from the Stored Procedures folder, the Deploy wizard allows you to select multiple stored procedures from the folder. When a stored procedure of the same name exists on the server, this option instructs Data Studio to ignore deployment of this stored procedure. Data Studio continues to process and deploy the other stored procedures in the list. The default is to terminate when a duplicate is found.
[7] When deploying a stored procedure from a list, if the stored procedure has errors (e.g. missing objects), then this option instructs Data Studio to ignore these errors, and continue deploying the other stored procedures in the list.

Figure 27-47 shows how to set the JDK home for a specific stored procedure. Note that IBM Data Studio cannot determine the JVM level of the DB2 for z/OS server and defaults the Database JRE version to 1.4. You can override this value if you know the JRE version on the server.

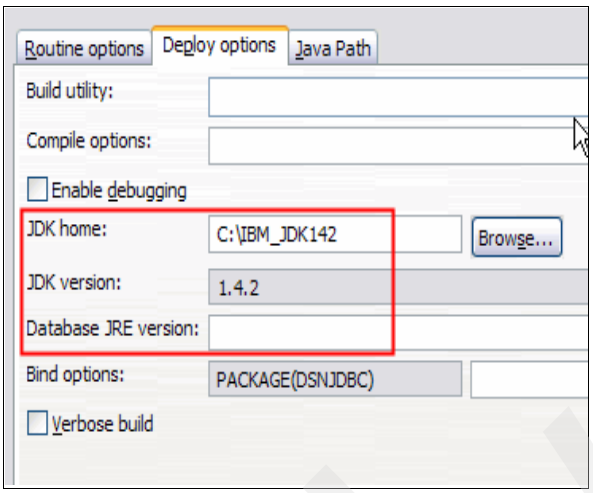

Figure 27-47 Changing the JDK level

Setting the bind options in native SQL stored procedures

Native SQL stored procedures process certain bind options as procedure options. The New Stored Procedure wizard allows you to specify non-default bind options in the Z/OS Options dialog, shown in Figure 27-48. Click the ellipsis to type the options in a larger text area. These options appear in the CREATE PROCEDURE DDL after the VERSION keyword.

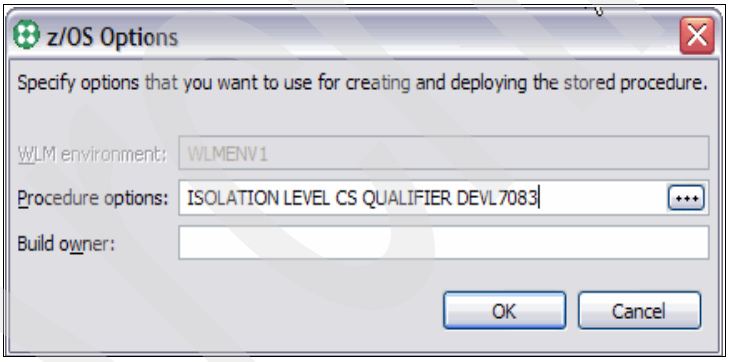

Figure 27-48 Native SQL z/OS Options

Enabling debug

Refer to 28.2, “The Unified Debugger” on page 738 on how to enable debug using the Unified Debugger.

Changing deploy options at deploy time

IBM Data Studio allows you to do a one-time change of the deploy options at deploy time. You may want to do this when you are deploying the stored procedure to a different database or against a different schema.

27.6.6 Executing a stored procedure

After you've deployed your stored procedure, you can execute it from the following areas:

- Database Explorer → a specific stored procedure
- Data Project Explorer → a specific stored procedure
- Routine Editor → Configurations page (see Figure 27-43 on page 704)

If there are any input parameters in your stored procedure, the Specify Parameter Values dialog is launched, as shown in Figure 27-49.

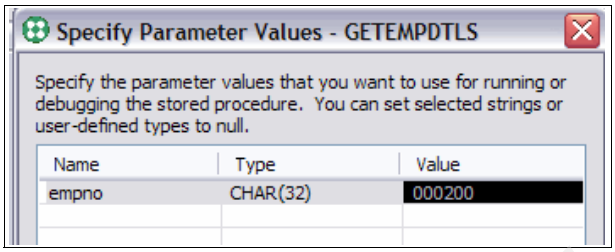

Figure 27-49 Specify parameter values at SP execution

Note that the tooling can recognize that a string was entered even though single quotes were not entered. However, double quotes are preserved in the input value. To enter FOR BIT DATA, type an `X` in front of a quoted value. For example: `X'F1F2F3'`.

Run Settings dialog

You can opt to execute SQL statements before and after calling a stored procedure as when a stored procedure is manipulating a table. You can do this using the Run Settings dialog, as shown in Figure 27-50.

So, for example, to execute the stored procedure we imported and deployed in 27.5.5, "Importing a Java stored procedure" on page 703, from the Data Project Explorer, right-click **GetEmpDtls** and then select **Run Settings**.

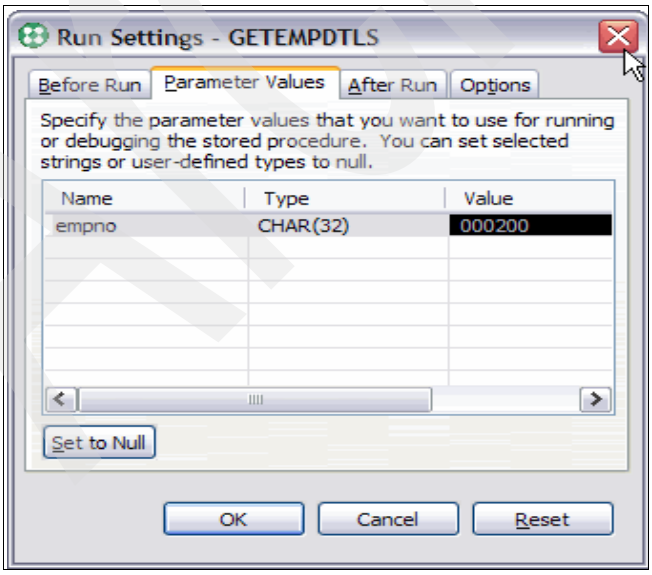

Figure 27-50 Run Settings dialog

Processing information in the Data Output view

In 27.3.3, “Output view” on page 673, we discussed the two areas in the Data Output view. Here we discuss the actions you can take on these areas, as follows:

- Status History table

  Select a cell in the table; right-click the cell and then select **Delete / Delete All** entries.

- In the Messages tab

  Right-click → **Select All**, then right-click → **Copy** to copy the messages to the clipboard.

- In the Messages, Parameters, and Results tabs, right-click the tab, then click **Save As**. This saves the output to a file in your file system.

  If you have multiple result sets, only the result set visible is saved. Click the > to the right of the panel to see additional result sets.

  If your result set contains more than one row, all rows are saved. No formatting is done when saving the column values.

- In the Messages, Parameters, and Results tabs, right-click the tab, then click **Print.** This will allow you to select a printer and print the output.

27.7 Advanced IBM Data Studio topics

This section includes information on the following advanced topics:

- Modifying an existing stored procedure
- Using code fragments
- Generating multiple results
- Drag and Drop or Copy and Paste
- Behavior when setting the Current Schema project property
- Package owner and Build owner
- Export and Ant deploy of stored procedures
- Deploying SQL or Java stored procedures without recompiling
- Creating package variations
- Multiple Jar support for Java stored procedures
- Migrating DC projects to IBM Data Studio

27.7.1 Modifying an existing stored procedure

When you complete the New Stored Procedure wizard, the Routine Editor is populated with a two-tabbed view of the stored procedure. The first tab, Source*,* shows the CREATE PROCEDURE DDL. The second tab, Configuration*,* contains the various properties of the stored procedure.

For Java stored procedures, the Java source can be viewed from the Configuration tab. So to view the Java source for our imported stored procedure, in Figure 27-43 on page 704, click EmpDtls.java. The source will display in another tab in the Editor view. To view the DDL again, click the **GETEMPLDTLS** tab in the Editor View and then the **Source** tab of this view.

Edit Java source

The source of a Java stored procedure is displayed in a Java editor, which includes features such as:

- Syntax highlighting and checking
- Content or code assist
- Code formatting
- Import assistance
- Quick fix

When referring to methods or classes that are in other jar files, you need to add these jar files in the Build Path.

Add or edit parameters

To add parameters to an SQL stored procedure, edit the source and add the new parameter in the CREATE PROCEDURE DDL. Save the stored procedure. Click the **Configuration** tab and verify that the new parameter was added to the Parameters section.

To add parameters to a Java stored procedure, edit the source and add the new parameter in the method signature. So, for example in our test stored procedure, JAVATEST, add java.lang.String tableName in the signature, as shown in Example 27-11.

Example 27-11 Adding a parameter to a Java stored procedure

```
public class JAVATEST {
/**
* Member data fragment inserted from SP_JAVA_MBR.FRAGMENT
*/

public static void jAVATEST(java.lang.String tableName, java.lang.String[] SQLSTATE_OUT,
   int[] SQLCODE_OUT, java.lang.String[] SQLMessage_OUT,
   ResultSet[] rs1) throws Exception {
```

As in SQL stored procedures, you will see the new parameters in the Parameters section of the Configuration tab in the Routine Editor, as shown in Figure 27-51.

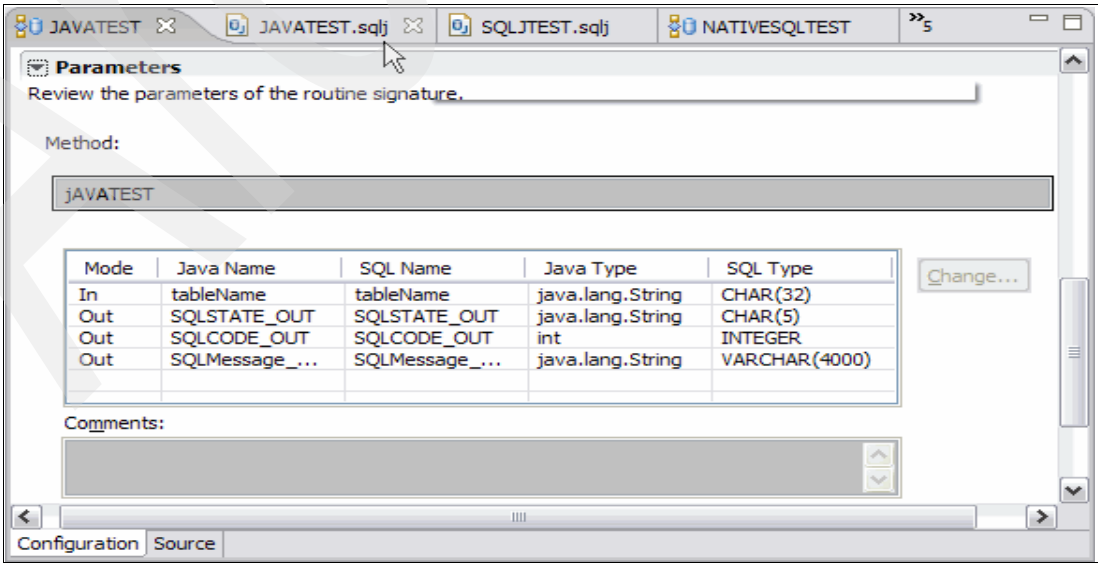

Figure 27-51 Parameter section in Routine Editor's Configuration tab

27.7.2 Using code fragments

IBM Data Studio wizards for creating a new SQL or Java stored procedure include the option to include code fragments on the Definition page of the wizard.

For SQL stored procedures, these fragments relate to the following:

- Header fragment - Enter code that the wizard will insert in the stored procedure header.
- Variable declaration fragment - Enter code that the wizard will insert in the stored procedure variable declaration section.
- Exception handlers fragment - Enter code that the wizard will insert in the stored procedure exception handlers section.
- Pre-return fragment - Enter code that the wizard will insert in the stored procedure pre-return section.

For Java stored procedures, these fragments relate to the following:

- Header fragment - Enter code that the wizard will insert in the stored procedure header.
- Imports fragment - Enter code that the wizard will insert in the stored procedure import section.
- Data fragment - Enter code that the wizard will insert in the stored procedure data section.
- Method fragment - Enter code that the wizard will insert in the stored procedure method section.

Note: In the file browser launched for selecting the code fragments, you can select more than one file. Use Ctrl + Click to multi-select files.

27.7.3 Generating multiple results

You can create multiple statements in the SQL Statements page of the New Stored Procedure wizard by clicking **Add** in the Statements section, as shown in Figure 27-52 on page 713. If each of the statements is a query that returns a result set, then cursors are generated for each of the queries.

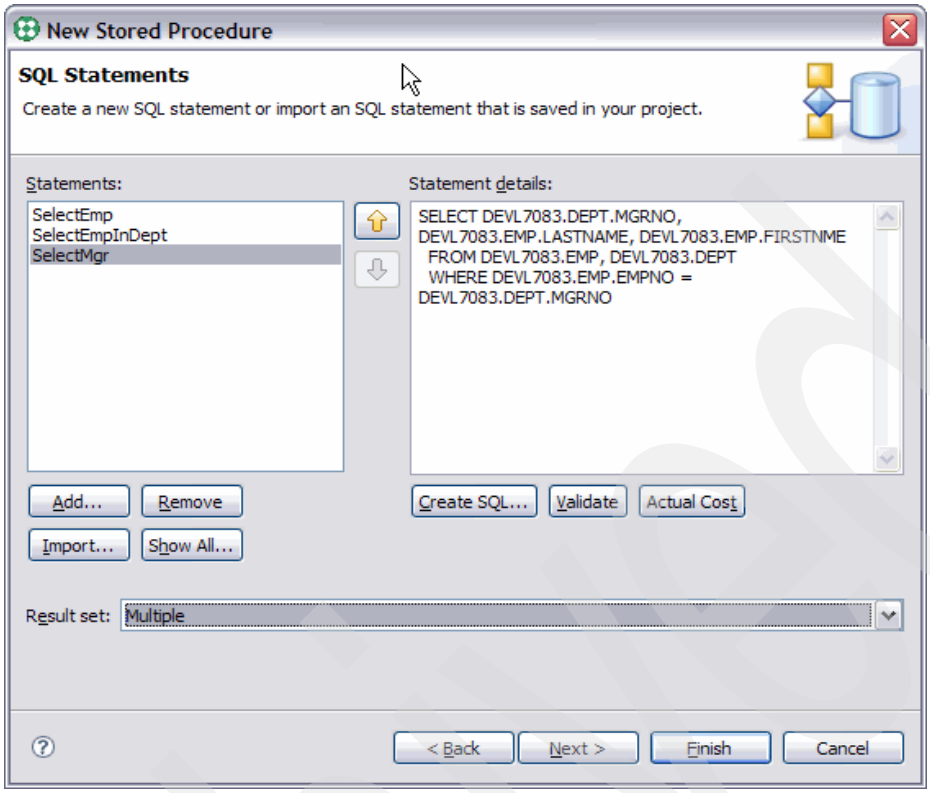

Figure 27-52 Generate multiple SQL statements

If you set the Result set to One instead of Multiple, a CASE statement is generated in the code, as shown in Example 27-12.

Example 27-12 Java stored procedure with multiple SQL statements and one result set.

```
/**
 * JDBC Stored Procedure JDBC_MRS
 * @param whichQuery
 */
/**
 * Header fragment inserted from SP_JAVA_HDR.FRAGMENT
 */
package marichu;

import java.sql.*; // JDBC classes

/**
 * Imports fragment inserted from SP_JAVA_IMPORT.FRAGMENT
 */

public class JDBC_MRS {
/**
* Member data fragment inserted from SP_JAVA_MBR.FRAGMENT
*/

public static void jDBC_MRS(int whichQuery, ResultSet[] rs1)
   throws SQLException, Exception {
   // Get connection to the database
   Connection con = DriverManager.getConnection("jdbc:default:connection");
   PreparedStatement stmt = null;
```

```
    boolean bFlag;
    String sql;

    switch (whichQuery) {
    case 0:
       sql = "SELECT WORKDEPT, EMPNO" + "   FROM DEVL7083.EMP"
          + "   GROUP BY WORKDEPT, EMPNO" + "   ORDER BY WORKDEPT";
       stmt = con.prepareStatement(sql);
       break;
    case 1:
       sql = "SELECT D.DEPTNO, D.DEPTNAME, E.LASTNAME, E.FIRSTNME, E.EMPNO"
          + "   FROM DEVL7083.EMP AS E, DEVL7083.DEPT AS D"
          + "   WHERE E.WORKDEPT = D.DEPTNO" + "   ORDER BY D.DEPTNO";
       stmt = con.prepareStatement(sql);
       break;
    case 2:
          sql = "SELECT DEVL7083.DEPT.MGRNO, DEVL7083.EMP.LASTNAME, DEVL7083.EMP.FIRSTNME"
          + "   FROM DEVL7083.EMP, DEVL7083.DEPT"
          + "   WHERE DEVL7083.EMP.EMPNO = DEVL7083.DEPT.MGRNO";
       stmt = con.prepareStatement(sql);
       break;
    default:
       sql = "SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES";
       stmt = con.prepareStatement(sql);
    }
    bFlag = stmt.execute();
    rs1[0] = stmt.getResultSet();
}

/**
* Methods fragment inserted from SP_JAVA_MTHD.FRAGMENT
*/
}
```

If, instead, we specify Multiple in the Result set option, multiple cursors are left open, as seen in Example 27-13.

Example 27-13 SQL stored procedure with multiple SQL statements and multiple result sets

```
CREATE PROCEDURE DEVL7083.NSQL_MRS (  )
   VERSION VERSION1
   ISOLATION LEVEL CS
   RESULT SETS 3
   LANGUAGE SQL
-------------------------------------------------------------------------
-- SQL Stored Procedure
-------------------------------------------------------------------------
P1: BEGIN
   -- Declare cursors
   DECLARE cursor1 CURSOR WITH RETURN FOR
   SELECT WORKDEPT, EMPNO
      FROM DEVL7083.EMP
      GROUP BY WORKDEPT, EMPNO
      ORDER BY WORKDEPT;
   DECLARE cursor2 CURSOR WITH RETURN FOR
      SELECT D.DEPTNO, D.DEPTNAME, E.LASTNAME, E.FIRSTNME, E.EMPNO
      FROM DEVL7083.EMP AS E, DEVL7083.DEPT AS D
      WHERE E.WORKDEPT = D.DEPTNO
      ORDER BY D.DEPTNO;
   DECLARE cursor3 CURSOR WITH RETURN FOR
```

```
      SELECT DEVL7083.DEPT.MGRNO, DEVL7083.EMP.LASTNAME, DEVL7083.EMP.FIRSTNME
      FROM DEVL7083.EMP, DEVL7083.DEPT
      WHERE DEVL7083.EMP.EMPNO = DEVL7083.DEPT.MGRNO;

   -- Cursor left open for client application
   OPEN cursor1;
   -- Cursor left open for client application
   OPEN cursor2;
   -- Cursor left open for client application
   OPEN cursor3;
END P1
```

27.7.4 Drag and Drop or Copy and Paste

In IBM Data Studio, you can either Copy or Paste or use Drag and Drop, which performs similar actions. You can copy an SQL or Java stored procedure from one server, paste it to a project, modify as needed, and then build on another server. This can be between like platforms and servers or different platforms and servers, where the syntax being used exists on both the source and target.

To copy from the Database Explorer to a project

- If the project already exists:
  - Select the stored procedure in the Database Explorer.
  - Drag and drop to the Stored Procedures folder of the project.
- If the project does not exist:
  - Right-click the stored procedure, then select **Open** → **With Routine Editor**.
  - Specify a project from the pull-down list in Specify Project as shown in Figure 27-53 on page 716

    or
  - Click **New** to create a new project. See 27.4.3, “Creating a Data Development Project” on page 684 for details on how to create a new data development project.

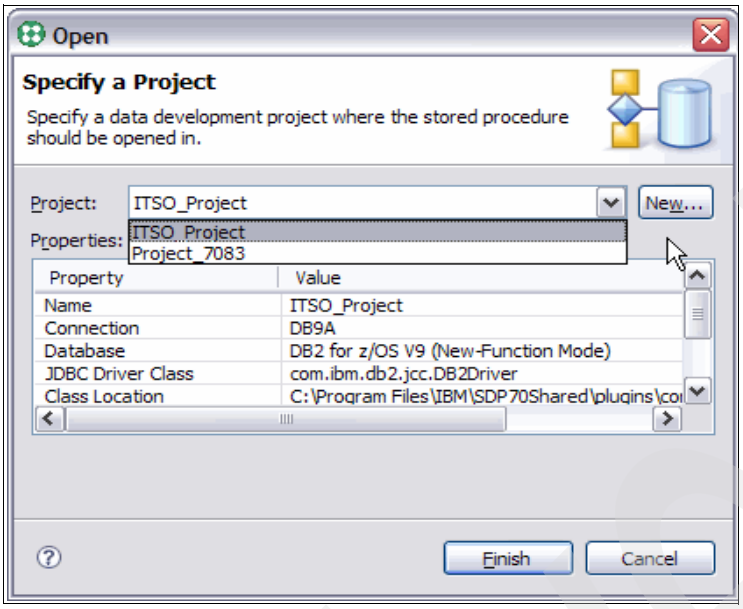

Figure 27-53 Specify a new or existing project

To copy and paste between projects

To copy and paste between projects requires having both projects open. You have two ways to do the copy and paste:

- Select the stored procedure in one project, then drag and drop it to the Stored Procedures folder of the second project.
- Right-click the stored procedure, then click **Copy**. See Figure 27-54. Then right-click the **Stored Procedures** folder of the second project and click **Paste**.

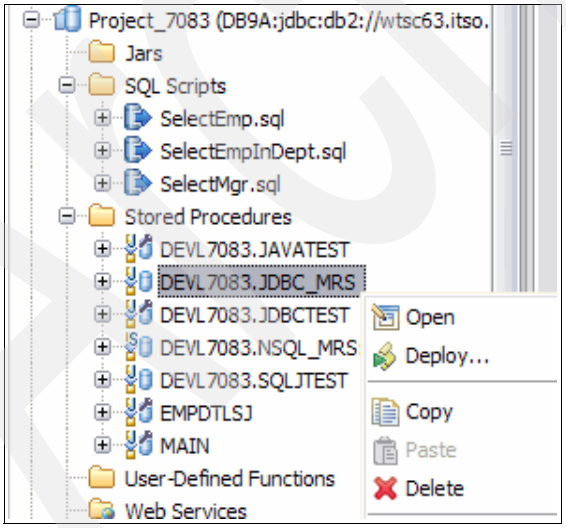

Figure 27-54 Copy a Java or SQL stored procedure to another project

After the copy, you need to modify the stored procedure and deploy it against the new server. Note that syntax differences may exist between source and target servers. Table 27-10 on page 706 shows you the valid combinations of source and target servers that IBM Data Studio supports.

27.7.5 Behavior when setting the Current Schema project property

The CURRENT SCHEMA value can be predefined in the project's Properties. This setting will be used as the default target schema when a stored procedure is deployed without a fully qualified name.

In the Data Project Explorer, right-click the stored procedure → **Properties** → **Development** → **Specify current schema in SQL Format** to set the CURRENT SCHEMA.

Prior to DB2 for z/OS V8 NFM, there was no CURRENT SCHEMA special register. The schema of an unqualified routine was determined by the value in the CURRENT SQLID special register if the routine is created via a dynamic statement. Thus, if a project is connected to a DB2 for z/OS V8 CM or earlier, the CURRENT SCHEMA in the project's Properties must be set to a valid authorization ID. In addition, the connection login ID must be able to issue the SET CURRENT SQLID to this CURRENT SCHEMA value. If the connection ID has SYSADM authority, there is no restriction on the CURRENT SCHEMA value.

DB2 for z/OS V8 NFM introduced a new special register, CURRENT SCHEMA. This special register is dedicated to resolving the schema when referencing an unqualified data object. For projects connected to DB2 for z/OS V8 NFM or later, the CURRENT SCHEMA setting in the project Properties has no restriction and can be any legitimate schema the user has CREATEIN privilege on.

Table 27-11 summarizes the behavior of setting the current schema in the project's Properties when:

- The connection login ID is PAOLOR5.
- The CURRENT SQLID is also set to PAOLOR5.
- DEVL7083 is a valid authorization ID.
- PAOLOR5 has the authority to SET CURRENT SQLID to DEVL7083.

See 7.5.7, "Resolution of unqualified stored procedure names at create time" on page 79 for more information about using the CURRENT SCHEMA on DB2 for z/OS outside IBM Data Studio.

Table 27-11 Current schema behavior

CURRENT SCHEMA[a]	Name specified in New SP Wizard	SYSROUTINES. SCHEMA column value
blank	PROC1	PAOLOR5
blank	DEVL7083.PROC1	DEVL7083
DEVL7083	PROC1	DEVL7083 (V8 NFM, V9[b]) PAOLOR5(V8 CM)
DEVL7083	PAOLOR1.PROC1	PAOLOR1

a. This is the value set in the project Properties.
b. This is the behavior for external SQL procedures created in V8 NFM if APAR PK49647 is applied, which updates the build utility DSNTPSMP to level 1.21.

27.7.6 Package owner and Build owner

In IBM Data studio, the Package owner and Build owner can also be predefined in your project's Properties. In the Data Project Explorer, right-click the stored procedure and then select **Properties** → **Routine Development** → **Package owner** and **Build owner** to set these values.

The Package Owner field will be used to predefine the Package owner for SQL and Java (SQLJ) stored procedures during stored procedure deployment. The Package owner can be overridden during stored procedure deployment (see 27.6.1, “The Deploy wizard” on page 703).

When the Build Owner field is set in the project Properties, the routine deployment will use this Build owner's authority instead of the connection login ID's. Basically, the CURRENT SQLID will be set to the value of Build Owner before the deployment starts. Therefore, the connection login ID must be able to issue SET CURRENT SQLID to this Build owner. Binary deployment is a special case; it cannot use the Build owner's authorization. In this scenario, the connection login ID itself must have proper privilege to create the stored procedure in the remote target server.

27.7.7 Export and ant deploy of stored procedures

IBM Data Studio allows you to export one or more stored procedures to the file system and deploy these stored procedures outside the tooling to a target server. You can optionally zip the files exported, then port the zip file to another workstation where you can issue the deploy.

You can export native SQL, external SQL, and Java stored procedures. Note, however, that to deploy native SQL stored procedures to another DB2 for z/OS V9 server outside of Data Studio, requires either a DB2 Connect or a DB2 on LUW database system running on the client issuing the deploy.

Exporting stored procedures

- In the Data Project Explorer, right-click the **Stored Procedures** folder and then click **Export**.
- The Export Wizard is launched. The first page is the Selection page.
- Select `DEVL7083.SQLJTEST`, `DEVL7083.NSQL_MRS`, `DEVL7083.JDBCTEST`, `DEVL7083.EMPDTLSS`. See Figure 27-55.

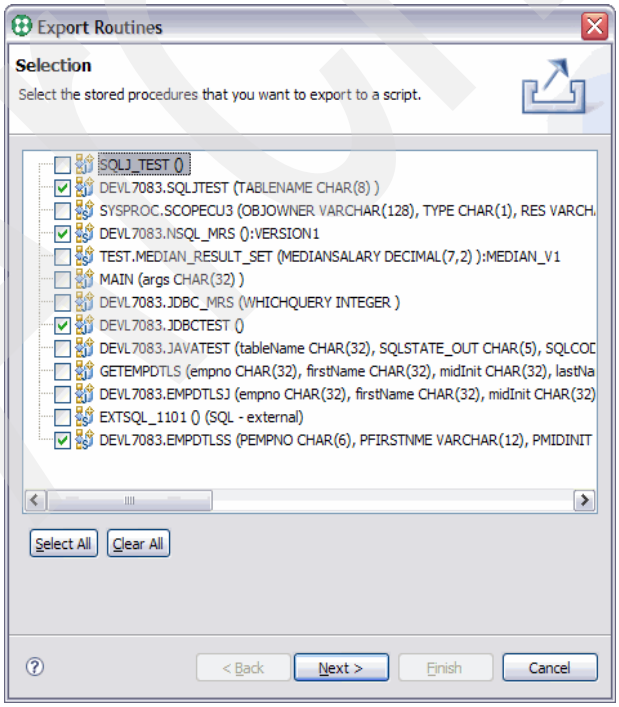

Figure 27-55 Export Wizard Selection page

- Click **Next.** In the Target and Location page, type `ITSO_Project` for the File name, and `C:\Export` for the Directory.
- Click the **Include DROP statements** checkbox. See Figure 27-56 for the completed page.
- Click **Finish.**

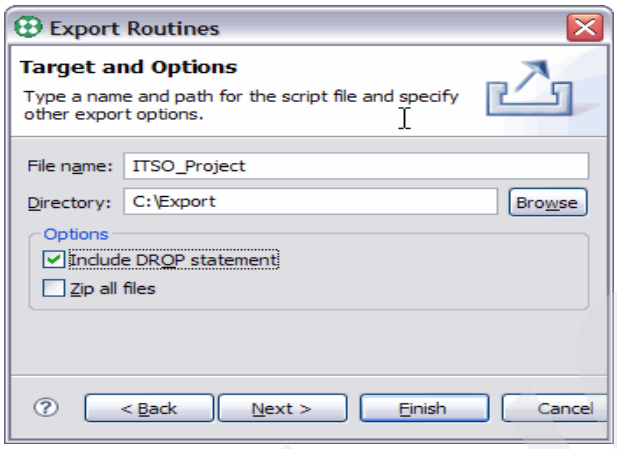

Figure 27-56 Export Wizard Target File name and location

- Verify that the Export is successful for each stored procedure in the Output view. So in our example, we exported four stored procedures. The Output view in Figure 27-57 shows four entries, all of which are successful.

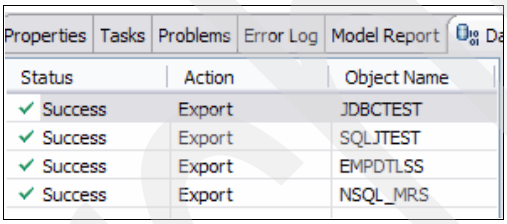

Figure 27-57 Export wizard, Output View Status table

Deploying the exported stored procedures

Data Studio generates several files to facilitate deploying the exported stored procedures and places these files in the directory specified during Export. In our example, we exported the following stored procedures:

Native SQL stored procedure, NSQL_MRS

External SQL stored procedure, EMPDTLSS

Java stored procedure using JDBC, JDBCTEST

Java stored procedure using SQLJ, SQLJTEST

Before deploying the stored procedures, ensure that your client's JAVA_HOME environment variable is pointing to a JDK level that is compatible with the JDK level at the server. The generated ant.bat file uses the JAVA_HOME setting when launching ant.

Now, open a DB2 command window and go to C:\Export directory.

To deploy the native SQL stored procedures, NSQL_MRS

- Edit the ITSO_Project.sql file.

- Uncomment the CONNECT TO statement.
- Update the userid and password to the login userid and password you use to connect to the target server.
- Modify the SET CURRENT SCHEMA statement to set a schema for unqualified database objects.
- Uncomment the CONNECT RESET statement.
- Uncomment the @ sign.
- Save the ITSO_Project.sql file.
- Open a DB2 command window.
- Run the following command: db2 -td@ -vf `ITSO_Project.sql`.

To deploy an external SQL stored procedure, EMPDTLSS

- Edit the ITSO_Project_sql.properties file.
- Update the properties where required. The target connection properties are defined in the following properties:

  db.userid=paolor5
  db.password=
  db.name=DB9A
  db.hostname=wtsc63.itso.ibm.com
  db.port=12347
- Save the ITSO_Project_sql.properties file.
- Run ant.bat to execute the script. Type `ant -buildfile ITSO_Project_sql.xml`

Example 27-14 shows the output of the above ant command.

Example 27-14 Output of ant deploy of stored procedure EMPDTLSS

```
Buildfile: ITSO_Project_sql.xml

init:

builddeploySps:
 [createsp] Debug options:
 [createsp]     file:/C:/Export/.options loaded
 [createsp] Could not connect to the target database.
 [createsp] [ibm][db2][jcc][t4][2013][11249] Connection authorization failure occurred.
Reason: User ID or Password invalid.

BUILD SUCCESSFUL
Total time: 3 seconds
```

To deploy a Java stored procedure, JDBCTEST and SQLJTEST

- Edit the ITSO_Project_java.properties file.
- Update the properties where required. The target connection properties are defined in the following properties:

  db.userid=paolor5
  db.password=
  db.name=DB9A
  db.hostname=wtsc63.itso.ibm.com
  db.port=12347
- Save the ITSO_Project_java.properties file.

- Run ant.bat to execute the script. Type `ant -buildfile ITSO_Project_java.xml`.

Excerpts from the output of the ant deploy of Java stored procedures are shown in Example 27-15.

Example 27-15 Output of ant deploy of SQLJTEST and JDBCTEST

```
Buildfile: ITSO_Project_java.xml

init:

builddeploySps:
 [createsp] Debug options:
 [createsp]     file:/C:/Export/.options loaded
 [createsp] DEVL7083.SQLJTEST - Deploy started.
[createsp] DEVL7083.SQLJTEST - Created temporary working directory
C:\Export\bld1196747605594.
 [createsp] DEVL7083.SQLJTEST - Translating the SQLJ source file
C:\Export\bld1196747605594\marichu\SQLJTEST.sqlj using
 [createsp]     SQLJ translator class: sqlj.tools.Sqlj
 [createsp]     SQLJ translator location: C:\SQLLIB\java\sqlj.zip;C:\Program...
 [createsp] DEVL7083.SQLJTEST - SQLJ translation completed.
 [createsp] C:\IBM_JDK15\bin\javac -classpath ".;C:\SQLLIB\java\sqlj.zip;C:\Program...
 [createsp] DEVL7083.SQLJTEST - Javac completed.
 [createsp] DEVL7083.SQLJTEST - Class file updated.
 [createsp] C:\IBM_JDK15\bin\javaw -cp ".\;C:\Program... C:\Export\bld1196747605594"
com.ibm.db2.jcc.sqlj.Customizer -url jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A -collection
DSNJDBC -qualifier PAOLOR4 -user PAOLOR4 -password xxxxxxxx -bindoptions "QUALIFIER
PAOLOR4" -rootPkgName S928288 marichu\SQLJTEST_SJProfile0.ser

 [createsp] [jcc][sqlj] Begin Customization
 [createsp] [jcc][sqlj] Set qualifier for online checking to SCHEMA: PAOLOR4
 [createsp] [jcc][sqlj] Loading profile: marichu\SQLJTEST_SJProfile0
 [createsp] [jcc][sqlj] Customization complete for profile marichu\SQLJTEST_SJProfile0.ser
 [createsp] [jcc][sqlj] Begin Bind
 [createsp] [jcc][sqlj] Loading profile: marichu\SQLJTEST_SJProfile0
 [createsp] [jcc][sqlj] User bind options: QUALIFIER PAOLOR4
 [createsp] [jcc][sqlj] Driver defaults(user may override): BLOCKING ALL VALIDATE BIND
 [createsp] [jcc][sqlj] Fixed driver options: DATETIME ISO DYNAMICRULES BIND
 [createsp] [jcc][sqlj] Binding package S9282881 at isolation level UR
 [createsp] [jcc][sqlj] Binding package S9282882 at isolation level CS
 [createsp] [jcc][sqlj] Binding package S9282883 at isolation level RS
 [createsp] [jcc][sqlj] Binding package S9282884 at isolation level RR
 [createsp] [jcc][sqlj] Bind complete for marichu\SQLJTEST_SJProfile0

 [createsp] DEVL7083.SQLJTEST - SQLJ profile customization completed.
 [createsp] C:\IBM_JDK15\bin\jar uf spjar.jar  marichu\SPContext.class
marichu\SQLJTEST.class marichu\SQLJTEST_Cursor1.class marichu\SQLJTEST_SJProfile0.ser
marichu\SQLJTEST_SJProfileKeys.class
 [createsp] DEVL7083.SQLJTEST - Jar file created.
 [createsp] DELETE FROM SYSIBM.SYSJAVAOPTS WHERE JARSCHEMA = 'PAOLOR4' AND JAR_ID =
'SQLJTEST'
 [createsp] Call SQLJ.DB2_REPLACE_JAR (<<C:\Export\bld1196747605594\spjar.jar>>,
'PAOLOR4.SQLJTEST')
 [createsp] DEVL7083.SQLJTEST - SQLJ.DB2_REPLACE_JAR using Jar name PAOLOR4.SQLJTEST
completed.
 [createsp] Call ALTER_JAVA_PATH ('PAOLOR4.SQLJTEST', '')
 [createsp] DEVL7083.SQLJTEST - SQLJ.ALTER_JAVA_PATH using Jar name PAOLOR4.SQLJTEST
completed.
 [createsp] DEVL7083.SQLJTEST - Supporting Jars installed successfully.
```

```
 [createsp] Call SQLJ.DB2_UPDATEJARINFO ('PAOLOR4.SQLJTEST', 'marichu.SQLJTEST',
<<C:\Export\bld1196747605594\marichu\SQLJTEST.sqlj>>, 'S928288', 'DSNJDBC',
'ACTION(REPLACE)')
 [createsp] DEVL7083.SQLJTEST - Source saved to the server.
 [createsp] DEVL7083.SQLJTEST - Removed temporary working directory
C:\Export\bld1196747605594.
 [createsp] DSNT540I  DB9AWLMJ WAS REFRESHED BY PAOLOR4  USING AUTHORITY FROM SQL ID
PAOLOR4 : 0
 [createsp] DEVL7083.SQLJTEST - Deploy successful.
 [createsp]
 [createsp] ====================================

 [createsp] DEVL7083.JDBCTEST - Deploy for debug started.
 [createsp] DEVL7083.JDBCTEST - Created temporary working directory
C:\Export\bld1196747618983.
 [createsp] C:\IBM_JDK15\bin\javac -classpath "..... -g -source 1.4 -target 1.4
marichu\JDBCTEST.java
 [createsp] DEVL7083.JDBCTEST - Javac completed.
 [createsp] C:\IBM_JDK15\bin\jar uf spjar.jar  marichu\JDBCTEST.class
 [createsp] DEVL7083.JDBCTEST - Jar file created.
 [createsp] DELETE FROM SYSIBM.SYSJAVAOPTS WHERE JARSCHEMA = 'PAOLOR5' AND JAR_ID =
'JDBCTEST'
 [createsp] Call SQLJ.DB2_REPLACE_JAR (<<C:\Export\bld1196747618983\spjar.jar>>,
'PAOLOR5.JDBCTEST')
 [createsp] DEVL7083.JDBCTEST - SQLJ.DB2_REPLACE_JAR using Jar name PAOLOR5.JDBCTEST
completed.
 [createsp] Call ALTER_JAVA_PATH ('PAOLOR5.JDBCTEST', '')
 [createsp] DEVL7083.JDBCTEST - SQLJ.ALTER_JAVA_PATH using Jar name PAOLOR5.JDBCTEST
completed.
 [createsp] DEVL7083.JDBCTEST - Supporting Jars installed successfully.
 [createsp] Call SQLJ.DB2_UPDATEJARINFO ('PAOLOR5.JDBCTEST', 'marichu.JDBCTEST',
<<C:\Export\bld1196747618983\marichu\JDBCTEST.java>>, '', 'DSNJDBC', 'ACTION(REPLACE)')
 [createsp] DEVL7083.JDBCTEST - Source saved to the server.
 [createsp] DEVL7083.JDBCTEST - Removed temporary working directory
C:\Export\bld1196747618983.
 [createsp] DSNT540I  DB9AWLMJ WAS REFRESHED BY PAOLOR4  USING AUTHORITY FROM SQL ID
PAOLOR4 : 0
 [createsp] DEVL7083.JDBCTEST - Deploy for debug successful.
 [createsp]
 [createsp] ====================================


BUILD SUCCESSFUL
Total time: 24 seconds
```

27.7.8 Deploying SQL or Java stored procedures without recompiling

Some customers want to migrate compiled code and not rebuild the stored procedure on the target server. You can do this manually following the steps in 18.3.1, “Compile just once” on page 378.

The IBM Data Studio’s Deploy wizard can be used to deploy using binaries—SQL or Java stored procedures between source and target servers on the same platform.

External SQL stored procedures

Deploying external SQL stored procedures without rebuilding requires doing the following steps from the tooling:

- Right-click the stored procedure and then click **Deploy.**
- In the Deploy Wizard, change the Target connection to “Use Different Database”. Select your target database from the pull-down list, or create a new connection to this database (see 27.4.2, “Creating a connection” on page 682).
- If the stored procedure is unqualified, select the schema name to be used for this stored procedure.
- Set your duplicate and error handling options.
- Click the checkbox **Deploy using binaries, if available in the database**.
- The Target Load Library will be enabled. Specify a PDS file to receive the compiled SQL module in the target database.
- Click **Next** to change the deploy and routine options, as discussed in 27.6, “Deploying a stored procedure” on page 703.
- Click **Finish** to complete the deploy.

Figure 27-58 on page 723 shows the completed Deploy wizard.

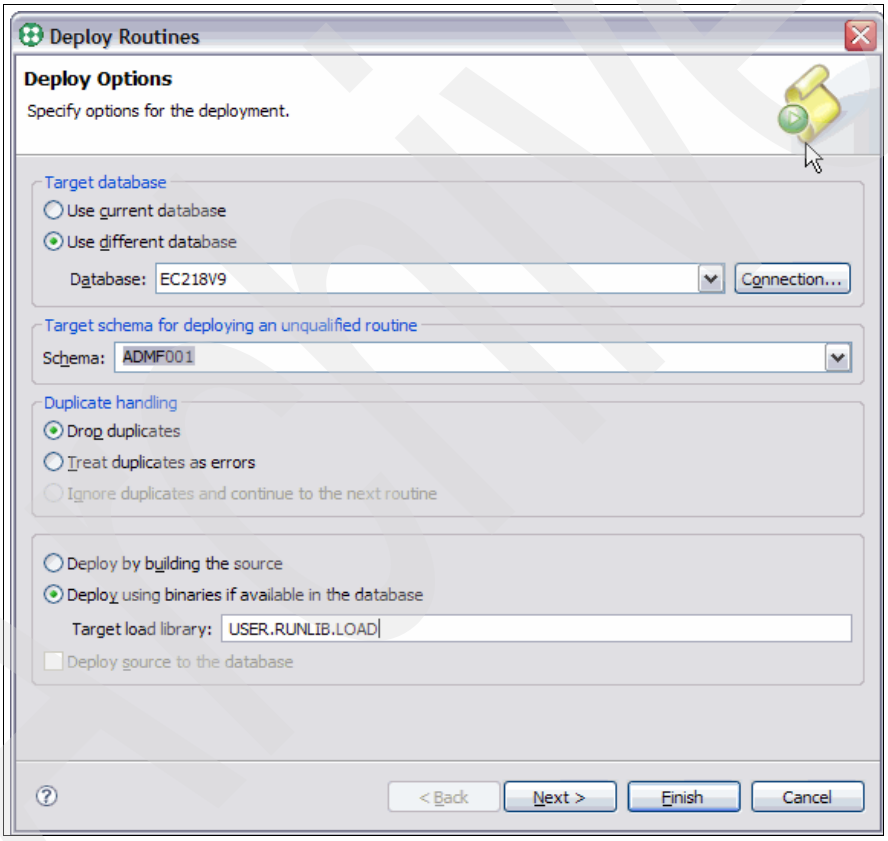

Figure 27-58 Deploy using binaries

In addition to deploying the binaries, you may want to set up the same authorizations in the target server. To do this, do the following:

- In the Database Explorer, right-click the stored procedure you just deployed and then select **Generate DDL**.
- In the Options page, click **Deselect All** → **Next.**
- In the Objects page, select **Privileges** and **Stored Procedures** → **Next**.

- In the Run and Save DDL page (see Figure 27-59 on page 724, do the following:
  - Specify a project Folder. You can save the DDL in a project that is connected to the source, target, or any compatible server.
  - Specify a File Name for the generated script file.
  - Verify that the generated GRANTs are correct in the Preview DDL pane.
  - Select the **Open DDL for editing** checkbox.
  - Click **Next**.
- Click **Finish.** The generated DDL will display in the Editor view.
- Right-click on whitespace in the Editor view and then select **Use Database Connection** to set the target connection.
- Right-click on whitespace, or right-click the script name in the project's outline and then click **Run SQL.**

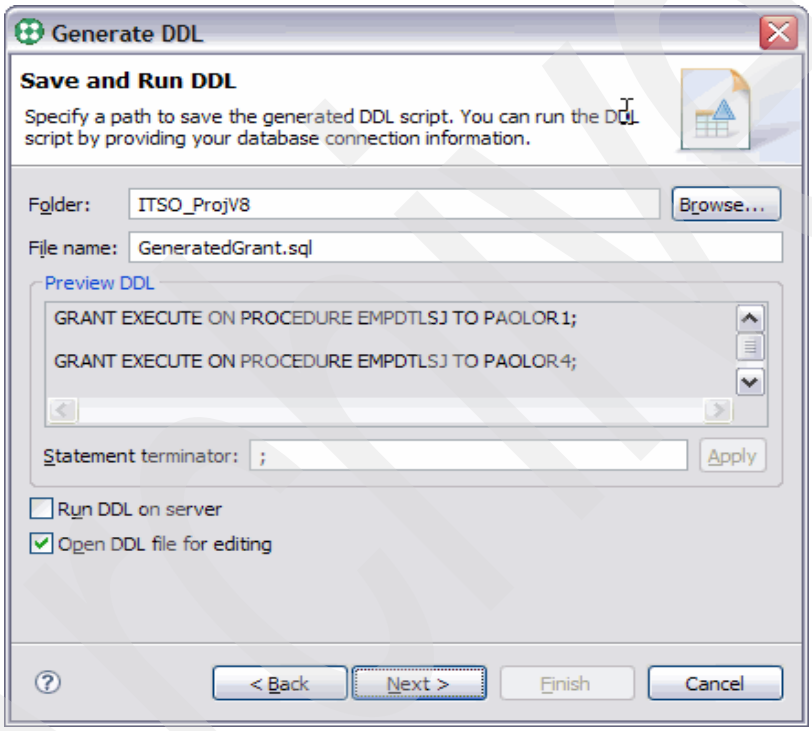

Figure 27-59 Generate privileges

Native stored procedures

To deploy native stored procedures using binaries, do the following:

- Right-click the stored procedure and then select **Deploy**.
- In the Deploy Wizard → Target connection, click **Use Different Database**.
- Select your target database from the pull-down list, or create a new connection to this database (see 27.4.2, "Creating a connection" on page 682).
- If the stored procedure is unqualified, select a schema name to be used for this stored procedure.
- Set your duplicate and error handling options.
- Click **Deploy using binaries, if available in the database**.

- Click **Next** to change the deploy and routine options, as discussed in 27.6, “Deploying a stored procedure” on page 703.
- Click **Finish** to complete the deploy.

Java stored procedures

IIBM Data Studio supports deploying using binaries only if the Java stored procedure was initially built on the client, and the connection used the IBM Universal driver. Java stored procedures built using the utility DSNTJSPP are not eligible to be deployed using binaries.

The steps to deploy Java stored procedures using binaries are the same as those followed for native SQL stored procedures.

27.7.9 Creating package variations

IBM Data Studio supports creating packages with different bind options for SQL and SQLJ Java stored procedures on DB2 for z/OS V8 NFM and DB2 for z/OS V9. The packages are created in different collection IDs. Use the New Package Variation wizard (see Figure 27-60) to create a package variation of an existing stored procedure package.

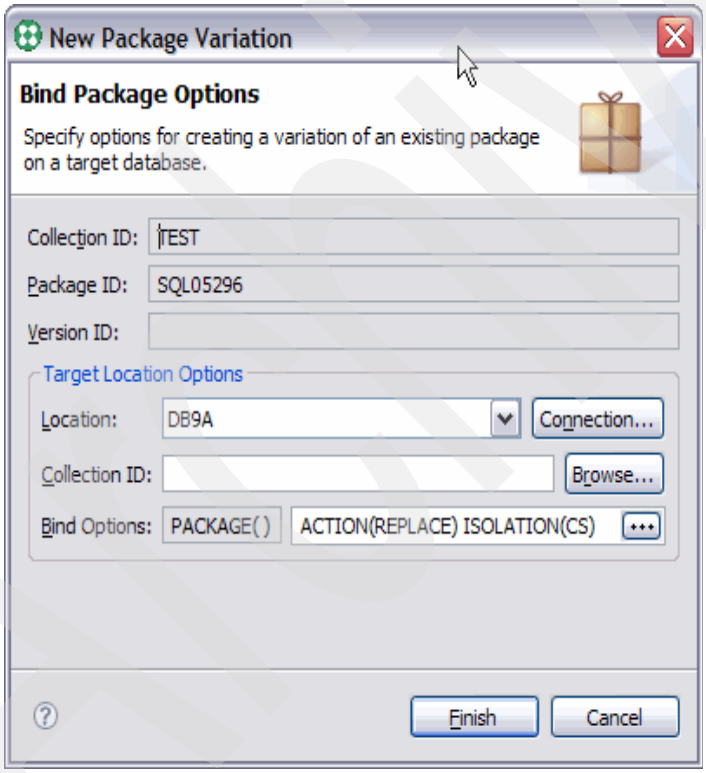

Figure 27-60 New Package Variation wizard

To create a package variation, do the following:

- Initially deploy the SQL or SQLJ stored procedure on a DB2 for z/OS server using the NO COLLID option.
- Bind the collid you want the new package to reside in using com.ibm.db2.jcc.DB2Binder. Otherwise, you will get a -805 at execution time.
- From the Database Explorer, expand the connection to the newly deployed stored procedure.

- Right-click the stored procedure and the select **Packages** → **New**. The New Package Variation wizard is launched.
- Select a Target Connection. The default is the current database connection.
- Select a Collection ID. Click **Browse** to view a list of available collection IDs.
- Specify the Bind options for this package.
- Click **Finish** to deploy the new package to the server. The new package will be added to the Packages folder of the stored procedure.

To execute a stored procedure using a specific package, do the following:

- Right-click the stored procedure and then select **Run Settings** → **Options** tab.
- Type the collection ID associated with the package version you wish to execute in the Collection ID field. Click **Browse** to view a list of the collection IDs of the packages associated with this stored procedure.
- Click **OK**.
- Right-click the stored procedure and then click **Run** to execute the stored procedure.

For more information on package variation, see the article, "Create package variations for z/OS DB2 stored procedures" on developerWorks® at:

`http://www.ibm.com/developerworks/db2/library/techarticle/dm-0608parmeshwar/`

Note: If the user did not specify a Package ID, the tooling generates a Package ID with a value of "SQL" + a randomly generated number. After deploying the stored procedure, you can browse the deployed stored procedure's properties in the Database Explorer using the Property browser. The generated Package ID is in the Options tab of the stored procedure's Properties View. You can also browse the package variation properties in the same manner.

27.7.10 Multiple Jar support for Java stored procedures

This feature, available with DB2 for z/OS V9 only, allows users to reference classes in supporting jar files. A "supporting jar" is a jar that will be referenced by another jar at runtime. Java stored procedures can reference classes that are either in the CLASSPATH of the associated Java stored procedure WLM proc, or in the jar that the stored procedure resides in.

IBM Data Studio associates an installed jar file with a Java path and allows the user to specify this Java path.

In the additional materials for this book, we provide two Java stored procedures, EmpDtlsMJ.java and Getters_staff.jar. We use these to illustrate how to add supporting jars to your DB2 stored procedure.

Import supporting jars to the project

1. In the Data Project Explorer, open project Project_7083. Right-click it and then click **Jars->Import**.
2. In the Import wizard, specify `DEVL7083.GETTERS_STAFF` for the JAR ID. Click **Browse**.
3. A file browser is launched. Point to the location of Getters_staff.jar and click **OK**.
4. Check the **Deploy** checkbox.
5. Type `DEVL7083` for Current Schema.

6. Figure 27-61 on page 727 shows the completed page. Click **Finish.**

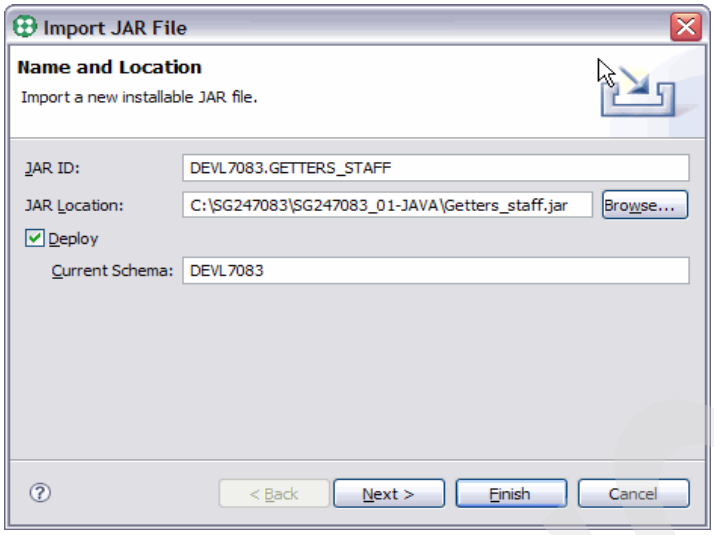

Figure 27-61 Import a jar file

Create a Java stored procedure with a supporting jar

1. Create a Java Stored procedure. EmpDtlsMJ, using steps in 27.5.3, “Creating a Java stored procedure from the wizard” on page 696.
2. In the Name and Language page
   a. Type `DEVL7083.EMPDTLSMJ` for the Name.
   b. Select **Java** from the Language pull-down list.
   c. Type `com.ibm.test` for the Java package.
3. In the Deploy Options page, type `DEVL7038,EMPDTLSMJ` for JAR ID and click **Next.**
4. In the Java Path, click **Add**.
5. In the Add Installable Jar dialog, select **DEVL7087.GETTERS_STAFF.**
6. Click **Browse** next to Class Reference pattern.
7. Select the default class shown in the View GETTERS_STAFF dialog and click **OK.**
8. Click **OK** again to return to the New Stored Procedure wizard. The completed Java Path page is shown in Figure 27-62 on page 728.
9. Click **Finish.**

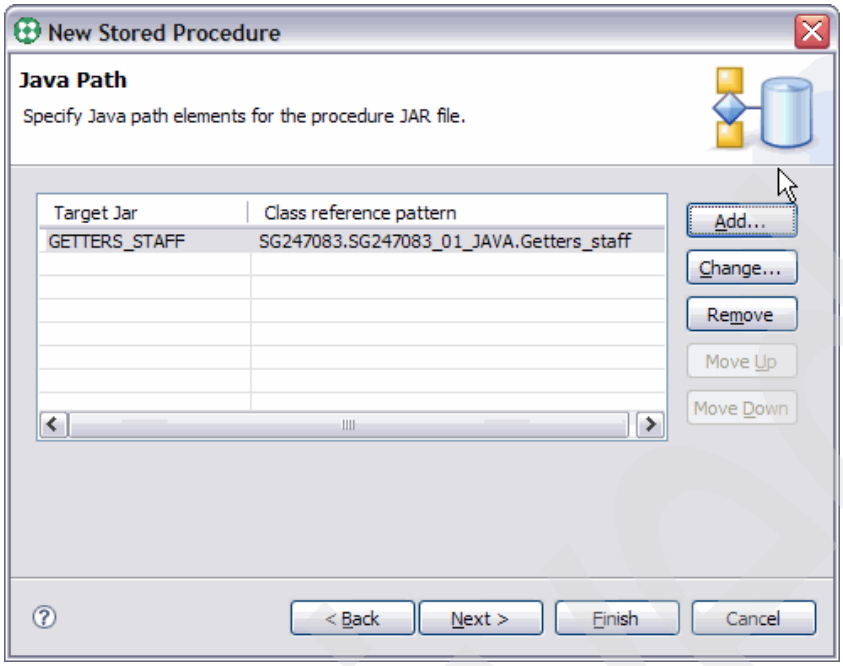

Figure 27-62 Add supporting jar for Java stored procedure

Data Studio creates a dependency on the added jar file and reports this in the Routine Editor's Configuration tab → Files section, as shown in Figure 27-63.

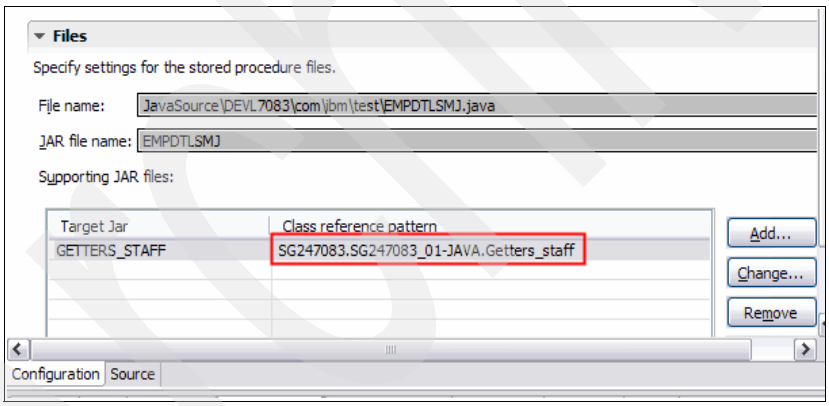

Figure 27-63 Routine Editor → Configuration tab → Files shows supporting jars

Modify Java source to use supporting jar classes

1. In the Routine Editor, click the **Configuration** tab.
2. Click **EmpDtlsMJ.java**.
3. Ctrl-A on the Java source and click **Delete.**
4. Copy the source from the downloaded file, EmpDtlsMJ.java, and paste it to the Routine Editor.
5. Edit the source file by adding the following import statement:

   ```
   import SG247083.SG247083_01_JAVA.*;
   ```

6. Ctrl-S or click **Save** to save your changes.

At this point, you will notice that there are no compilation errors in your Java stored procedure. The calls to methods in the Getters_staff class are resolved.

In the above example, we created the jar by importing it from the file system. This created an entry in the Jars folder of the project. You can also add previously deployed jars by dragging and dropping them from the Database Explorer.

From the Database Explorer, you can also

- View or browse all jars installed on the server.
- View the catalog data and Java path in the Property Browser for each jar.
- View jars that are dependent on a particular jar.
- Drop supporting jars.

From the Data Project Explorer, you can

- Deploy a jar to the server.
- Replace the jar if already installed in the server.
- Edit the supporting jar characteristics, such as the Java path during deploy.
- Deploy a Java stored procedure and all its supporting jars into the same jar file.

27.7.11 Migrating DC projects to IBM Data Studio

You can reuse projects created in Development Center and continue your work in Developer Workbench or IBM Data Studio. These projects can then exploit the new functions in DWB and Data Studio.

This feature is not automatically installed in IBM Data Studio. Select this feature when installing IBM Data Studio as shown in Figure 27-64. If you already have IBM Data Studio installed, use the IBM Installation Manager to Modify Packages, and add this feature.

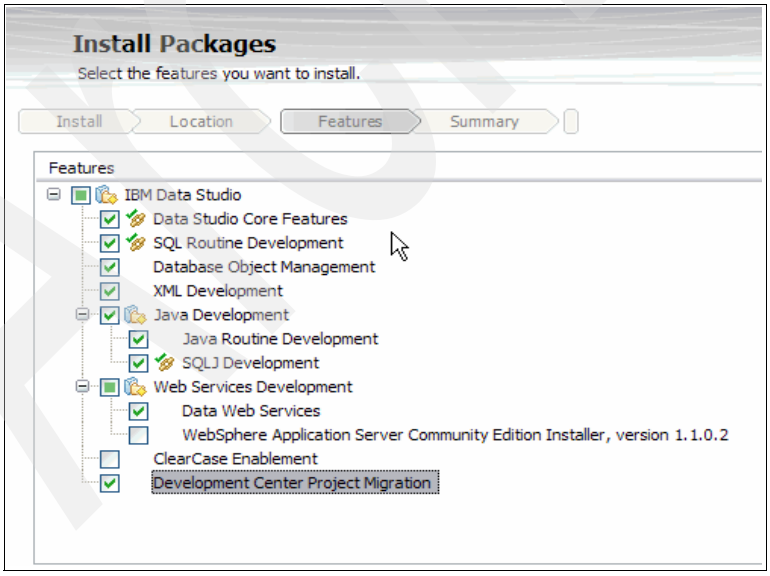

Figure 27-64 Installing the DC Project Migration feature in Installation Manager

To migrate projects using the Migrate DC Project wizard, do the following:

- Click **File** → **New** → **DC Project Migration** → **DB2 Development Center Project** (see Figure 27-65 on page 730). Click **Next.**

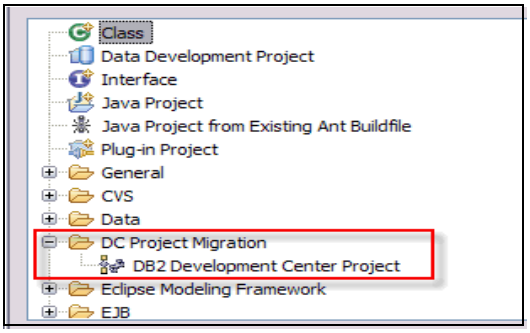

Figure 27-65 Launch the DC Project Migration wizard

- In the next page, click the ellipsis for Select a project to migrate.
- Using the File browser, point to the .dcp file you want to migrate. The input DC project must be a .dcp file. Click **Open.**
- Once the project is loaded, all DC project connections are listed, as shown in Figure 27-66.

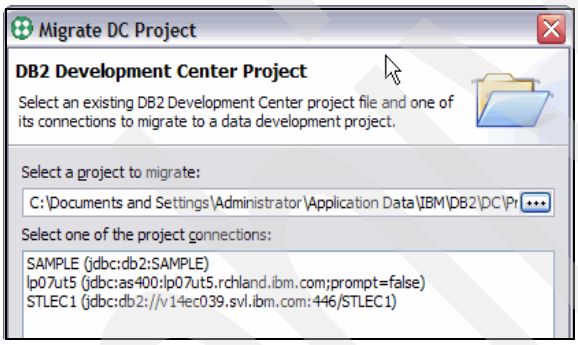

Figure 27-66 DC Project .dcp file and connections.

DC projects allowed multiple connections; Data Studio projects only have one connection.

- In the Select one of the project connections: list box, select a connection. Click **Next.**
- In the Data Development Project page, you can leave the old DC project name or rename the project. You can also set the project's CURRENT SCHEMA. Click **Next.**
- In the Select Connection page, you can select an existing connection or create a new connection (see Figure 27-67 on page 731). Data Studio will attempt to match the selected connection to existing connections in the Database Explorer. Click **Next.**
- In the next page, you can add a default Package Owner and Build Owner for routines in this new project.

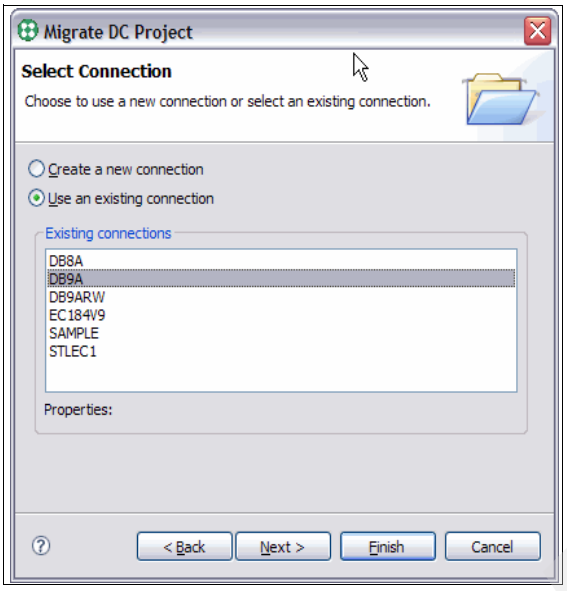

Figure 27-67 Migrate DC projects, select connection

27.7.12 Creating a Web Service from a stored procedure

IBM Data Studio offers a feature to create Web services that expose database operations (SQL SELECT and DML statements, XQuery expressions, or calls to stored procedures) to client applications.

Here we show how to add the EMPDTLSJ stored procedure to a Web service.

- In the Data Explorer → PROJECT_7083, right-click the Web Services folder and then select **New Web Service**, as shown in Figure 27-68.

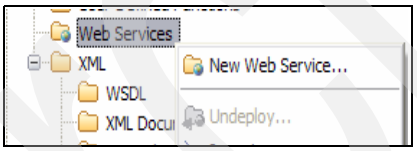

Figure 27-68 Create a new Web service

- The New Web Service dialog is displayed, as in Figure 27-69. Type `WebService7083` for the Name. We will use the default URI for this Web service. Click **Finish.**

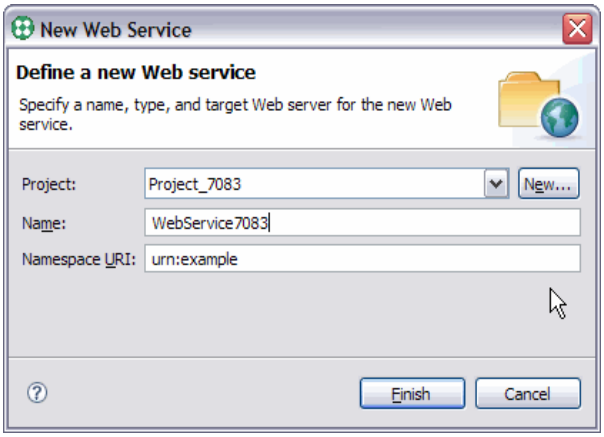

Figure 27-69 Define a new Web Service

- Now let's add a stored procedure to this Web service. Right-click the **EMPDTLS** stored procedure and then select **Add to Web Service** (see Figure 27-70).

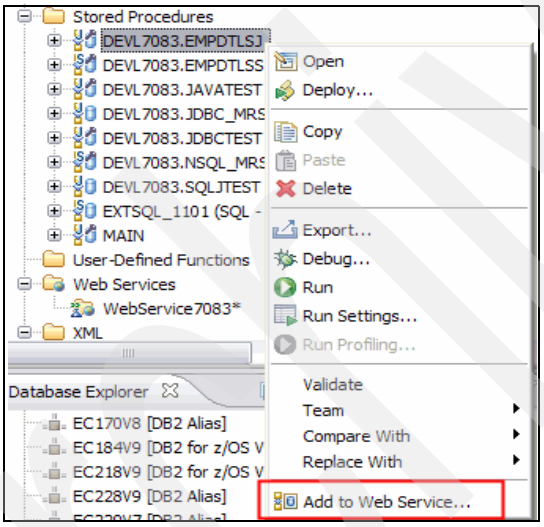

Figure 27-70 Add a stored procedure call to a Web service

- In the Add Operation to Web Services page, select **WebService7083** on the left panel. Click **>.** The Web service is moved to the right panel, as shown in Figure 27-71 on page 733. Click **Next.**

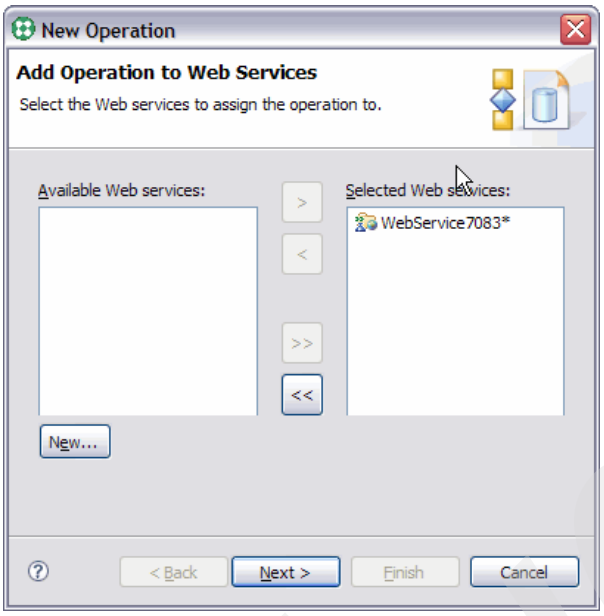

Figure 27-71 Select the Web Service to add this stored procedure to

- In the Name and Operator page, the name and operation are prefilled with the stored procedure's name and a generated CALL to the stored procedure. Click **Next.**

  Data Studio can generate detailed or generic XML schemas for stored procedures that accept non-varying input values and return result sets that are always the same. To make the XML schema as detailed as possible, Data Studio needs to know the structure of these unchanging result sets. This information is obtained by running the stored procedure and capturing the input and result set information. The XML schema is generated from this run.
- In the Generate XML Schema for Stored Procedure page shown in Figure 27-72 on page 733, click **Generate.**

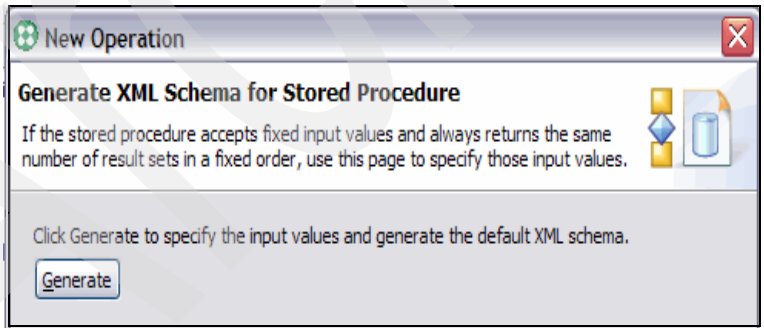

Figure 27-72 Generate XML Schema for stored procedure

- The Run Settings dialog is launched. Type `000100` in the Value column for EMPNO. Click **Finish.**
- Now, right-click the Web service, **WebService7083** and select **Build and Deploy**. This launches the Deploy a Web Service dialog shown in Figure 27-73 on page 734. You can specify the deploy options you want in this page.

Data Studio can optionally install the WebSphere Application Server Community Edition, which you can use when developing your Web services. For our example, we select **Build .war file only, do not deploy to server**. Click **Finish.**

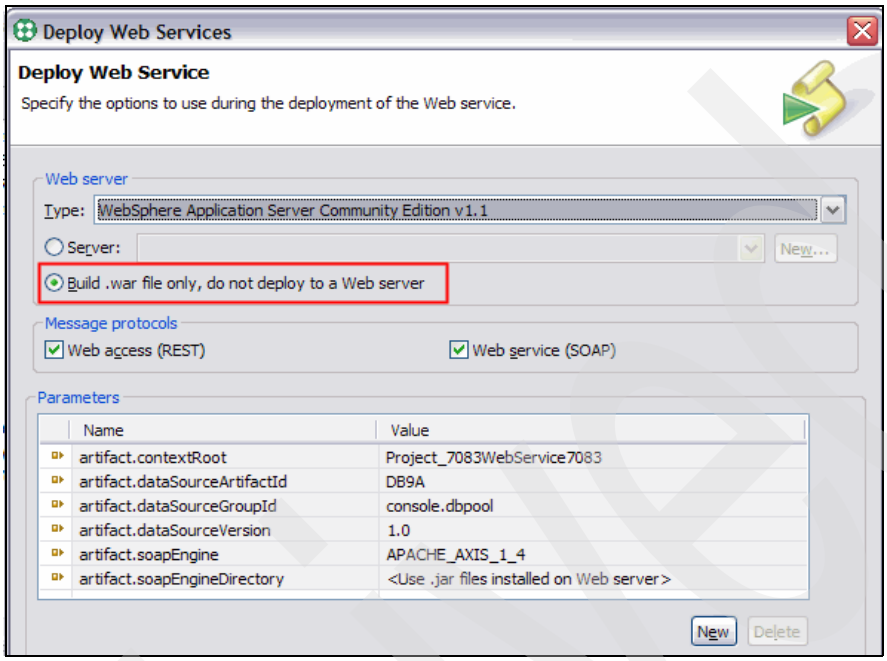

Figure 27-73 Specify options for deploying the Web service

Data Studio generates the Web service runtime files such as the WebSphere Definition Language (WSDL) file. The project folder is refreshed and the WSDL is added in the XML folder, as shown in Figure 27-74.

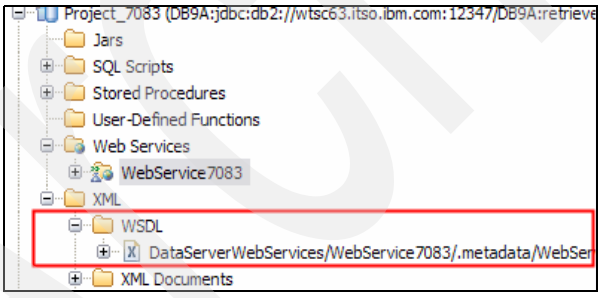

Figure 27-74 Generated WSDL file

28

Tools for debugging DB2 stored procedures

Today, multiple languages, multiple platforms, and multiple debugging tools are available for use with the development and debugging of DB2 Universal Database™ stored procedures. While this book focuses on DB2 for z/OS, many debugging tools' options for the distributed platforms (Linux, UNIX, and Windows) are also described in this chapter since they are often used across all platforms.

This chapter discusses the following topics:

- Debugging options at a glance
- The Unified Debugger
- Debugging SQL procedures on z/OS, Linux, UNIX, and Windows
- Debugging COBOL, PL/I, and C/C++ procedures on z/OS
- Debugging options for DB2 Java procedures on z/OS
- Debugging Java procedures on Linux, UNIX, and Windows

28.1 Debugging options at a glance

This section describes your options for debugging stored procedures on the different platforms. These options also depend on the programming language that you use.

Table 28-1 DB2 debugging options for z/OS

Language	DB2 Release	Client	Server
COBOL, PL/I, C/C++	All supported DB2 releases	Optional GUI: -IBM Rational Developer v7, (RDz) for System z -IBM Visual Age for Cobol -C/C++ Productivity Tools for z/OS	DB2 for z/OS V8 and V9 IBM Debug Tool
SQL - external	DB2 V8 and V9	Unified Debugger in IBM Data Studio; Unified Debugger in Developer Workbench (DWB) SQL Debugger in Development Center (DC) V8.2	DB2 V8 and V9 DSNTPSMP set up including all prerequisites (C compiler, WLM, RRS, REXX) member DSNTIJSD from DSN810 / DSN8910.SDSNSAMP
SQL - native	DB2 V9	Unified Debugger in IBM Data Studio; Unified Debugger in Developer Workbench	DB2 V9 DSN8910.SDSNSAMP (DSNTIJSD)
Java	V8, with PTF UK28560 applied	Unified Debugger in IBM Data Studio	DB2 V8
Java	V9	Unified Debugger in IBM Data Studio	DB2 V9

IBM Rational Developer

The IBM Rational Developer for V7 for System z (RDz v7) provides an application development framework for COBOL, PL/I, and Assembler. It provides remote access to z/OS environments or source code. It is possible to debug C and C++ stored procedures using the Debug Perspective. RDz V7 can be used with the IBM Debug Tool for debugging COBOL, PL/I, C, and C++ stored procedures, and provides the GUI on the workstation.

IBM Debug Tool

The IBM Debug Tool runs on the z/OS server. The IBM Debug Tool Version 8.1 is now part of the IBM Debug Tool Utilities and Advanced Functions V8.1, and no longer needs to be ordered separately. It provides an interactive, full-screen, IBM 3270 system-based terminal interface with three windows: a monitor, source, and log window, that enables single-step debugging, dynamic patching and breakpoints. When used with a workstation tool, such as RDz v7, developers can set breakpoints, step code, change execution points and data and fully debug logic execution.

VisualAge COBOL

If you have VisualAge COBOL installed on your workstation and the Debug Tool installed on your z/OS system, you can use the VisualAge COBOL Edit/Compile/Debug component with

the Debug Tool to debug COBOL stored procedures that run in a WLM-established stored procedures address space and have been prepared for debug.

C/C++ Productivity Tools for z/OS

If you have the C/C++ Productivity Tools for z/OS installed on your workstation and the Debug Tool installed on your z/OS system, you can debug a C or C++ stored procedure that runs in a WLM-established stored procedures address space. The code against which you run the debug tools is the C source program that is produced by the program preparation process for the stored procedure.

For more information on debugging COBOL/C/C++ stored procedures using the above products, see *DB2 V9.1 for z/OS, Application Programming and SQL Guide*, SC18-9841-01.

DSNTPSMP

DSNTPSMP is the DB2 SQL Procedure processor that builds SQL stored procedures on DB2 for z/OS V8 (see 27.2.8, “Overview of routine development with IBM Data Studio” on page 663 for details).

Unified Debugger

The Unified Debugger is a single debugger that can be used to debug SQL and Java stored procedures on DB2 for z/OS, DB2 on iSeries and DB2 on Linux, UNIX and Windows. It is included in IBM Data Studio and Developer Workbench (DWB) products. See the IBM Data Studio or DWB online help for assistance in debugging SQL and Java stored procedures.

While this book focuses on DB2 for z/OS stored procedures, many customers prototype their applications on DB2 for Windows or UNIX, including stored procedure development. The distributed platforms provide some additional options for debugging Java and SQL stored procedures. The IBM Data Studio assists in developing *cross-platform* SQL, and Java stored procedures. When comparable, DB2 for z/OS DDL and stored procedure code can be defined on DB2 for Windows and UNIX; first-level development, testing, and debugging can be performed in the distributed environment. Table 28-2 summarizes the DB2 debugging options for the distributed platforms.

Table 28-2 DB2 debugging options for the distributed platforms

Database Server\Debug Server[a] are below and debug clients are on the right	**IBM Data Studio, DWB**	**RAD V7**	**DC**	**.Net Studio**
DB2 9 for LUW, DB2 9 for z/OS and DB2 on iSeries V5R4 support external and native SQL and Java procedures	yes	yes	no	SQL only
DB2 V8 for LUW, FP14 or higher, DB2 V8 for z/OS[b] - supports SQL procedures	yes[c]	yes	no	yes
DB2 V8 for LUW, FP12 or lower, DB2V8 for z/OS prior to applying APAR PK41138 - supports SQL procedures	no	no	yes	yes

a. Debug Server - Debugger engine that is integrated with the database server. It is available as part of DB2.
b. Need to apply APAR PK41138 - Unified Debugger V9 FP2 on DB2 for z/OS V8
c. Customers must start Session Manager on the client side.

28.2 The Unified Debugger

Users investing in DB2 application development have a need for formal debugging facilities. This need is even intensified when working on DB2 stored procedure development since the code runs in isolation at a DB2 server.

For classic languages, such as COBOL and C on the one hand, the compiler products and their associated runtime facilities provide debug capabilities. On the other hand, we have interpreted languages such as SQL and Java where debugging sometimes has caused some problems. The Unified Debugger that has been introduced with DB2 9 focuses on these newer languages.

With the Unified Debugger, you can observe the execution of SQL procedure code, set breakpoints for lines, and view or modify variable values. The Unified Debugger supports external and native SQL procedures, including nested stored procedures.

The Unified Debugger builds upon and includes the SQL Debugger technology from DB2 Version 8.2. The name has been changed to Unified Debugger to embrace the unified support for both Language SQL and Java stored procedure debugging. Following this, one advanced capability that is now offered to you through the Unified Debugger is the ability to debug nested SQL or Java stored procedures sharing the same client application call stack. This means that users debugging a Java routine can step into and debug a called SQL procedure.

The Unified Debugger itself is *middleware*. It originates from, and is serviced by, DB2 9 for Linux, UNIX and Windows. This middle layer code is distributed to the DB2 server platforms as object code (for running on each of the various operating systems that host DB2). There are three basic elements:

- The Unified Debugger server library - APIs that provide the interface for the DB2 servers and the supported routine objects.
- The Session Manager and the Unified Debugger routers stored procedures that implement the client interface to the Unified Debugger at DB2 servers. This layer is independent of any particular platform or DB2 server type.
- A debug client that is written to support the Unified Debugger middleware. This client is only based on LUW clients.

28.2.1 Processing overview of the Unified Debugger

The IBM Data Studio (DS) or Developer Workbench (DWB) can be used to debug both native and external SQL stored procedures and Java stored procedures against DB2 for z/OS V9; and DB2 for z/OS V8 external SQL stored procedure debugging. A quick overview of the process follows.

1. Decide where to run the Session Manager. DS/DWB can start the Session Manager (SM) either on the workstation, such as the client's workstation, or the z/OS server. You inform DS/DWB of this decision in **Window → Preferences → Run/Debug → DB2 Stored Procedure Debugger.** Select **Run the session manager on each connected server** or **Use already running session manager**. In the latter selection, the user must start the SM and capture the IP address or port of the SM.
2. Create, Edit, and Deploy with Debug Enabled an SQL or Java stored procedure.
3. Debug rather than Run the stored procedure. You do this by right-clicking the stored procedure and then selecting **Debug**. DS/DWB will contact the designated SM prior to calling the stored procedure.
4. The DB2 server hosting the stored procedure engages the SM via the debug server, establishing the debug session.

5. DS/DWB will then switch to the Debug Perspective, signalling that debugging is underway.
6. For Java stored procedures, DS/DWB drives the debug session by directly communicating with the JVM at the server.
7. For SQL procedure debugging, the SM is the central coordinating agent.
   – 7a: DS drives the debug session with direct TCP/IP communication to the SM.
   – 7b: DWB drives the debug session with indirect communication to the SM. DWB calls the debug router procedures that communicate with the SM via TCP/IP.
8. SQL procedures keep the debug server informed of program status. The debug server coordinates control from DS/DWB via TCP/IP communication with the SM.

The chart in Figure 28-1 describes the processing flow for debugging stored procedures using the IBM Data Studio Unified Debugger.

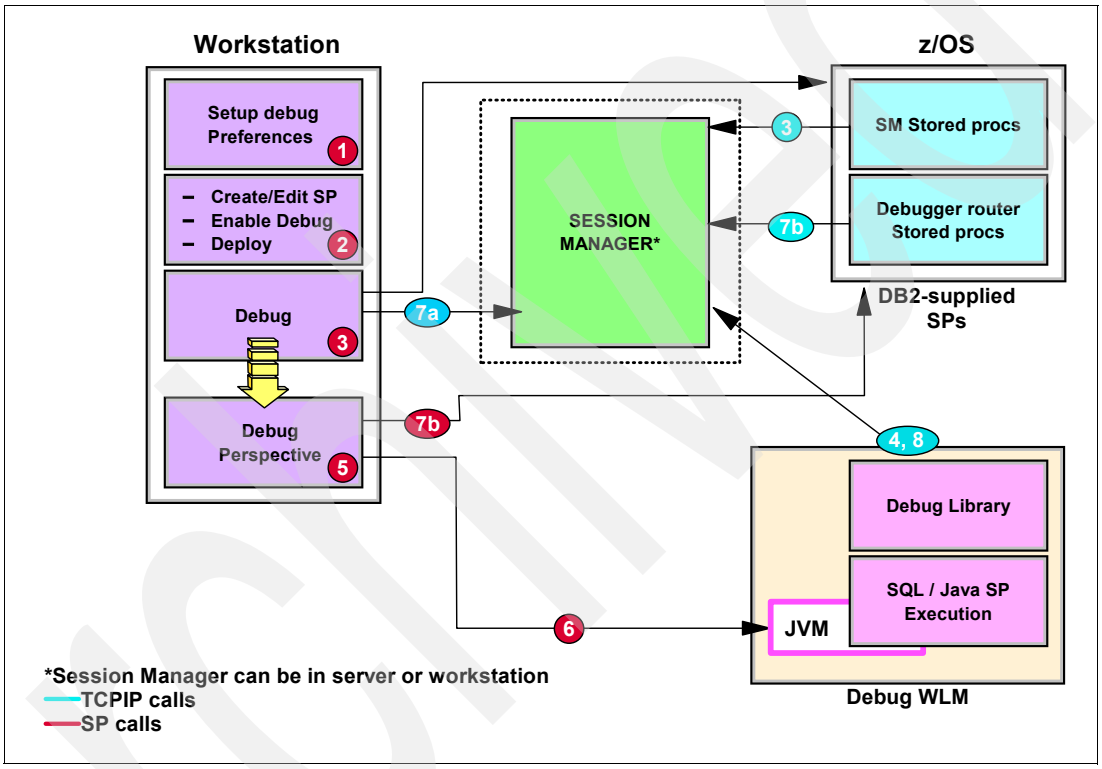

Figure 28-1 Processing overview - Unified Debugger with DB2 9 for z/OS

28.2.2 Setting up the Unified Debugger components

There are a couple of steps that you need to go through in order to be able to use the Unified Debugger functionality.

Install the IBM Data Studio or Developer Workbench Client

To install the IBM Data Studio or DWB, just execute the LAUNCHPAD.EXE file that comes with the product (27.2.1, “Client setup” on page 648). Unlike some other products, there is hardly anything you need to decide during your installation. The IBM Data Studio is available as a free download from:

https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=swg-ids

The Unified Debugger is also available in Developer Workbench (DWB). DWB is also available as a download, or part of DB2 9 on LUW. The DWB download site is:

https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=swg-dm-db2dwb

Run the post-install job DSNTIJSD

You must run the post-install job DSNTIJSD to create DB2 server objects that are required when using the Unified Debugger or the SQL Debugger.

When we say that this is a post-install job, it means that after stepping through the installation or migration panels, you will not find a customized version of this job in your hlq.NEW.SDSNSAMP library. Instead, you must copy this job from the DSN910.SDSNSAMP library and customize it manually. A description of what you must change in order to get it to work is provided within the job.

If you run this job, it will create the following DB2-supplied stored procedures for you:

- DB2DEBUG.DEBUGGERLEVEL
- DB2DEBUG.CREATE_SESSION
- DB2DEBUG.DESTROY_SESSION
- DB2DEBUG.QUERY_SESSION
- DB2DEBUG.LIST_SESSION
- DB2DEBUG.PUT_COMMAND
- DB2DEBUG.GET_REPORT
- SYSPROC.DBG_INITIALIZECLIENT
- SYSPROC.DBG_TERMINATECLIENT
- SYSPROC.DBG_SENDCLIENTREQUESTS
- SYSPROC.DBG_SENDCLIENTCOMMANDS
- SYSPROC.DBG_RECVCLIENTREPORTS
- SYSPROC.DBG_ENDSESSIONMANAGER
- SYSPROC.DBG_PINGSESSIONMANAGER
- SYSPROC.DBG_LOOKUPSESSIONMANAGER
- SYSPROC.DBG_RUNSESSIONMANAGER

Make sure that you specify an appropriate WLM environment for every single stored procedure. We recommend that you use NUMTCB > 5 for the application environment that you are going to assign to them.

Note: Stored procedure DBG_RUNSESSIONMANAGER is only available for DB2 V9 on z/OS and must run as an authorized program in an authorized environment.

Run the post-install job DSNTIJMS

This post-install job creates objects required for the DB2 JDBC and ODBC metadata methods.

Again, you will not find a customized version of this job in your hlq.NEW.SDSNSAMP library. Instead, copy this job from the DSN910.SDSNSAMP library and customize it manually. A description of what you must change in order to get it to work is provided within the job.

As a result of executing this job, you will have some packages bound on your system and the following stored procedures will be created:

- SYSIBM.SQLCOLPRIVILEGES
- SYSIBM.SQLCOLUMNS
- SYSIBM.SQLFOREIGNKEYS
- SYSIBM.SQLPRIMARYKEYS
- SYSIBM.SQLPROCEDURECOLS
- SYSIBM.SQLPROCEDURES
- SYSIBM.SQLSPECIALCOLUMNS
- SYSIBM.SQLSTATISTICS
- SYSIBM.SQLTABLEPRIVILEGES

- SYSIBM.SQLTABLES
- SYSIBM.SQLGETTYPEINFO
- SYSIBM.SQLUDTS
- SYSIBM.SQLCAMESSAGE

Note: The JDBC metadata routine setup job (DSNTIJMS) has no direct relationship to debugging. Rather, that job is related to setup for JDBC (JCC connected) clients, such as DWB, merely to support operations such as run, meaning the calling of a stored procedure.

Set up the WLM environment

A WLM procedure must be defined for executing the stored procedure. For SQL stored procedures, this WLM procedure optionally includes a DD card for collecting information during the debug session.

- For SQL stored procedures, a //PSMDEBUG statement can be added in the WLM procedure. The //PSMDEBUG statement defines a physical sequential data set with RECFM=VBA, LRECL=4096. This data set should only be included in the WLM procedure when requested by IBM Level 2 as the //PSMDEBUG statement presence causes records to be written to it for SQL Debugger problems that will impact performance.
- For Java stored procedures, a //JSPDEBUG statement can be added in the WLM procedure. The data set definition and usage of the file referenced in the //JSPDEBUG statement is the same as that used in the //PSMDEBUG statement.

Set up the Session Manager

You can have the Session Manager (SM) run on any platform that you prefer. The SM is the middleware that handles the debug session between the debug client and the debug server. We recommend using the SM on the server. We discuss setting up the Session Manager on the client in 28.4.2, “Prerequisites and setup” on page 761.

If you are using .Net on your client, or you want to set up the SM on z/OS, follow the steps described below. You can think of the SM as a daemon that in our case is running as a z/OS started task that is waiting for work to perform. Setting up the SM on z/OS basically consists of three steps, which we describe next.

Step 1: Define the started task to RACF

The RACF definitions shown in Example 28-1 must be added to your z/OS security system.

Example 28-1 Define DB2UDSMD to RACF

```
//UDBG1    JOB ('RACF'),CLASS=A,MSGCLASS=A,MSGLEVEL=1,
//         USER=********, PASSWORD=*******
//*--------------------------------------------------------------------
//* Define the Unified Debugger Session Manager Started Task to RACF.
//* A security manager ID must be used to perform the definitions.
//*
//* The STARTED task is defined by a RACF profile named DB2UDSMD.**
//* USRT005 will be the ID associated with this Started Task.
//* Since the task will run a java program from OMVS, also assign an
//* OMVS segment definition to the user (UID, home dir, etc.)
//* Finally, activate the STARTED task definition in current memory.
//*--------------------------------------------------------------------
//RACFDEF  EXEC TSOBATCH
//SYSTSIN DD *
  ALTUSER USRT005 OMVS( UID(5) HOME('/u/usrt005') PROGRAM('/bin/sh') )
  RDEFINE STARTED DB2UDSMD.** STDATA(USER(USRT005))
  SETROPTS RACLIST(STARTED) REFRESH
```

```
END
```

The STARTED task is defined by a RACF profile named DB2UDSMD.**.

Important: It is mandatory to use DB2UDSMD as the started task name. The Session Manager is not tied to a specific DB2 subsystem, nor is it tied to any DB2 subsystem at all.

USRT005 was designated to be the ID associated with this started task. Since the task will run a Java program from OMVS, also assign an OMVS segment definition to the user (that is, UID, home dir, and so on). Finally, activate the STARTED task definition in current memory.

Step 2: Step 2: Create the environment settings

The job shown in Example 28-2 is used to create a file in the HFS to hold the environment settings used by the Session Manager's started task.

Example 28-2 Job to create a file in HFS to hold the environment settings

```
//UDBG2    JOB 'USER=$$USER','<USERNAME:JOBNAME>',CLASS=A,
//         MSGCLASS=A,MSGLEVEL=(1,1), REGION=4096K,
//         USER=USRT005,PASSWORD=*******
//*--------------------------------------------------------------------
//* Create a file in the HFS to hold the Environment settings used when
//* the Unified Debugger Session Manager runs as a Started Task on z/OS
//*
//* USRT005 is the ID associated with the Started Task.
//* Place the file in that users home directory.
//* Name the file DB2UDSMDenvironment
//*--------------------------------------------------------------------
//*--------------------------------------------------------------------
//* Create a file in the HFS from inline data using COPY
//*--------------------------------------------------------------------
//OCOPY    EXEC PGM=IKJEFT01,DYNAMNBR=30
//SYSTSPRT DD SYSOUT=*
//HFSOUT   DD  PATH='/u/usrt005/DB2UDSMDenvironment',
//             PATHOPTS=(OWRONLY,OCREAT,OAPPEND,OTRUNC),
//             PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIROTH)
//INLINE   DD *
#--------------------------------------------------------------------
# Environment settings for running the Unified Debugger Session Manager
# * _BPX_BATCH_SPAWN=NO
# * _BPX_SHAREAS=YES
#     Arrange for the JVM to run in the same address space. This avoids
#     launching 2 additional address spaces for the Started Task.
# * ENV=
#     Reference this file. Insulates from PATH and CLASSPATH changes
#     present in etc/profile.
# * PATH=
#     The location of the desired JAVA release, and system binaries.
# * CLASSPATH=
#     The location of the UDBG Session Manager jar file
# * JAVA_COMPILER=NONE
#     Disable the JIT. The Started Task runs the Session Manager only
#     one time, so disabling this saves space that will not be used.
#--------------------------------------------------------------------
_BPX_BATCH_SPAWN=NO
_BPX_SHAREAS=YES
ENV=/u/usrt005/DB2UDSMDenvironment
PATH=/usr/lpp/java150/J5.0/bin:/bin
```

```
CLASSPATH=/usr/lpp/db2/db2910_base/classes/db2dbgm.jar
JAVA_COMPILER=NONE
//SYSTSIN DD *
OCOPY INDD(INLINE) OUTDD(HFSOUT) TEXT
//
```

Create a file in the HFS to hold the environment settings used when the Unified Debugger Session Manager runs as a started task on z/OS.

An ID must be designated to be associated with the started task. Suppose that USRT005 is that ID. Place a file in that user's home directory to serve as an execution environment profile. The file must point to the location of the Session Manager jar file, db2dbgm.jar. Name the file something distinctive, such as DB2UDSMDprofile.

Note: Ensure that the CLASSPATH points to where the db2dbgm.jar file is installed. Otherwise, the started task, DB2UDSMD, will not come up.

In both Example 28-1 on page 741 and Example 28-2 on page 742 we have used USRT005 as the place holder for the ID associated with started task DB2UDSMD that you must change to your specific situation.

The use of a named HFS application profile is suggested for simple setup and segregation of duties. At a minimum, it needs to define the CLASSPATH to the Session Manager Java program. Other settings to tune the Java execution environment can be included. Note that BPXBATCH reads the STDENV file, so no shell script symbol substitution can be utilized here. Symbol substitution processing is only available to the user profile (.profile) for the started task user ID and scripts executed from the shell command line.

The Session Manager is independent of DB2, so it can run anywhere in the network. But the server platform (that is, the operating system that the stored procedure that you want to debug runs) is often a better default choice than running at the client workstation. The session Manager JAR file is now distributed on all server platforms, so it does not have to be obtained, downloaded, sent, pulled, pushed, or transported by you.

Step 3 - Create a started task JCL

Create the started task JCL for DB2UDSMD and place it in the system proclib (Example 28-3). This is used to launch the Unified Debugger Session Manager on z/OS. USRT005 is the ID associated with this started task, as defined in the RACF STARTED class profile DB2UDSMD.**.

Example 28-3 Sample started task JCL for the Session Manager on z/OS

```
//UDBG3    JOB 'USER=$$USER','<USERNAME:JOBNAME>',CLASS=A,
//         MSGCLASS=A,MSGLEVEL=(1,1), REGION=4096K,
//         USER=********,PASSWORD=*******
//*--------------------------------------------------------------------
//* Create the Started Task JCL for DB2UDSMD. A START command will then
//* be able to launch the Unified Debugger Session Manager on z/OS.
//* USRT005 is the ID associated with the Started Task, as defined in
//* the RACF STARTED class profile DB2UDSMD.**
//*--------------------------------------------------------------------
//*--------------------------------------------------------------------
//* Use IEBUPDTE to write a JCL member into SYS1.PROCLIB
//*--------------------------------------------------------------------
//WRITEJCL EXEC PGM=IEBUPDTE,PARM=NEW
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
```

```
//SYSUT2   DD DISP=SHR,DSN=SYS1.PROCLIB
//SYSIN    DD DATA
./  ADD NAME=DB2UDSMD
//DB2UDSMD PROC PORT=4553,TIMEOUT=60
//*
//*    DB2 Unified Debugger Session Manager DAEMON FOR OPENEDITION
//*
//*    This JCL assumes no .profile exists for the user.
//*
//* Environment settings (PATH, CLASSPATH) come from STDENV file.
//*
//DB2UDSMD EXEC PGM=BPXBATCH,DYNAMNBR=128,REGION=0M,TIME=1440,
//             PARM='SH date;java com.ibm.db2.psmd.mgr.Daemon -timeout
//             &TIMEOUT -port &PORT -log /dev/null;date'
//STDOUT   DD  PATH='/tmp/DB2UDSMD.stdout',
//             PATHOPTS=(OWRONLY,OCREAT,OAPPEND,OTRUNC),
//             PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIROTH)
//STDERR   DD  PATH='/tmp/DB2UDSMD.stdout',
//             PATHOPTS=(OWRONLY,OCREAT,OAPPEND,OTRUNC),
//             PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIROTH)
//STDENV   DD  PATH='/u/usrt005/DB2UDSMDenvironment',
//             PATHOPTS=ORDONLY
//             PATHOPTS=ORDONLY
./  ENDUP
/*
```

Note: BPXBATCH does not derive PATH from STDENV. For the Session Manager, PATH only needs to point to the Java runtime. One approach is to specify the PATH directly on the command line, as shown in Example 28-2 on page 742. Another method requires the use of a shell script profile (.profile) for the started task user, which we have not included in this documentation. Note that the job step options in the JCL as shown, which includes the OMVS shell command, were carefully arranged to efficiently utilize the limited space available. The space is limited to 100 characters in total. As shown in Example 28-3 on page 743, there are still about 18 characters left to adjust the path specification for Java.

Some considerations when coding the DB2UDSMD started task

1. The PARM field

   When running the above task, you may get the following error:

   ```
   IEF642I EXCESSIVE PARAMETER LENGTH IN THE PARM FIELD
   ```

   This is because the PARM filed is limited to 100 characters only. To fix this, you can do either of the following options.

   – If the PARM field does not contain comma-delimited parameters and the PARM field contains less than 100 characters, enter PARM field data through pos. 71 and then continue exactly in pos. 16 of the next line and enclose entire PARM field with apostrophes as shown in Example 28-4.

Example 28-4 Coding long PARM field into next line

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
//STEP3___EXEC PGM=IEFBR14,PARM='LONG PARAMETER FIELD WITHOUT COMMA DEL
//_____________IMITERS - CONTINUED IN COLUMN 16 OF THE NEXT LINE'
```

   – If PARM field doesn't contain comma-delimited parameters, and the PARM field is greater than 100 characters, use an STDPARM DD card. This workaround is available with APAR OA11699, see
   http://www-1.ibm.com/support/docview.wss?uid=isg1OA11699

The STDPARM file can accept 65.356 bytes.

So, for example, Create file ADMF001.DB2UDSMD.PARMOMVS which contains the following data:

```
SH date;/ZOS17TC/usr/lpp/java/j1.4_64/bin/java com.ibm.db2/psmd.mgr.Daemon -timeout
&TIMEOUT -port &PORT;date
```

Code the DB2UDSMD procedure as shown in Example 28-5.

Example 28-5 DB2UDSMD procedure with STDPARM

```
//DB2UDSMD  EXEC PGM=BPXBATCH,DYNAMNBR=128,REGION=0M,TIME=1440
//STDPARM  DD DISP=SHR,DSN=ADMF001.DB2UDSMD.PARMOMVS
```

2. STDERR and STDOUT files

   The HFS files pointed to by STDERR and STDOUT need to have the property access authorizations. Otherwise, the DB2UDSMD task will not start, and the user may see messages similar to Example 28-6 in the console.

Example 28-6 Authorization error

```
IEF403I DB2UDSMD - STARTED - TIME=15.52.36
ICH408I USER(DB2UDSMD) GROUP($STCGRP ) NAME(DB2UDSMD
  /app/db2/tmp/DB2UDSMD.stdout
  CL(DIRSRCH ) FID(01C1F7D7D9F0F4009621000000000003)
  INSUFFICIENT AUTHORITY TO STAT
  ACCESS INTENT(--X)  ACCESS ALLOWED(OTHER      ---)

IEF403I DB2UDSMD - STARTED - TIME=16.05.08
ICH408I USER(DB2UDSMD) GROUP($STCGRP ) NAME(DB2UDSMD
  /app/db2/tmp CL(DIRSRCH ) FID(01C1F7D7D9F0F4009621000000000
  INSUFFICIENT AUTHORITY TO CHDIR
  ACCESS INTENT(--X)  ACCESS ALLOWED(OTHER      ---)
  EFFECTIVE UID(0000000005)  EFFECTIVE GID(0000000100)

 IEF403I DB2UDSMD - STARTED - TIME=16.29.18
 ICH408I USER(DB2UDSMD) GROUP($STCGRP ) NAME(DB2UDSMD
   /app/db2/tmp/DB2UDSMD.stdout
   CL(DIRACC  ) FID(00000001000000010000000000000000)
   INSUFFICIENT AUTHORITY TO OPEN
   ACCESS INTENT(-W-)  ACCESS ALLOWED(OTHER      R-X)
```

To fix this, change the permission bits for the /app/db2 directory from 700 to 755. DB2UDSMD does not have authority to change directory at the /app/db2 directory. Every directory in the path has to have the permissions in order to change directory.

To change to the /app/db2/tmp, the userid has to have directory permission to the:

- /app directory,
- /app/db2 directory, and
- /app/db2/tmp directory.

Start the Session Manager started task on z/OS

After you have executed all the JCLs, you can now test if the Session Manager is working.

1. From the z/OS console, issue a START DB2UDSMD,TIMEOUT=1 operator command to start your Session Manager.
2. Wait a few minutes. The Session Manager will timeout after 1 minute of inactivity
3. Check the contents of the HFS file in STDOUT, /tmp/DB2UDSMD.stdout. Figure 28-2 shows an example of a good output from the Session Manager.

```
Mon Mar 2 12:48:32 PST 2009
args[0]: -timeout
args[1]: 1
args[2]: -port
args[3]: 4553
args[4]: -log
args[5]: /dev/null
Code Level: 070418
Debug Session Manager started on IP: 9.30.88.135 - port: 4553
idleTimeOut: 1
Mon Mar 2 12:49:37 PST 2009
```

Figure 28-2 Example of a successful output from the Session Manager

Grant DEBUGSESSION privilege

Grant the DEBUGSESSION privilege to the user that runs the debug client. The DEBUGSESSION privilege is a new system authorization. Refer to the new catalog column DEBUGSESSIONAUTH in table SYSIBM.SYSUSERAUTH to obtain information about who has already been granted this privilege.

Tip: For more information about the Unified Debugger, see information about the DB2 Developer Workbench in the DB2 Database for Linux, UNIX, and Windows information center at:

`http://publib.boulder.ibm.com/infocenter/db2help/index.jsp`

Prepare your stored procedures for debugging

After you have successfully set up the environment for debugging your SQL stored procedures, you must now decide for every single stored procedure if you want to debug it.

For a native SQL procedure, define the procedure with the ALLOW DEBUG MODE option and the WLM ENVIRONMENT FOR DEBUG MODE option.

For an external SQL procedure, use DSNTPSMP or the IBM Data Studio to build the SQL procedure with the BUILD_DEBUG option.

For a Java procedure, define the procedure with the ALLOW DEBUG MODE option, select an appropriate WLM environment for Java debugging, and compile the Java code with the -G option.

28.3 Debugging SQL procedures on z/OS, Linux, UNIX, and Windows

The IBM Data Studio and its predecessor, Developer Workbench, provide the ability to debug SQL stored procedures on DB2 for z/OS V8[1] and V9 through the Unified Debugger. This provides the debugging user interface for debugging SQL and Java stored procedures with DB2 for z/OS V9, and SQL stored procedures with DB2 for z/OS V8. The examples shown in this section also apply when debugging SQL and Java stored procedures on Linux, UNIX, and Windows.

This section describes how to get you started with debugging an SQL stored procedure:

1. Setting up the Session Manager

[1] with PTF PK41138 applied

2. Creating SQL stored procedures for debugging
3. Debugging SQL stored procedures
4. Defining the EMPDTLSS SQL case study for debugging
5. Debugging the EMPDTLSS SQL case study

28.3.1 Setting up the Session Manager

Setting up the Session Manager on a client

To set up the Session Manager on a workstation where the IBM Data Studio is installed:

1. Open a command window and go to the directory where the IBM Data Studio is installed, for example C:\Program Files\IBM\SDP70\dwb\bin>. From this directory, run db2dbgm.bat. Note the IP address and port of the Session Manager, as shown in Figure 28-3.

```
C:\Program Files\IBM\SDP70\dwb\bin>db2dbgm.bat
args[0]: -port
args[1]: 4554
args[2]: -timeout
args[3]: 50
Code Level: 070418
Debug Session Manager started on IP: 9.30.28.113 - port: 4554
idleTimeOut: 50
```

Figure 28-3 Debug Session Manager startup

2. Launch the IBM Data Studio. Click **Window** → **Preferences** → **Run / Debug** → **DB2 Stored Procedure Debugger**. In this page, click **Use already running session manager**. Fill in the *Host* IP address and *Port* number of the session manager as shown in Figure 28-4 on page 748.
3. Click **Apply** → **OK**.

Setting up the Session Manager on the server

In 28.2, “The Unified Debugger” on page 738, we gave the steps for setting up the Session Manager on the z/OS server. If your client has a firewall, then it may not be feasible for the server to initiate communication with the Session Manager in the client. You may then want to use the Session Manager in the z/OS server.

To use the Session Manager, in the IBM Data Studio preferences shown in Figure 28-4 on page 748, click **Run the session manager on each connected server**.

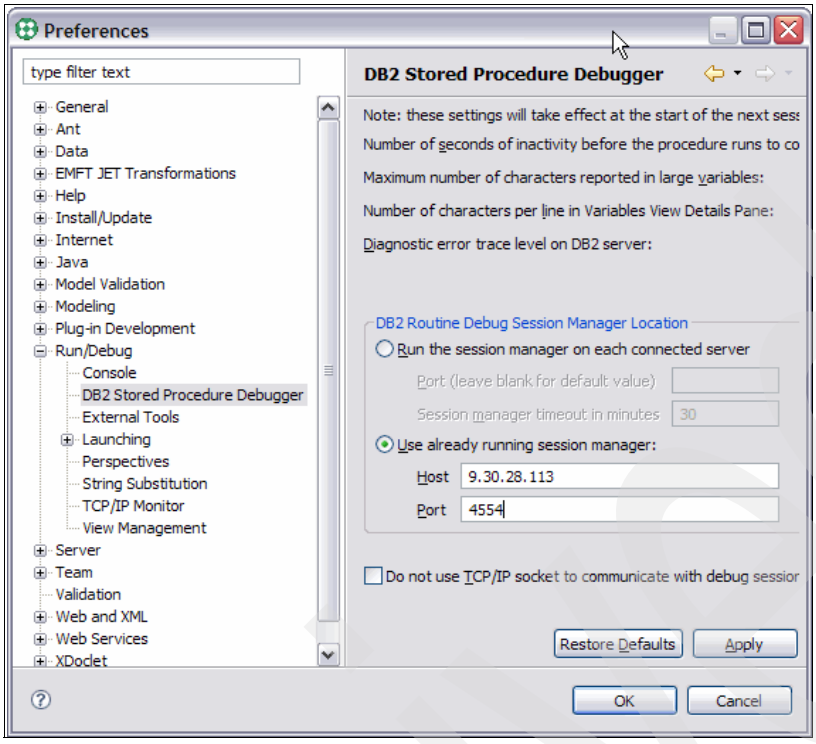

Figure 28-4 Preferences for using the client Session Manager

28.3.2 Creating SQL stored procedures for debugging

IBM Data Studio can be used to create SQL stored procedures, though it is not required. However, it must be used to build the stored procedure for debug. First, we describe the general steps to create an SQL stored procedure for debugging, followed by our EMPDTLSS case study:

1. Start IBM Data Studio from **Start** → **All Programs** → **IBM Software Development Platform** → **IBM Data Studio** → **IBM Data Studio.**
2. Create a new or reconnect to an existing DB2 for z/OS V9 server.
3. Create a new or open an existing Data Development Project.
4. Optionally update the Unified Debugger time-out value.

   The default time-out value is 300 seconds. This means that while in debug mode, after 300 seconds of inactivity, the Debugger terminates and any locks held by the stored procedure being debugged are released. The SQL stored procedure will run to completion.

   This value can be changed from the Preferences pages, as shown in Figure 28-4 on page 748. In this Preference page, this is the *"Number of seconds of inactivity before the procedure runs to completion".* We will use the default value for our case study. Click **OK** to accept the default value.
5. Create a new stored procedure. During the creation of the SQL stored procedure using the New Stored Procedure wizard, check the **Enable debugging** check box in the Deploy Options page. See 27.5.1, "Creating a new native stored procedure using the wizard" on page 689 and 27.5.2, "Creating an external SQL stored procedure from the wizard" on page 694 for detailed steps on how to create a new SQL stored procedure.

The IBM Data Studio Data Output window includes a message indicating whether the stored procedure performed a build for debug. For an external SQL stored procedure, the build utility

reports a BUILD_DEBUG function, and whether it was successful or not, as shown in Example 28-7.

Example 28-7 BUILD_DEBUG function was completed successfully

```
Build utility function requested: BUILD_DEBUG
DSNT540I  DB9AWLMR WAS REFRESHED BY PAOLOR5  USING AUTHORITY FROM SQL ID PAOLOR5 : 0
PAOLOR5.EXTSQL_1101 - Deploy for debug successful.
```

For a native stored procedure, the output displays the beginning message `Deploy for debug started` and ends with the message `Deploy for debug successful.`

28.3.3 Debugging SQL stored procedures

Once the SQL stored procedure is successfully built for debug mode, we can debug the stored procedure. In 28.3.1, “Setting up the Session Manager” on page 747, we discussed how to launch the Session Manager. It is now ready to handle the communication between IBM Data Studio and the Unified Debugger code on the server. You can start debugging from three launch points:

- From Data Project Explorer, select the SQL stored procedure and right-click **Debug**.
- With the SQL stored procedure opened in the Routine Editor (which can be done by selecting the SQL stored procedure in Data Project Explorer, right-click **Open**), click the **Source** tab. Right-click on whitespace → **Debug As** → **Debug**.
- Still from the Routine Editor → Configuration tab and then click **Debug**, as shown in Figure 28-5.

Figure 28-5 Starting the Debugger from the Routine Editor

When the stored procedure is launched in debug mode, the user is prompted to switch into the Debug Perspective. Click **Yes**. In the Debug Perspective, you can set breakpoints in the prefix area to the left of a valid statement, monitor and change the values of variables, and interactively debug.

28.3.4 Defining the EMPDTLSS SQL case study for debugging

EMPDTLSS was initially created without the tooling, directly on DB9A, the DB2 for z/OS V9 server described in 14.2.6, “Passing parameters” on page 243.

In Chapter 3, “Our case study” on page 23, we downloaded the source of this stored procedure from the additional materials link in this book to our workstation. This SQL stored procedure can be brought into IBM Data Studio in one of two ways:

- Using the Import wizard
- Using the editor

We will show both ways.

Using the Import wizard

From the Project_7083 project, select the **Stored Procedure** folder and right-click **Import**, as shown in Figure 28-6.

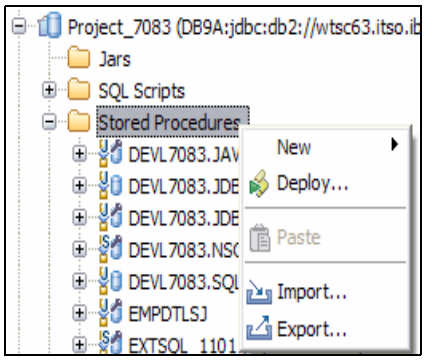

Figure 28-6 Import a stored procedure into the project

The Import wizard is opened and the Import Stored Procedure window is displayed. Select **File System** → **Browse** to point to the location of the source file in the file system, as shown in Figure 28-7 on page 750.

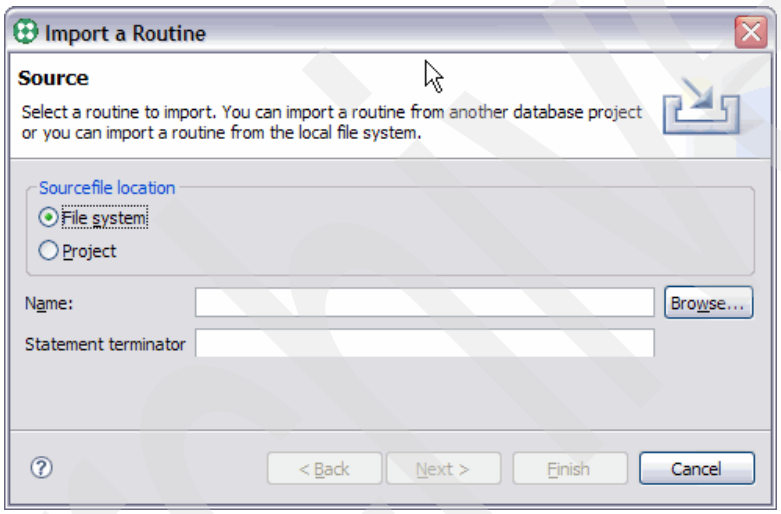

Figure 28-7 Import wizard start page

The Import wizard includes the following six UI pages to be completed. The wizard parses the imported source to fill in the appropriate fields when possible:

- Source

  Click **Browse** at the right of the Name field to locate the source EMPDTLSS.sql that we had previously saved on our workstation. Click **Next** to continue.

- Entry Points

  Any routines that are included in this stored procedure are listed by the wizard where the developer can select the routine to be used as the main entry point. The EMPDTLSS stored procedure has one routine, and that routine is selected. Click **Next** to continue.

- Parameters

  One input parameter and nine output parameters are listed. Click **Next** to continue.

- Options

  The COLLID of DEVL7083 is automatically filled in from our source. Next, select **Advanced** on this window, which opens the z/OS Options window. The WLM ENVIRONMENT name included in the source is automatically filled in. Replace this with

DB9AWLMR, which is the WLM Environment for external SQL stored procedures in our server. Click **OK** to end the Advanced Options window. Click the **Enable debugging** checkbox causing our SQL stored procedure to be built for debugging. Click **Next** to continue.

Note: When connected to a DB2 V9 server, IBM Data Studio assumes that if the imported SQL stored procedure contains a WLM Environment, then the stored procedure is an EXTERNAL type, rather than Native. You need to manually edit the CREATE PROCEDURE DDL before importing and add the FENCED keyword to correctly identify this as an External SQL procedure.

- Summary

  The above settings are summarized. Click **Finish** to build the stored procedure for debug. Example 28-8 shows the modified listing.

Example 28-8 Modified EMPTDTLSS source for IBM Data Studio to build/debug on DB9A

```
CREATE PROCEDURE DEVL7083.EMPDTLSS
(
 IN  PEMPNO        CHAR(6)
,OUT PFIRSTNME     VARCHAR(12)
,OUT PMIDINIT      CHAR(1)
,OUT PLASTNAME     VARCHAR(15)
,OUT PWORKDEPT     CHAR(3)
,OUT PHIREDATE     DATE
,OUT PSALARY       DEC(9,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
RESULT SETS 0
MODIFIES SQL DATA
FENCED
NO DBINFO
WLM ENVIRONMENT DB9AWLM
STAY RESIDENT NO
COLLID DEVL7083
PROGRAM TYPE MAIN
RUN OPTIONS 'NOTEST(NONE,*,*,*)'
COMMIT ON RETURN NO
LANGUAGE SQL
BEGIN
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
SELECT
       FIRSTNME
     , MIDINIT
     , LASTNAME
     , WORKDEPT
    , HIREDATE
     , SALARY
INTO   PFIRSTNME
     , PMIDINIT
     , PLASTNAME
     , PWORKDEPT
     , PHIREDATE
     , PSALARY
FROM   EMP
```

```
WHERE  EMPNO = PEMPNO
;
 SET PSQLCODE   = SQLCODE ;
 SET PSQLSTATE  = SQLSTATE;
 SET PSQLERRMC  = 'ADIOS';

END
```

Using the editor

You can create a default stored procedure in IBM Data Studio and replace the generated stored procedure source code with the source code from another stored procedure. To do this, do the following:

- Create a default external SQL stored procedure named DEVL7083.EMPDTLSS. Follow the steps in 27.5.1, “Creating a new native stored procedure using the wizard” on page 689 and 27.5.2, “Creating an external SQL stored procedure from the wizard” on page 694 for detailed steps on how to create a new SQL stored procedure. *Do not check* the Deploy on Finish checkbox in the wizard.
- After the wizard is dismissed, the newly created stored procedure source is displayed in the Routine Editor.
- Select the entire source (Ctrl-A), and press Delete on your keyboard. The Routine Editor is now blank.
- Open EMPDTLSS.DDL using a workstation editor. Make the following modifications:
  - Add the FENCED keyword after NO DBINFO.
  - Change the WLM Environment to DB9AWLM.
  - Save the file.
  - Select all (Ctrl-A) and then copy (Ctrl-C).
- Paste (Ctrl-V) the modified source code into the open editor.
- Click **Save** or Ctrl-S.

The routine editor will now contain the new SQL stored procedure source as shown in Figure 28-8. Click the **Configuration** tab to see the modified WLM and build utility.

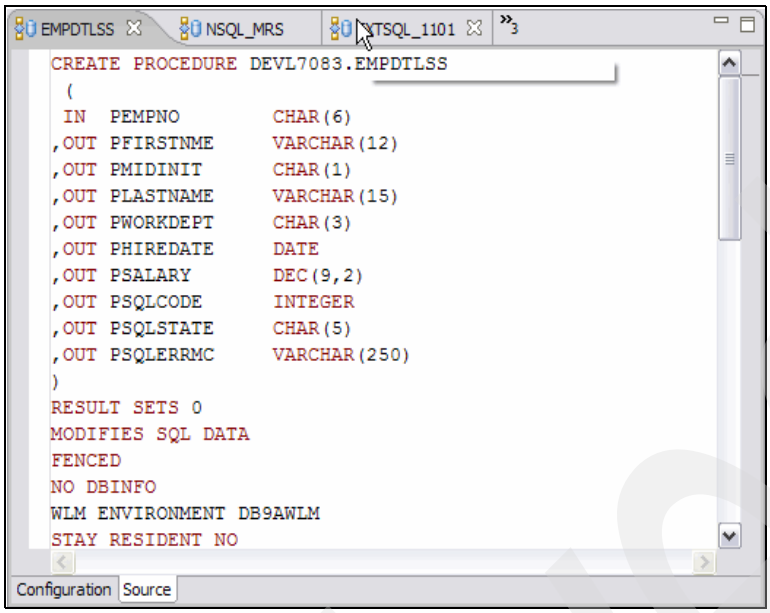

Figure 28-8 Create the procedure EMPDTLSS using the editor

Whether you use the Import wizard, or the Routine Editor option, the build process returns the same results when we have successfully created the SQL stored procedure for debugging. In addition to the build process, the WLM ENVIRONMENT, DB9AWLM where EMPDTLSS executes, is automatically refreshed to pick up our latest changes.

Using the Unified Debugger

In here we go through the features and tasks if you use the Unified Debugger. An IBM Developerworks article that talks about the Unified Debugger as well as some other Problem Determination tips is at:

```
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0706scanlon/
```

When a debug session has been established, IBM Data Studio initiates a switch from the Data Perspective to the Debug Perspective.

The Debug Perspective

The Unified Debugger launches the Eclipse Debug Perspective, shown in Figure 28-9, when a stored procedure is being debugged. This perspective is the same graphical interface used when debugging an SQL stored procedure, a Java stored procedure, a Java application, or any other resource in Eclipse.

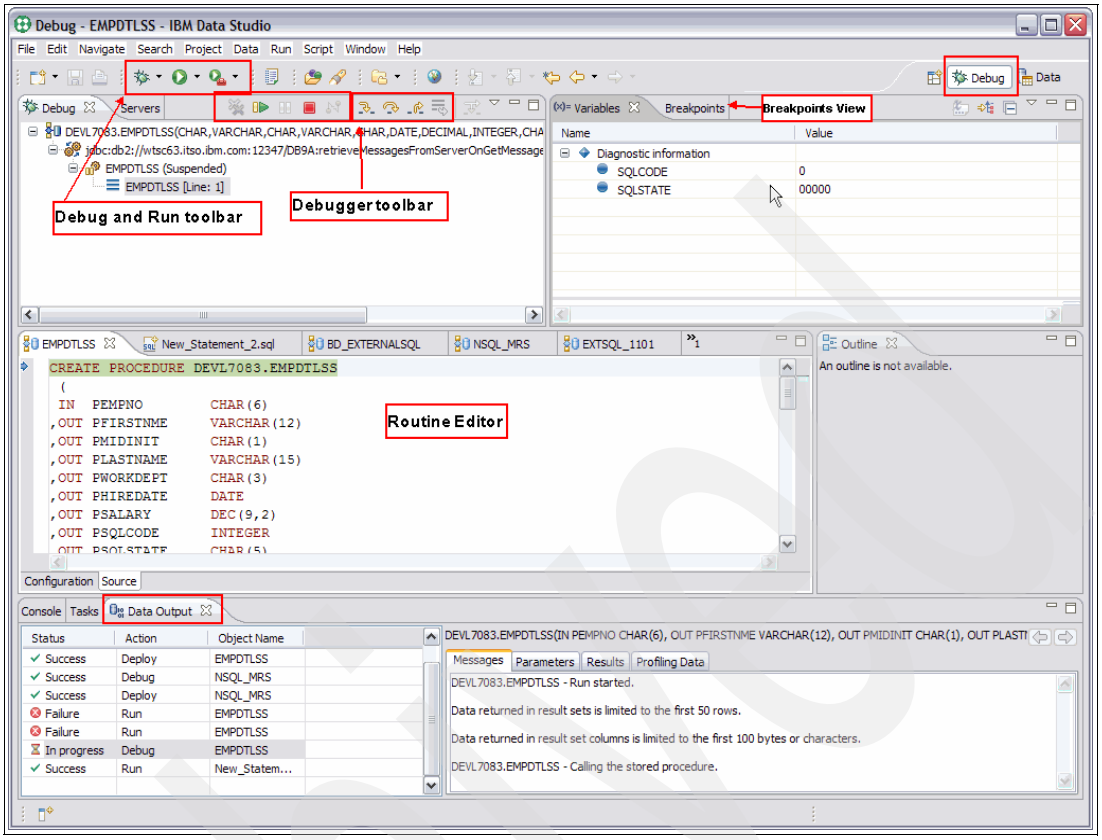

Figure 28-9 The Debug Perspective

The Debug Perspective is made up of the following related views and tool bars:

- Routine Editor View - Shows the SQL code.
- Breakpoints View - Shows the list of break points currently set.
- Variables View - Shows the list of defined variables.
- Outline View - Shows the variables and methods of the stored procedure under execution.
- Data Output - Shows the status history, execution messages, parameters, and result sets, if any.
- Execution toolbar - Provides icons to debug or simply execute a stored procedure; also keeps a list of most recently executed stored procedures.
- Debugger toolbar - Provides icons for the various debug execution step commands: step into, step over, step return, resume, pause, and terminate.

These views are connected in the sense that the break points and variables views show the debug data for the stored procedure currently shown in the Routine Editor View. Switching to a different procedure in the Routine Editor causes the break points and variables views to display the debug data for the newly selected stored procedure code. The Debug Perspective also includes a specialized set of toolbars for debugging.

- Routine Editor view

  The routine editor displays the stored procedures being debugged. Each tab in the Editor view displays an open resource. You can set breakpoints in this view, either during debug or before initiating the debug in the Data Perspective.

- Execution toolbar

The Execution toolbar includes three actions, as shown in Figure 28-10.

- Debug - Executes a stored procedure in Debug mode. The pull-down list next to the icon shows all previously debugged stored procedures. The default is to debug the last executed stored procedure.
- Run - Executes a stored procedure. The pull-down list next to the icon shows all previously executed stored procedures. The default is to execute the last executed stored procedure.
- External tools - Executes an Ant script or another tool. IBM Data Studio does not use this action.

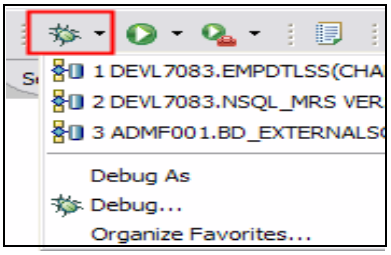

Figure 28-10 Debug and Run toolbar

- Debugger toolbar

  The Debugger toolbar includes the debug step commands illustrated in Table 28-3.

Table 28-3 Execution toolbar

Icon	Command description
	Step into next line or block of SQL code. If the current statement is a stored procedure call, then the next line is the first line of the called stored procedure.
	Step over to the next line of execution. If the current line is a call to a nested stored procedure or the next line is an indented block of code, then the nested procedure or block of code will be executed as one statement unless a break point was encountered.
	Step return causes execution to resume at the next line in the parent stored procedure of the current nested stored procedure unless a break point is encountered. If the current stored procedure is the only stored procedure in the call stack, then execution will run to completion or the next break point encountered.
	Resume causes the stored procedure being debugged to run and stop or break at the next breakpoint.
	Pause causes the execution of the stored procedure to be suspended. Click **Resume** or **Terminate** to continue or terminate execution.
	Terminate causes the execution of the stored procedure to stop. This does not cause the stored procedure to run to completion. This simulates an abort.

- Variables View

  The Debug Perspective's Variables view displays the current values of the defined variables in the stored procedures. When debugging a Java stored procedure, the Variables view will also list inherited variables.

- Outline View

  The Outline view in the Debug Perspective is the same as in the Data Perspective. It shows on a higher level where in the stored procedure code, execution is stopped. When debugging a Java stored procedure, this view shows the method where the stored procedure is stopped.

- Breakpoints View

  The Debug Perspective's breakpoints view and its associated toolbar allow you to manage the breakpoints you've set. When you set a breakpoint in the Routine Editor, an entry is added in this view, with the Resource name and line number. A checkbox next to this entry indicates that the breakpoint is active**.** To disable this breakpoint, but not remove it, uncheck the entry. (See Figure 28-11.

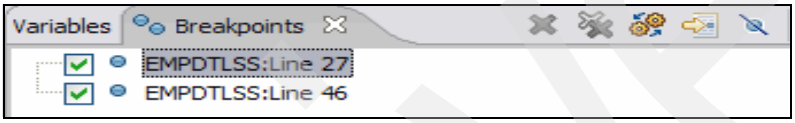

Figure 28-11 Breakpoints view

The Breakpoints view also has a specialized toolbar for managing the breakpoints, as illustrated in Table 28-4.

Table 28-4 Breakpoints view toolbar

Icon	Command description
	Remove a breakpoint. Click on an entry or several entries and click on this icon to remove the breakpoint
	Remove all breakpoints.
	Skip all breakpoints. This causes the execution to complete without stopping.
	Show all breakpoints.

Valid Unified Debugger breakpoint statements

The Unified Debugger highlights certain statements during a debug session. The highlighted statements are the only statements that you can step into or put breakpoints on. Some SQL

statements change variables, while other statements do nothing. This is summarized in Table 28-5.

Table 28-5 Valid SQL Debugger breakpoint and change variable statements

Category of statements	Statements

Statements that accept breakpoints (highlighted statements)	All SQL statements
	The following SQLPL statements:
	BEGIN
	BEGIN NOT ATOMIC
	BEGIN ATOMIC
	CLOSE CURSOR
	DECLARE var without default
	DECLARE RESULT_SET_LOCATOR [VARYING]
	DECLARE SQLSTATE
	DECLARE SQLCODE (unless there is a default)
	DO (inside a FOR)
	END
	FOR .. END
	FOR *select statement...*
	GOTO(LABEL)
	IF (EXPRESSION)
	ITERATE
	LEAVE
	OPEN CURSOR
	RESIGNAL
	SIGNAL
	RETURN(value)
	SELECT <..> INTO
	SET (EXPRESSION)
	UNTIL (EXPRESSION)
	WHEN (VALUE)
	WHILE (EXPRESSION)

Statements that change variables	CALL FETCH <..> INTO GET DIAGNOSTICS SELECT <..> INTO SET

Statements that do not accept breakpoints and do not impact the Unified Debugger processing	DECLARE cursor WITH RETURN FOR <sql statement>
	DECLARE CONDITION (CONDITION) FOR SQLSTATE (VALUE) "..."
	DECLARE CONTINUE HANDLER
	DECLARE CURSOR
	DECLARE EXIT HANDLER
	DECLARE UNDO HANDLER (unless they are entered)
	DO
	ELSE
	END CASE
	END IF
	END FOR
	END REPEAT
	END WHILE
	LOOP
	REPEAT (as a keyword alone)
	THEN
	labels, e.g. P1: :

28.3.5 Debugging the EMPDTLSS SQL case study

Launch the Unified Debugger

Launch the Unified Debugger using any of the following instructions:

1. **Data Project Explorer** → **Stored Procedures** → **DEVL7083.EMPDTLSS** and right-click **Debug.**
2. **Data Project Explorer** → **Stored Procedures** → **DEVL7083.EMPDTLSS** and right-click **Open.** In the Source tab, right-click on whitespace, select **Debug As** → **Debug.**
3. **Data Project Explorer** → **Stored Procedures** → **DEVL7083.EMPDTLSS** and right-click **Open.** In the Configuration tab, click the **Debug** icon.

Figure 28-12 shows the debug configuration dialog when using method 1.

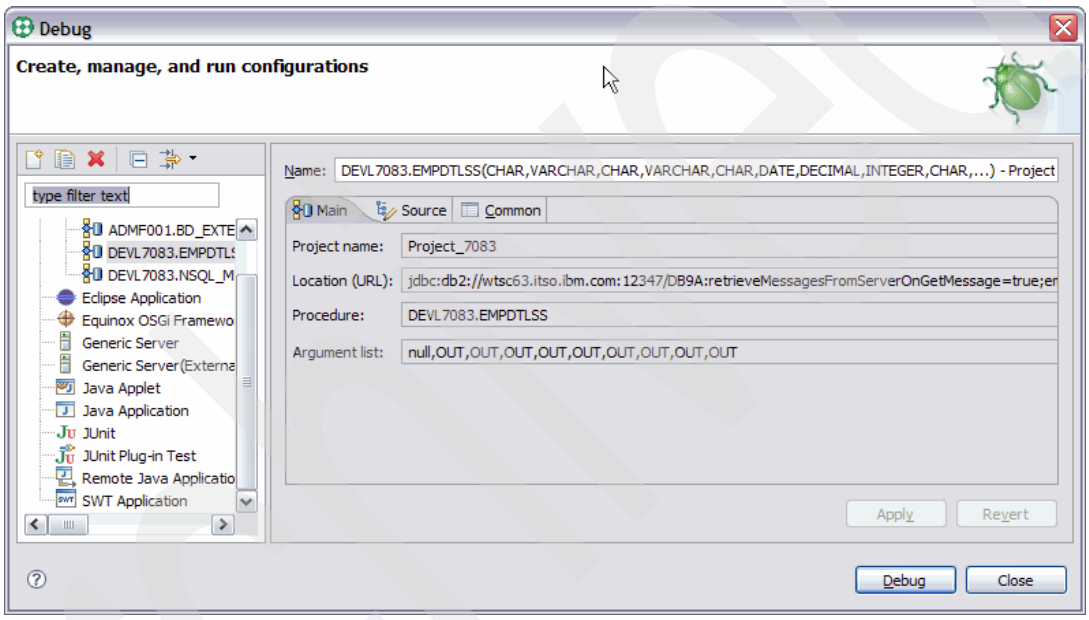

Figure 28-12 Start debugging for EMPDTLSS

After clicking **Debug**, the Specify Parameter Values dialog is displayed. This is because our stored procedure has one input parameter. See Figure 28-13.

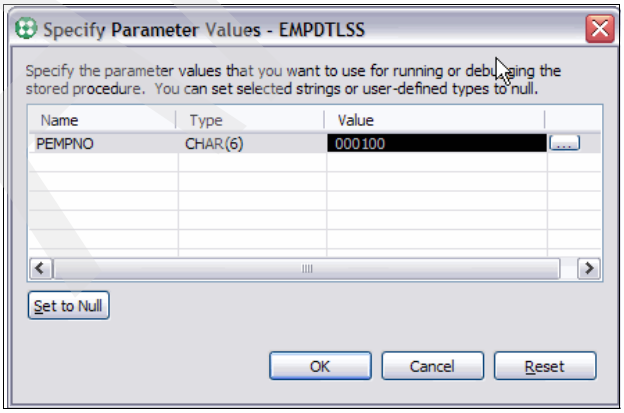

Figure 28-13 Specify Parameter Values

Type 000100 in the Value input area, then click **OK**.

If this is the first time you are debugging, IBM Data Studio shows a dialog to confirm switching to the Debug Perspective. Confirm by clicking **Yes**. See Figure 28-14.

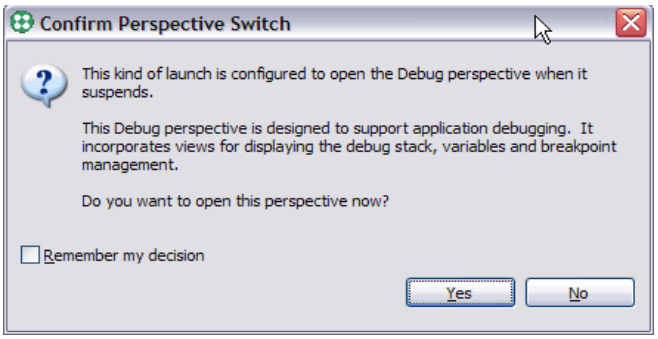

Figure 28-14 Confirm switch to Debug Perspective

The Debug Perspective shown in Figure 28-9 on page 754 is displayed. In the Routine Editor, the first line, `CREATE PROCEDURE...` is highlighted in blue.

Set breakpoints

Next, we set a breakpoint in two places.

1. Locate the `DECLARE SQLSTATE CHAR(5)` statement.
2. Place the cursor in the left-hand prefix.
3. Double-click to set the breakpoint. An entry for this line is added in the Breakpoints View.
4. Locate the `SET PSQLERRMC = 'ADIOS'` statement.
5. Set a breakpoint on this line.

To run to this breakpoint, we select the **Resume** icon from the toolbar.

From this point, we step through the remaining lines of code using the Step Into icon and view the variables in the output window at the bottom as we progress through the code. At the end of the stored procedure, all values as they have been processed by our code appear in the Editor. See Figure 28-15.

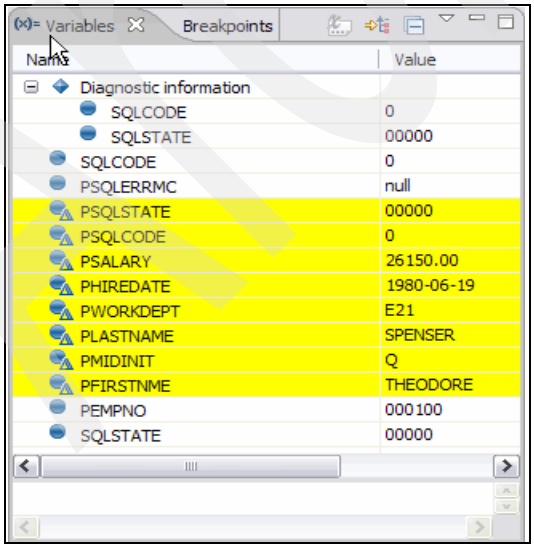

Figure 28-15 Unified Debugger Variables display

Furthermore, you can view the output parameters in the Data Output view → **Parameters** tab, as shown in Figure 28-16.

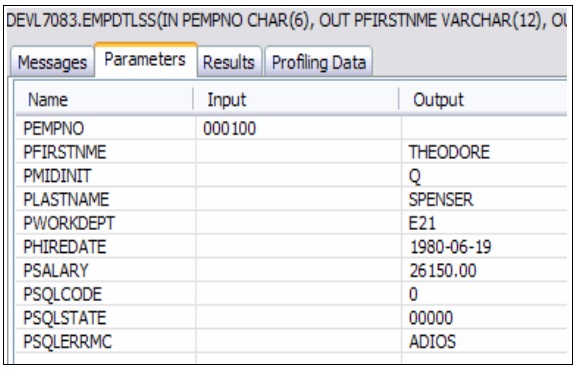

Figure 28-16 Unified Debugger Data Output view's Parameters tab

For more information on the Unified Debugger, see the Information Center Help included with IBM Data Studio.

28.4 Debugging COBOL, PL/I, and C/C++ procedures on z/OS

Our COBOL debugging example for this section used the COBOL stored procedure, DEVL7083.EMPDTLSC, created in Example 10-1 on page 115. Similar setup steps to those described in this section can be used for debugging PL/I and C/C++ stored procedures.

Here we discuss the following using the IBM Debug Tool with Rational Developer v7 for System Z (RDz v7):

1. Overview of debugging COBOL procedures with the IBM Debug Tool
2. Prerequisites and setup on the:
   - Workstation
   - z/OS
3. Create the COBOL stored procedure source file
4. Modify the COBOL stored procedure source
5. Register the COBOL stored procedure
6. Prepare the stored procedure for debug
7. Set up a WLM AE with required data sets
8. Debug using RDz V7
9. Debug using TSO (3270 interface only)

28.4.1 Overview of debugging COBOL procedures with the IBM Debug Tool

The chart in Figure 28-17 describes the processing flow for debugging COBOL stored procedures using RDz V7 and the IBM Debug Tool.

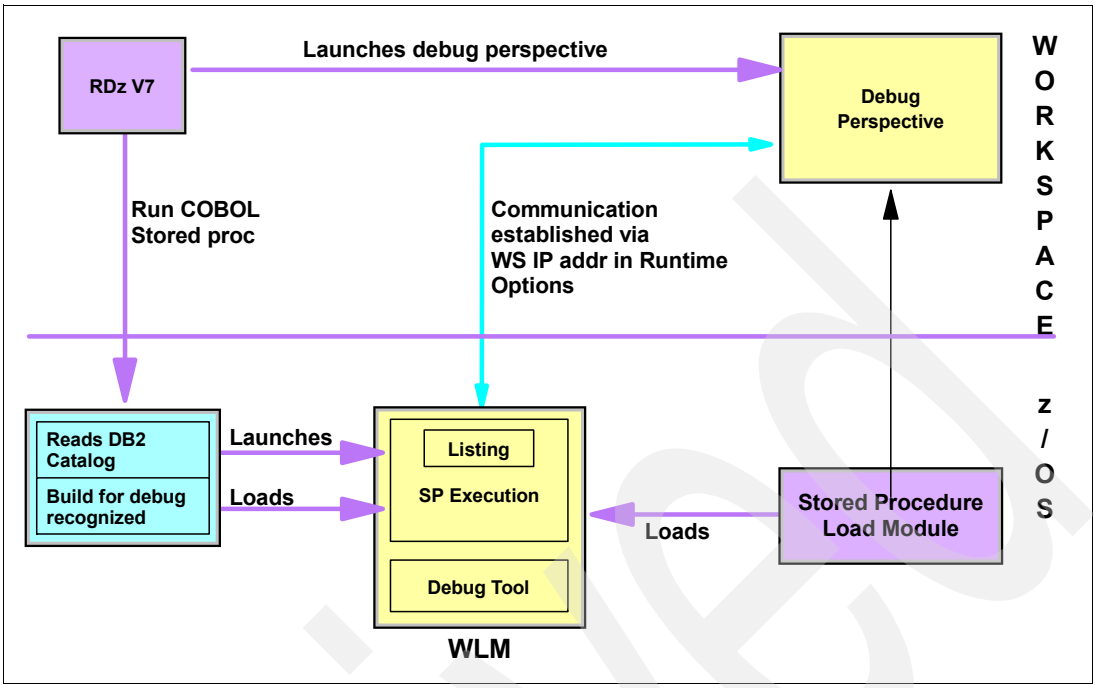

Figure 28-17 Processing overview - RDz and Debug Tool

The chart shows how RDz v7 and the Debug Tool (DT) interact to launch a debug session accessing the language listing file.

1. RDz v7 is used to CALL the stored procedure in debug mode by clicking the **Run** action against the COBOL stored procedure.
2. During initialization of the stored procedure into the SPAS, Language Environment (LE) recognizes the TEST run option and loads Debug Tool (DT) into the stored procedure address space. Control is passed from LE to DT. DT then parses the remaining portion of the RUN OPTIONS string, locating the workstation IP address. Communication is established between the Workstation and the stored procedure address space, and debugging is started using the stored procedure listing file.
3. Once communication is established, RDz launches the Debug Perspective and displays the listing.

28.4.2 Prerequisites and setup

Here we describe the prerequisites for the workstation and z/OS environments.

Workstation

No client tool is required to debug COBOL, PL/I, or C/C++ stored procedures on z/OS when using the IBM Debug Tool on z/OS, which can use a 3270 TSO interface for the debugging. See 16.5, “IBM Debug Tool” on page 333 for more information on Debug Tool, and 16.5.2, “IBM Debug Tool on z/OS: An example” on page 336.

For our testing, we used RDz V7, which is a product that is installed on top of the Rational Application Developer V7 product. This made the debugging easy for an application developer used to workstation Debuggers. Furthermore, since both the IBM Data Studio and RDz v7 are Eclipse-based, the RDz Debug Perspective is exactly the same as the Debug Perspective in IBM Data Studio. To debug COBOL, PL/I, or C/C++ stored procedures, we will need to install:

- Rational Application Developer V7 Fix Pack 4 (optional - fee based)
- Rational Developer V7 for System z (FMID HHOP710)

Like IBM Data Studio, RDz v7 ships the license jar files and does not require DB2 Connect. The location of the license jars is the same in both products.

z/OS

The products needed on the host are the LE runtime, the Debug Tool (DT) for z/OS library, and the RDz V7 Host components.

- LE runtime
  - hlq.SCEERUN
- Debug Tool (DT) library
  - hlq.SEQAMOD

  Once Debug Tool is installed on z/OS, you must make certain Debug Tool load modules are available in an APF-authorized data set that is in the system link list concatenation. The hlq.SEQAMOD data set must be in the load module search path whenever you debug a program with Debug Tool. The *Customization Guide* in the Debug Tool for z/OS Web site below gives more information on how to configure the Debug Tool:

  ```
  http://www-306.ibm.com/software/awdtools/debugtool/
  ```

  We discuss how to use the IBM Debug Tool in our case study at 16.5, “IBM Debug Tool” on page 333.
- Host configuration for RDz v7

  RDz v7 requires a connection to the TSO command service on z/OS. The RDz component that provides the core services for client-host communication is the Remote Systems Explorer (RSE).

  RDz v7 has a list of prerequisite software that must be installed and operational before the product will work. Below is a summary of these requirements:

  - The Software Configuration and Library Manager (SCLM) Developer toolkit (FMID HSD3310).
  - The C/C++ DLL class library CBC.SCLBDLL and the Language Environment (LE) runtime libraries CEE.SCEERUN and CEE.SCEERUN2 must be in LINKLIST.
  - INETD for setting up the client-host connection.
  - TCP/IP and Resolver configuration files must be set up.

  For additional information on configuring the z/OS server for RDz V7, see *The Rational Developer for System z Host Configuration Guide*, SC23-7658.
- WLM AE setup for debugging

  Since Debug Tool is loaded in the WLM environment when the COBOL, PL/I, C/C++ language stored procedure has been compiled with the TEST parm and has been executed, you may want to set up a separate WLM environment for debugging these stored procedures.

  The WLM procedure where the DB2 COBOL stored procedure executes needs to have the following data sets added to STEPLIB, if they are not already in LINKLST:

  ```
  LE data set; hlq.SCEERUN
  DT data set; hlq.SEQAMOD
  Stored Procedure Load Library
  ```

  Example 28-9 is a sample WLM procedure for running DB2 Cobol stored procedures.

Example 28-9 WLM AE procedure for running DB2 COBOL stored procedures

```
//*************************************************************
//*    JCL FOR RUNNING THE WLM-ESTABLISHED STORED PROCEDURES
//*    ADDRESS SPACE
//*       RGN     -- THE MVS REGION SIZE FOR THE ADDRESS SPACE.
//*       DB2SSN  -- THE DB2 SUBSYSTEM NAME.
//*       NUMTCB  -- THE NUMBER OF TCBS USED TO PROCESS
//*                  END USER REQUESTS.
//*       APPLENV -- THE MVS WLM APPLICATION ENVIRONMENT
//*                  SUPPORTED BY THIS JCL PROCEDURE.
//* THIS IS FOR USER SQL STORED PROCS - 2003.05.19 M.SCANLON
//*************************************************************
//DSN7WL4  PROC RGN=0K,APPLENV=DSN7WL4,DB2SSN=DSN7,NUMTCB=15
//IEFPROC  EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//         PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD DISP=SHR,DSN=MEL.DSN7.RUNLIB.LOAD
//         DD DISP=SHR,DSN=DSN.DSN7.RUNLIB.LOAD
//         DD DISP=SHR,DSN=DSN.DSN7.SDSNEXIT
//         DD DISP=SHR,DSN=DSN.DSN7.SDSNLOAD
//         DD DISP=SHR,DSN=SYS1.SCEERUN
//SYSTSPRT DD SYSOUT=H
//CEEDUMP  DD SYSOUT=*
//SYSPRINT DD SYSOUT=H
//SYSABEND DD DUMMY
//SQLDUMMY DD DUMMY
```

RDz uses a REXX stored procedure, ELAXMREX, located in hlq.SFEKPROC. You will need to define the WLM environment for this stored procedure. Example 28-10 shows a sample WLM procedure for ELAXMREX. Because this is a REXX stored procedure, the NUMTCB is always set to 1. The workfiles at the bottom of the procedure are needed by the compiler.

Example 28-10 WLM AE sample procedure for ELAXMREX

```
//DSN9WLMB PROC RGN=0K,APPLENV=DSN9WLMB,DB2SSN=DSN9,NUMTCB=1
//IEFPROC  EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,DYNAMNBR=10,
//         PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD DISP=SHR,DSN=SYS1.SCEERUN
//         DD DISP=SHR,DSN=SYS1.SIGYCOMP
//         DD DISP=SHR,DSN=DSN.DSN9.SDSNEXIT
//         DD DISP=SHR,DSN=FOWLERM.ERRFDBK8.FIDUCIA1.LOAD
//*        DD DISP=SHR,DSN=COBTSTA.V5R1M1.SCCULOAD
//         DD DISP=SHR,DSN=ENPLI340.SIBMZCMP
//         DD DISP=SHR,DSN=DSN.DSN9.SDSNLOAD
//         DD DISP=SHR,DSN=MEL.DSN9.RUNLIB.LOAD
//*YSEXEC  DD DISP=SHR,DSN=MEL.DSN9.SDSNCLST USE SAME ELAXMREX AS DSN7
//SYSEXEC  DD DISP=SHR,DSN=MEL.DSN7.SDSNCLST
//SYSTSPRT DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSABEND DD DUMMY
//*********************************************************************
//* WORKFILES FOR COMPILERS AND BINDER
//*********************************************************************
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
```

```
//SYSUT3   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
```

28.4.3 Create the COBOL stored procedure source file

Once the z/OS server and RDz v7 are set up, the COBOL stored procedure can be created for debug. In our examples here, we connect to a server, DSN9, that has been configured for RDz.

RDz v7 is Eclipse-based and uses much of the same infrastructure as IBM Data Studio. As a result, you will notice that the Data Perspective, the views, and the wizards are similar. Where the user interface and procedural steps are the same, we will refer you to the IBM Data Studio chapter.

We can create our COBOL stored procedure on the z/OS side by using TSO to create and edit our COBOL stored procedure source.

On the client side, we can use RDz v7. Here, we show how one creates a template[2] COBOL stored procedure, edits it, and then deploys it to the z/OS server.

Create a database connection

- Click **Start** → **All Programs** → **IBM Software Development Platform** → **Rational Developer for System z** → **Rational Developer for System z.**
- In the Workspace Launcher dialog, append /RDzTest to the default workspace directory specified. See Figure 28-18.

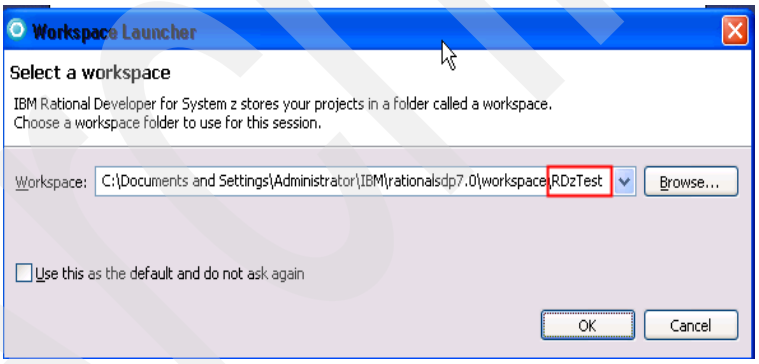

Figure 28-18 RDz V7 Workspace launcher

- RDz v7 launches with the z/OS Projects Perspective. Click **Window** → **Open Per***spective* → *Data* to switch to the Data Perspective.
- Click the **New Connection** icon, and create a new connection to DSN9. Follow the steps in 27.4.2, “Creating a connection” on page 682. For our testing, we used the following information:

  ```
  Location: DSN9
  Host: stplex4a.svl.ibm.com
  Port: 8016
  ```

  Remember to uncheck the checkbox “Retrieve objects created by this user only.” The completed Connection Wizard is shown in Figure 28-19.

[2] RDz creates a template only. You you will need to edit the stored procedure after leaving the New Stored Procedure wizard

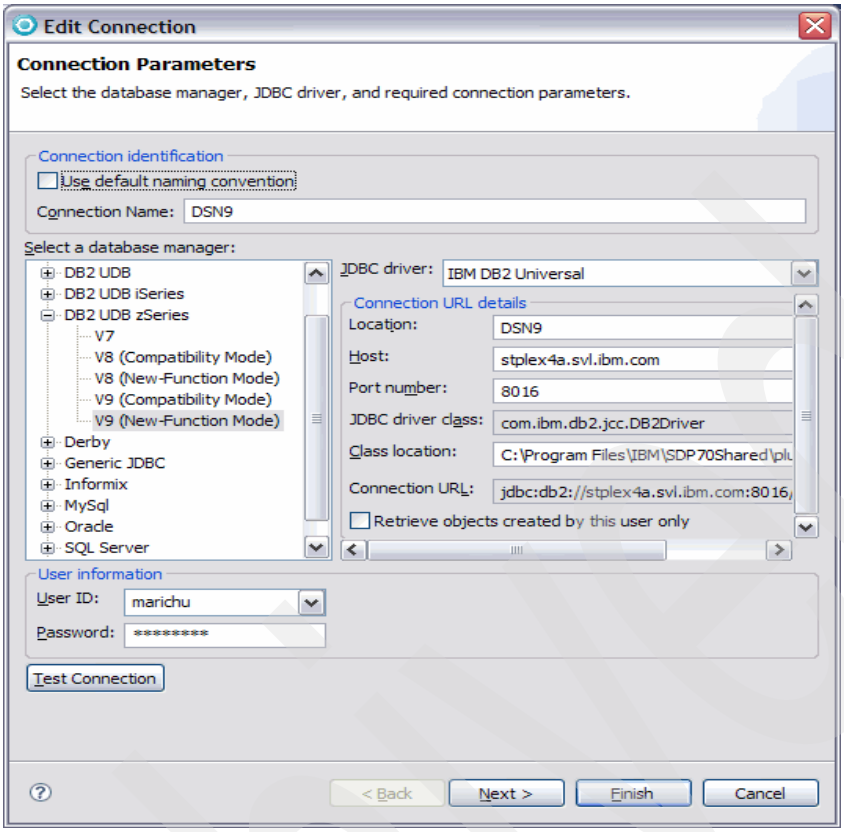

Figure 28-19 RDz - New Database connection

Create a Remote System connection

In RDz, the Remote Systems view allows you to access and manage your non-DB2 objects, such as the Partitioned Data Sets, in the DSN9B server.

- In the Remote Systems view, right-click **z/OS Objects → New Connection. See Figure 28-20.**
- Follow the wizard to enter the information for your z/OS system. This information should be provided to you by your system programmer. At the end of the wizard, you should see a connection created to your z/OS system.

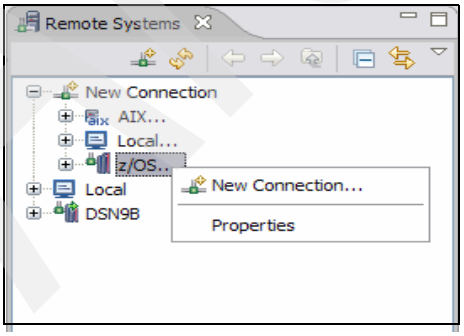

Figure 28-20 RDz - new remote connection

Create a Database Development Project

- Click **File → New → Data Development Project.**
- Type `CobolProject2` for the project name. Click **Next.**

- Select connection **DSN9B** for the Target connection. Click **Finish.**

Create a new stored procedure using the New Stored Procedure wizard

Note: See 27.5.1, “Creating a new native stored procedure using the wizard” on page 689, for details on each of the pages of the New Stored Procedure wizard.

- In the Data Project Explorer, right-click the **Stored Procedures** folder → **New** → **Stored Procedure.**
- The New Stored Procedure wizard is launched. The default project is CobolProject2. Click **Next.**
- In the Name and Language page, type `EMPDTLC` for the name. This is the name of both the stored procedure and the member name in the data set that you will specify in the next page. Leave the default setting for Language as Cobol. See Figure 28-21. Click **Next.**

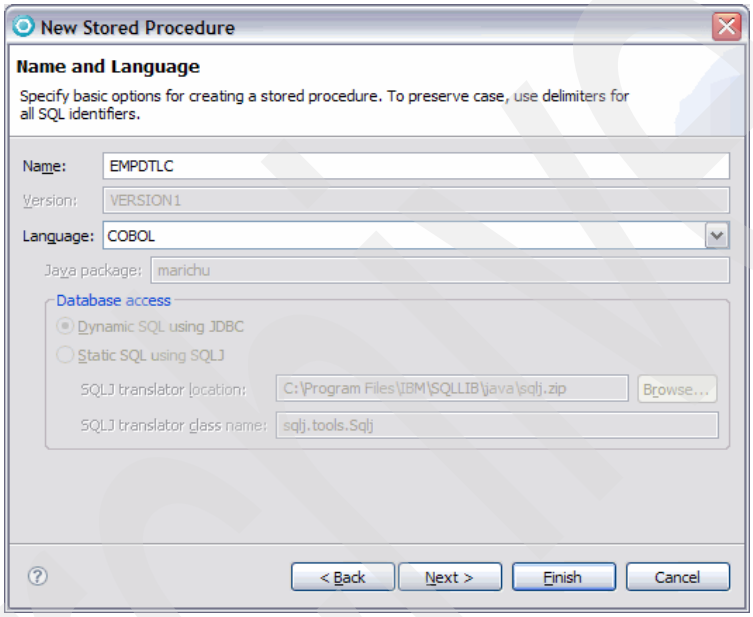

Figure 28-21 RDz - New stored procedure, Name and Language

- In “Select target and name”, we specify the member name in the z/OS PDS for this stored procedure. This should be the same as what was specified in the previous page. In RDz, this name is limited to 7 characters. Type `EMPDTLC` for the name. See Figure 28-22. Click **Next.**

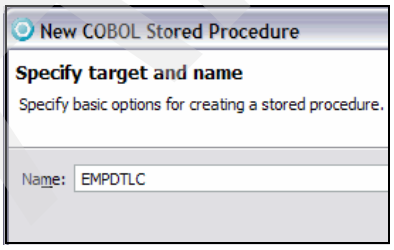

Figure 28-22 RDz - target name for stored procedure PDS member names

- In the Source Location page, RDz recognizes that you are connected to the Remote System, DSN9, and sets the Connected System to this value and the z/OS Filter String to the default high level qualifier in this system, which is the login userid. See Figure 28-23.

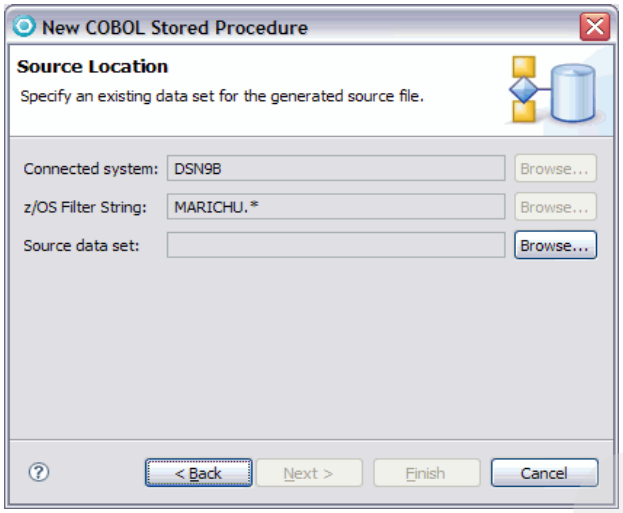

Figure 28-23 RDz - Source Location

- Click **Browse** to specify the Source data set. RDz will display a list of Select Data Sets. Select **hlg.SOURCE.COBOL**. This is the Partitioned Data Set (PDS) in your z/OS system that will contain the source code of the COBOL stored procedure. See Figure 28-24. Click **Next.**

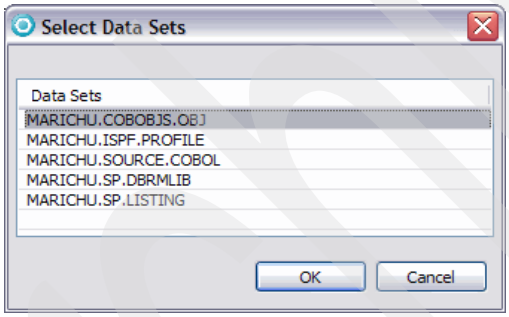

Figure 28-24 RDz - Select data sets for Source

- In the SQL Statements page, click **Create SQL.** This launches the SQL wizard.
  - We will use defaults in the Specify SQL Statement Information page. Click **Next.**
  - In the Tables tab, select **DSN8910** → **EMP**. See Figure 28-25 on page 768. Click >.

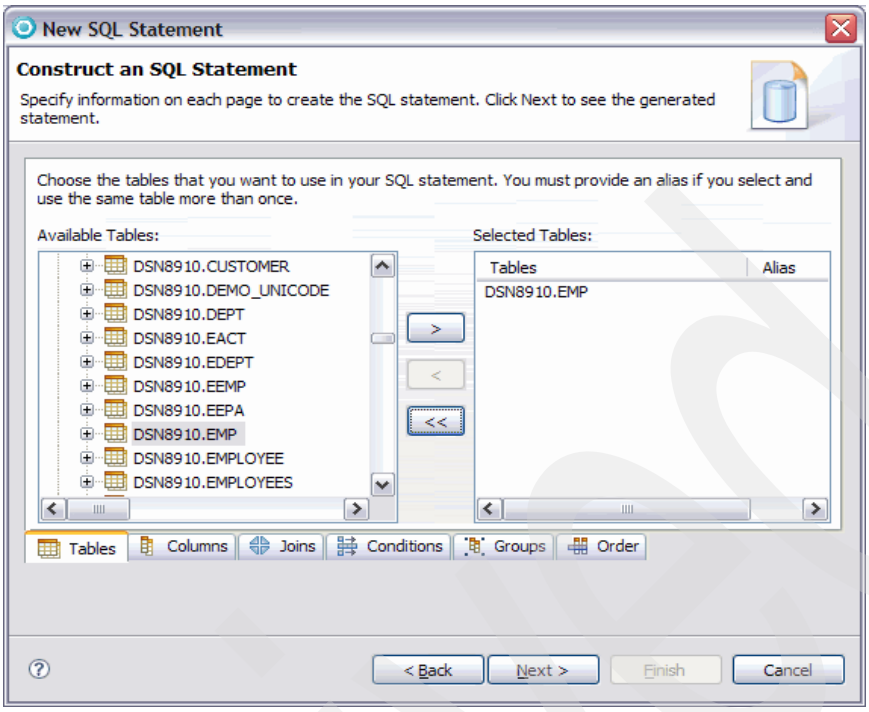

Figure 28-25 RDz - SQL wizard's Tables tab

– In the Columns tab, Ctrl-click `FIRSTNME`, `MIDINIT`, `LASTNAME`, `WORKDEPT`, `HIREDATE` and `SALARY`. Click >. See Figure 28-26.

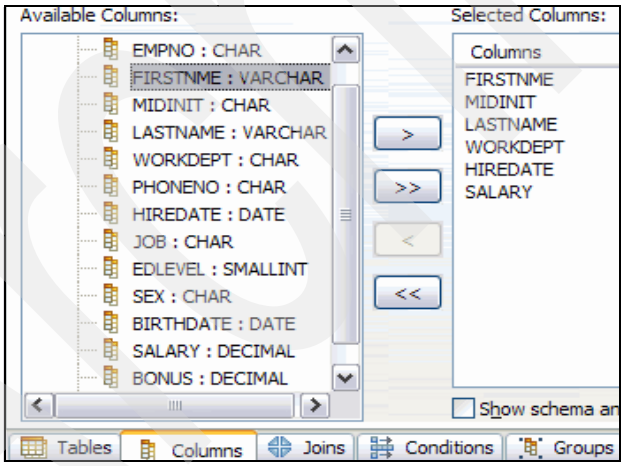

Figure 28-26 RDz - SQL wizard, Columns tab, selecting result columns

– In the Conditions tab, click the first cell Columns column, and select **EMP.EMPNO**. See Figure 28-27. Press Enter.

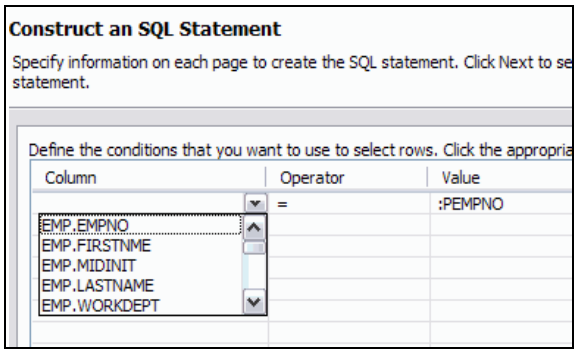

Figure 28-27 RDz - SQL wizard, Conditions tab

- The Operator cell contains the default =. Go to the Value cell and type `:PEMPNO`. Press Enter. Click **Next.**
- The Statement page shows the generated SQL Statement. Click **Finish.**

For more details on the SQL wizard and other SQL statement building tools within RDz, see 27.4.4, “Creating SQL statements to use in your stored procedure” on page 685.

- Examine the generated SQL statement in the Statement details text window. We will use the default Result Set setting. Click **Next.**
- In the Parameters page, click **Add**. The Add Parameter dialog is launched. Create the parameters as shown in Table 28-6. In Chapter 27, “The IBM Data Studio” on page 643, see“Parameters” on page 697 for details on using this dialog.

Table 28-6 Parameter names and values for EMPDTLC COBOL stored procedure

Parameter Mode	Name	SQL Type
OUT	PFIRSTNME	VARCHAR(12)
OUT	PMIDINIT	CHAR(1)
OUT	PLASTNAME	VARCHAR(15)
OUT	PWORKDEPT	CHAR(3)
OUT	PHIREDATE	DATE
OUT	PSALARY	DEC(9,2)
OUT	PSQLCODE	INTEGER
OUT	PSQLSTATE	CHAR(5)
OUT	PSQLERRMC	VARCHAR(250)

Although not visible in the list, RDz has generated PEMPNO as an INPUT parameter. The completed Parameters page is shown in <RDz_Parameters.gif>. Click **Next.**

- In the Deploy Options page, type `SG247083` for the Collection ID. Click **Advanced.**
- In the z/OS Options dialog, select the **Stored Procedure Options** tab and check the box to Build Stored Procedure for Debugging. Note that the TEST options are added. This field contains the TCP/IP address of the client. If the IP address of the client is changed, you will need to reset this by unchecking this checkbox, saving the stored procedure, and resetting the checkbox.
- Still in this tab, type `DSN9WL4` for the WLM Environment. Click the **Deploy Options** tab.

- In the z/OS Options dialog, select the Deploy Options tab, click the ellipsis next to the grayed-out text area PACKAGE(SG247083). The Bind Options dialog is displayed.
- Type `ISOLATION(CS)` in the text area. Click **OK**. Click **OK** again to return to the wizard.
- Click **Next.** We will not add any additional code to this stored procedure, so we skip the Code Fragments page. See IBM DS for an example on how to use this feature. Click **Next.**
- The last page of the wizard is the Summary page. We examine our inputs. Click **Finish** to complete creation of our stored procedure.

Figure 28-28 shows the generated COBOL stored procedure source code.

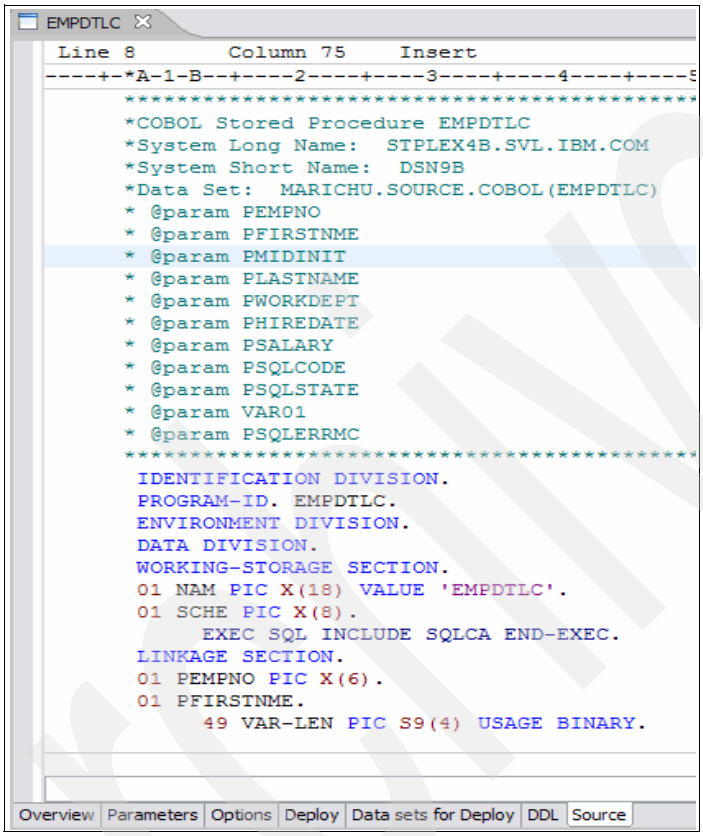

Figure 28-28 COBOL stored procedure source listing in the Editor View

Modify the stored procedure

While the generated COBOL stored procedure is deployable, we need to edit the source code to handle the result set from the query. The changes we made to the generated COBOL source code are summarized in Table 28-7.

Table 28-7 Summary of changes to the generated COBOL stored procedure

Location in generated template	Changes
Working Storage	Add work variable `WS-PEMPNO` for storing the input parameter `PEMPNO`,
Working Storage	Add a copylib, the DCLGEN file for table EMP.
SELECT PEMPNO...	Convert `SELECT PEMPNO...` into `SELECT PEMPNO..INTO` working storage variables; Change WHERE clause to `PEMPNO = :WS-PEMPNO`

Location in generated template	Changes
Add Error Handling	Add Evaluate statement to check SQLCODE. If non-zero, process the error in 9000_DBERROR. Add procedure 9000_DBERROR.

We did additional modifications that were cosmetic in nature. After our changes, the COBOL stored procedure is shown in Example 28-11.

Example 28-11 Modified Cobol stored procedure source

```
000100*---------------------------------------------------------------00010000
000200*    EMPDTL1C - SAMPLE COBOL STORED PROCEDURE                    00020002
000300*               USES PARAMETER STYLE GENERAL                     00030007
000400*                                                                00040004
000500*    MODULE NAME = EMPDTL1C                                      00050002
000600*                                                                00060000
000700*    FUNCTION = THIS MODULE ACCEPTS AN EMPLOYEE NUMBER AND RETURNS00070002
000800*               EMPLOYEE INFORMATION                             00080002
000900*                                                                00090000
001000*---------------------------------------------------------------00100000
001100 IDENTIFICATION DIVISION.                                        00110000
001200*-----------------------                                        00120000
001300 PROGRAM-ID.    EMPDTLC.                                        00130002
001400/                                                               00140000
001500 ENVIRONMENT DIVISION.                                           00150000
001600*--------------------                                           00160000
001700 CONFIGURATION SECTION.                                          00170000
001800 INPUT-OUTPUT SECTION.                                           00180000
001900 FILE-CONTROL.                                                   00190000
002000 DATA DIVISION.                                                  00200000
002100*-------------                                                  00210000
002200 FILE SECTION.                                                   00220000
002300/                                                               00230000
002400 WORKING-STORAGE SECTION.                                        00240000
002500*---------------------------------------------------------------00250008
002600* WORKAREAS                                                     00260008
002700*---------------------------------------------------------------00270008
002800 01  WS-PARMAREA.                                               00280002
002900         02  WS-EMPNO              PIC X(06).                   00290002
003000                                                                00300002
003100*---------------------------------------------------------------00310003
003200* VARIABLES FOR ERROR-HANDLING                                  00320000
003300*---------------------------------------------------------------00330003
003400 01  ERROR-MESSAGE.                                             00340000
003500         02  ERROR-LEN   PIC S9(4)  COMP VALUE +960.            00350000
003600         02  ERROR-TEXT  PIC X(80) OCCURS 12 TIMES              00360000
003700                                    INDEXED BY ERROR-INDEX.     00370000
003800 77  ERROR-TEXT-LEN      PIC S9(9)  COMP VALUE +80.             00380000
003900                                                                00390000
004000/                                                              00400000
004100*---------------------------------------------------------------00410003
004200* SQLCA AND DCLGENS FOR TABLES                                  00420000
004300*---------------------------------------------------------------00430003
004400          EXEC SQL INCLUDE SQLCA  END-EXEC.                     00440000
004500                                                                00450000
004600     EXEC SQL INCLUDE EMP                                       00460002
004700     END-EXEC.                                                  00470002
004800                                                                00480002
```

```
004900/                                                                 00490000
005000 LINKAGE SECTION.                                                 00500002
005100 01  PEMPNO        PIC X(6).                                      00510002
005200 01  PFIRSTNME.                                                   00520002
005300     49  PFIRSTNME-LEN     PIC S9(4) COMP.                        00530002
005400     49  PFIRSTNME-TEXT    PIC X(12).                             00540002
005500 01  PMIDINIT      PIC X(1).                                      00550002
005600 01  PLASTNAME.                                                   00560002
005700     49  PLASTNAME-LEN     PIC S9(4) COMP.                        00570002
005800     49  PLASTNAME-TEXT    PIC X(15).                             00580002
005900 01  PWORKDEPT     PIC X(3).                                      00590002
006000 01  PHIREDATE     PIC X(10).                                     00600002
006100 01  PSALARY       PIC S9(7)V9(2) COMP-3.                         00610002
006200 01  PSQLCODE      PIC S9(9) COMP.                                00620002
006300 01  PSQLSTATE     PIC X(5).                                      00630002
006400 01  PSQLERRMC.                                                   00640002
006500     49  PSQLERRMC-LEN     PIC S9(4) COMP.                        00650002
006600     49  PSQLERRMC-TEXT    PIC X(250).                            00660002
006700                                                                 00670002
006800*-------------------------------------------------------------------00680003
006900* SQL CURSORS AND STATEMENTS                                     00690000
007000*-------------------------------------------------------------------00700003
007100                                                                 00710000
007200/                                                                 00720000
007300 PROCEDURE DIVISION USING PEMPNO, PFIRSTNME, PMIDINIT, PLASTNAME, 00730002
007400                   PWORKDEPT, PHIREDATE, PSALARY, PSQLCODE,      00740002
007500                   PSQLSTATE, PSQLERRMC.                         00750002
007600                                                                 00760000
007700*-------------------------------------------------------------------00770003
007800* MAIN PROGRAM ROUTINE                                           00780000
007900*-------------------------------------------------------------------00790003
008000 MAINLINE.                                                        00800000
008100         DISPLAY '++ START OF EMPDTL1C STARTING ++'.              00810002
008200                                                                  00820000
008300         EXEC SQL                                                 00830002
008400              SET CURRENT SQLID = USER                            00840002
008500         END-EXEC.                                                00850002
008600                                                                  00860002
008700         PERFORM 2000-PROCESS                                     00870000
008800         THRU    2000-EXIT.                                       00880000
008900                                                                  00890000
009000         DISPLAY '++ END OF EMPDTL1C ++'.                         00900002
009100                                                                  00910000
009200         GOBACK.                                                  00920000
009300/                                                                 00930000
009400*-------------------------------------------------------------------00940003
009500* 2000-PROCESS                                                   00950000
009600*-------------------------------------------------------------------00960003
009700 2000-PROCESS.                                                    00970000
009800         MOVE PEMPNO TO WS-EMPNO.                                 00980002
009900                                                                  00990002
010000         DISPLAY 'WS-EMPNO = ' WS-EMPNO.                          01000002
010100                                                                  01010000
010200         EXEC SQL                                                 01020002
010300           SELECT                                                 01030002
010400               FIRSTNME,                                          01040002
010500               MIDINIT,                                           01050002
010600               LASTNAME,                                          01060002
010700               WORKDEPT,                                          01070002
010800               HIREDATE,                                          01080002
```

```
010900                SALARY                                            01090002
011000              INTO                                                01100002
011100                 :PFIRSTNME                                       01110002
011200                , :PMIDINIT                                       01120002
011300                , :PLASTNAME                                      01130002
011400                , :PWORKDEPT                                      01140002
011500                , :PHIREDATE                                      01150002
011600                , :PSALARY                                        01160002
011700              FROM EMP                                            01170002
011800              WHERE EMPNO = :WS-EMPNO                             01180002
011900           END-EXEC.                                              01190002
012000                                                                  01200002
012100           DISPLAY '++ SQLCODE AFTER SELECT = ' SQLCODE.          01210002
012200                                                                  01220002
012300           MOVE SQLCODE  TO PSQLCODE.                             01230002
012400           MOVE SQLSTATE TO PSQLSTATE.                            01240002
012500           MOVE SQLERRMC TO PSQLERRMC.                            01250002
012600                                                                  01260002
012700           EVALUATE SQLCODE                                       01270002
012800               WHEN 0                                             01280002
012900                    CONTINUE                                      01290002
013000               WHEN OTHER                                         01300002
013100                    PERFORM 9000-DBERROR                          01310002
013200                    THRU    9000-EXIT                             01320002
013300           END-EVALUATE.                                          01330002
013400                                                                  01340002
013500 2000-EXIT.                                                       01350002
013600           EXIT.                                                  01360000
013700/                                                                 01370000
013800*----------------------------------------------------------------01380003
013900* 9000-DBERROR - GET ERROR MESSAGE                               01390000
014000*----------------------------------------------------------------01400003
014100 9000-DBERROR.                                                    01410000
014200           CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.01420000
014300           IF RETURN-CODE = ZERO                                  01430000
014400              PERFORM 9999-ERROR-DISPLAY THRU                     01440000
014500                      9999-EXIT                                   01450000
014600              VARYING ERROR-INDEX                                 01460000
014700              FROM    1 BY 1                                      01470000
014800              UNTIL   ERROR-INDEX GREATER THAN 12.                01480000
014900                                                                  01490000
015000           GOBACK.                                                01500000
015100                                                                  01510000
015200 9000-EXIT.                                                       01520000
015300           EXIT.                                                  01530000
015400/                                                                 01540000
015500*----------------------------------------------------------------01550003
015600* 9999-ERROR-DISPLAY                                             01560000
015700*----------------------------------------------------------------01570003
015800 9999-ERROR-DISPLAY.                                              01580000
015900           DISPLAY ERROR-TEXT (ERROR-INDEX).                      01590000
016000 9999-EXIT.                                                       01600000
016100           EXIT.                                                  01610000
```

28.4.4 Prepare the stored procedure for debug

Using JCLs

We can use your existing JCLs to build or alter our COBOL stored procedures for debug. The following steps are taken to enable debugging with the Debug Tool:

- Update the step that executes DSNHICOB to include a PARM.COB step specifying TEST(ALL). In addition to a specification of ALL, NONE, or BLOCK are available. With Enterprise COBOL for z/OS, the option of SEPARATE is also available, which puts debug information into a separate file from the load module information. The data set with the load module is very near the size of a NOTEST specification.
- Define a COB.SYSPRINT DD to a specific partitioned data set (PDS) member (could be a sequential data set, though a PDS is recommended):

  The COB.SYSPRINT DD requires data set characteristics of RECFM=FBA and LRECL=133.

The JCL to compile the COBOL procedure is shown in Example 28-12.

Example 28-12 COBOL compile procedure example

```
//PH061S03 EXEC DSNHICOB,MEM=EMPDTLSC,
...
//  PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)',
//         TEST(ALL),MAP,OFFSET)
//COB.SYSPRINT DD DSN=MARICHU.SOURCE.COBOL(EMPDTLC),
//            DISP=SHR
```

Deploying with RDz

When using RDz, we can compile the new COBOL Stored Procedure in debug mode and register the CREATE PROCEDURE ddl with a few clicks. We assume that you have all the data sets necessary for building a COBOL application or stored procedure created in your z/OS server. You can investigate what data sets exist in your z/OS server by opening the folder under your z/OS Remote System connection, DSN9B, as shown in Figure 28-29.

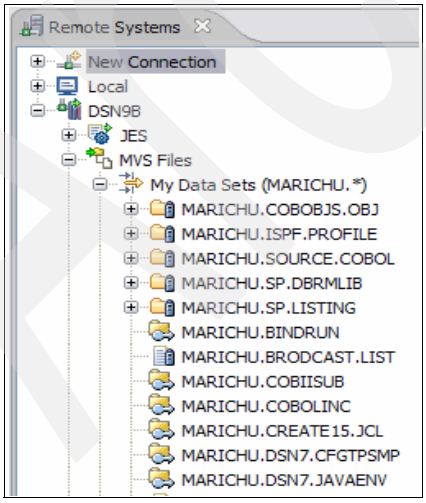

Figure 28-29 RDz Remote Systems view, MVS files

- We first identify the data sets needed for deploying the COBOL stored procedure. Click the **Data sets for Deploy** tab. Type the names for the data sets, as shown in Table 28-8.

Table 28-8 Data sets for Deploy in RDz v7

Data set description	What we typed in
Data set Qualifier for Compile Errors	MARICHU.ERRCOB
Object Deck Data Set	MARICHU.COBOBJS.OBJ
Copy Libraries[a]	MARICHU.SOURCE.COBOL
Listing Output Data Set	MARICHU.SP.LISTING
Database Request Module Location(DBRM)	MARICHU.SP.DBRMLIB
Load Module Location	MEL.DSN9.RUNLIB.LOAD[b]
Link Libraries	SCEELKED DSN.DSN9.SDSNLOAD

a. Optional. Specify whether you have included copylibs.
b. Needs to be APF-authorized.

Note: to specify multiple data sets, separate your entries with a blank.

- Click the **Save** icon, or Ctrl-S to save your changes.
- Right-click **EMPDTLC** and then click **Deploy.**[3]

The Data Output view displays the status and messages during compilation and registration. If there are compile errors, you can view them in the Remote Error List tab of the Output View, as shown in Figure 28-30.

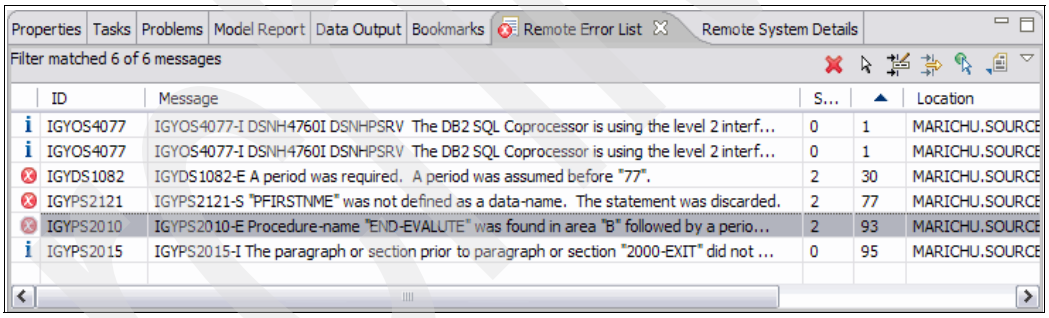

Figure 28-30 RDz Remote Error list

Note: Double-click an error in the Remote Error list to go straight to the line in the source code where the compile error is. This opens up a System z LPEX editor window.

28.4.5 Create and register the procedure in the DB2 Catalog

Registering outside RDz

Whether using the RDz or not, the CREATE PROCEDURE registration statement needs to specify the RUN OPTIONS parm, which includes:

- Parm TEST
- Sub-parm IP address

[3] Do not click Deploy... (with the ellipsis), this launches the Deploy Wizard which is supported only for SQL and Java stored procedures.

The current workstation IP address is determined by issuing an IPCONFIG command from a DOS command prompt on the workstation, as shown in Example 28-13. Workstations that are processing on a multi-network node, such as a home network, can be specified.

Example 28-13 Determine workstation IP address

```
Windows IP Configuration
Ethernet adapter Wireless Network Connection 4:
Media State . . . . . . . . . . . : Media disconnected
Ethernet adapter Local Area Connection:
      Connection-specific DNS Suffix  . : svl.ibm.com
      IP Address. . . . . . . . . . . . : 9.30.28.118
      Subnet Mask . . . . . . . . . . . : 255.255.255.0
      Default Gateway . . . . . . . . . : 9.30.28.1
```

We used the default listening port of 8001, which can alternatively be set to a different value. The port value is specified directly after the IP address in the RUN OPTIONS parm. See Example 28-14.

Example 28-14 CREATE PROCEDURE definition showing the IP address and port

```
CREATE PROCEDURE MARICHU.EMPDTLC ( IN PEMPNO CHAR(6), ...
   EXTERNAL NAME EMPDTLC
   LANGUAGE COBOL
   WLM ENVIRONMENT DSN9WL4
   RUN OPTIONS 'TEST(,,,TCPIP&9.30.28.118%8001:*)'
```

Registering with RDz

When we deployed our COBOL stored procedure from RDz, the tooling registered the stored procedure into the catalog. So, you don't have anything else to do.

28.4.6 Debugging using RDz v7

This section describes how to debug the same EMPDTLC COBOL stored procedure using the Rational Developer v7 for System Z. The IBM Debug Tool must be set up as described in 28.4.2, "Prerequisites and setup" on page 761, and the steps in 28.4.3, "Create the COBOL stored procedure source file" on page 764 must be completed.

Ensure correct TCP/IP address setting in Run Options

Before you start debugging, make sure that your IP address has not changed from when you built the COBOL stored procedure for debug.

- Without RDz

  You can use DSNTEP2 or SPUFI to register your stored procedure. Remember, to specify the TCP/IP address for the client, and not VADTCPIP, in the Run Options.

  If you have a new IP address (for example, connecting from home), and you registered the stored procedure using the above methods, you need to respecify the new TCP/IP address in the Run Options by issuing an ALTER DDL statement as shown in Example 28-15.

  Example 28-15 ALTER PROCEDURE for TCP/IP address

```
ALTER PROCEDURE MARICHU.EMPDTLC RUN OPTIONS
'TEST(,,,VADTCPIP&9.112.68.25%8001:*)'
```

- With RDz

  Alternatively, you can use RDz to change the run options dynamically. This avoids the need of issuing an ALTER PROCEDURE. However, you will need to drop and rebuild your stored procedure. To do this:

  - Right-click **EMPDTLC** and then click **Open**.
  - In the Editor, click the **Options** tab and uncheck Build Stored Procedure for Debugging.
  - Recheck the Build Stored Procedure for Debugging. This will capture the latest IP address of the client and generate a new string for the Run Options.
  - Click the **Save** icon or press Ctrl-S again.
  - In the Database Explorer, expand the DSN9 connection to the Stored Procedures folder.
  - Right-click **EMPDTLC** and then click **Drop**.
  - In the Data Project Explorer, right-click **EMPDTLC** and then click **Deploy**.

Run in debug mode

- **Start → All Programs → IBM Software Development Platform → Rational Developer for System z → Rational Developer for System z**.
- Select the default workspace **RDz_Test** and then click **OK**.

Running our COBOL stored procedure is performed from the Data Perspective. There are multiple ways to open the Data Perspective view. We opened the Data Perspective view from the top right-side toolbar, selecting the **table icon with a +**. Clicking this icon expands the Perspective selection list. Sometimes the Data keyword is displayed in this area as well. See Figure 28-31.

.

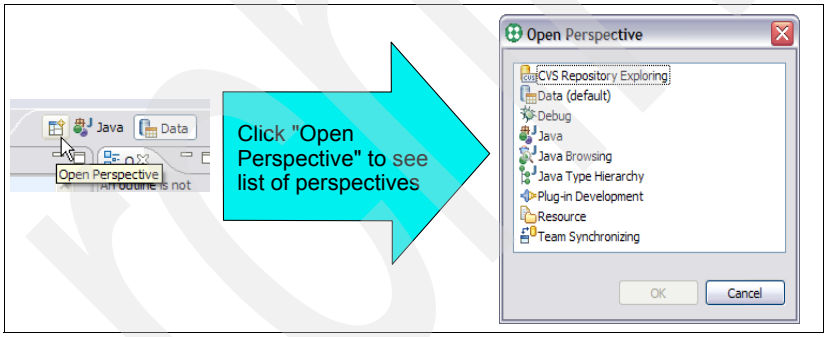

Figure 28-31 Open the RDz Data Perspective

- In the Data Project Explorer, right-click **EMPDTLC** and then click **Run**.
- RDz will request permission to launch the Debug Perspective. Click **Yes**.
- The Debug Perspective is launched, as in Figure 28-32.

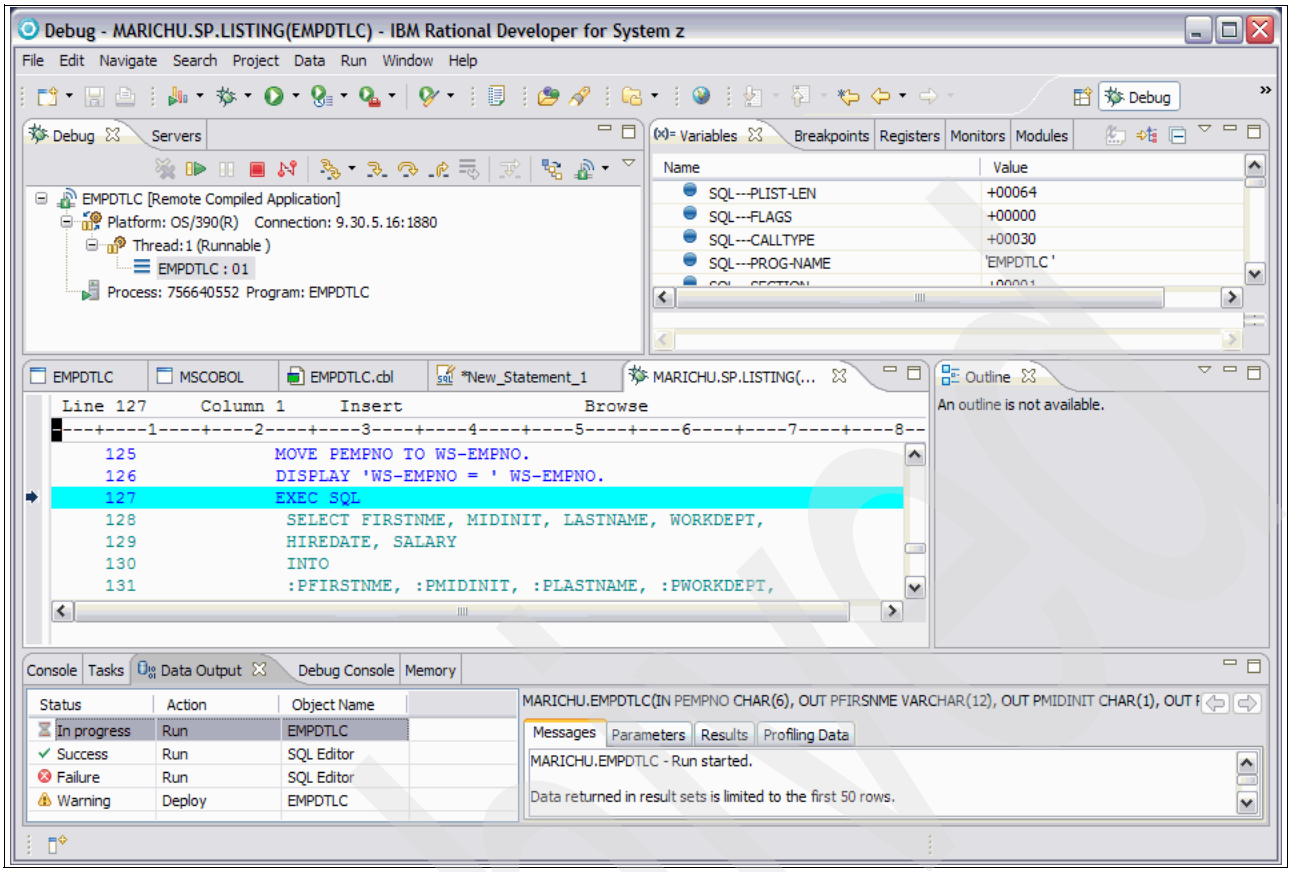

Figure 28-32 RDz Debug Perspective

At this point, you can use the same views and actions as discussed in 28.3, “Debugging SQL procedures on z/OS, Linux, UNIX, and Windows” on page 746.

28.5 Debugging options for DB2 Java procedures on z/OS

We briefly outline the options on DB2 V8 and V9 for debugging Java stored procedures.

28.5.1 DB2 9 for z/OS

The Unified Debugger allows you to debug a Java stored procedure with DB2 for z/OS V9. The setup and steps to prepare your Java stored procedure for debug are the same as for SQL stored procedures, as discussed in 28.3.3, “Debugging SQL stored procedures” on page 749. In the next sections, we look into how you can use the Unified Debugger to assist you in debugging your Java stored procedures.

28.5.2 DB2 for z/OS V8

There are three alternatives that you can use to debug DB2 Java stored procedures on z/OS:

- Create comparable DB2 for z/OS DDL and SQL on a DB2 for Linux, UNIX, and Windows; then use the Unified Debugger against the LUW server.

  When the debugging of the Java stored procedure is successfully completed, copy and paste the stored procedure to the DB2 for z/OS server. See “Drag and Drop or Copy and

Paste" on page 715. This requires the same z/OS setup for Java stored procedures as described in 27.2.2, "DB2 for z/OS setup" on page 650.

- Write System.out.println, System.err.println lines in your Java code to STDOUT and STDERR.

  To use the default directory structure for writing STDOUT and STDERR requires the presence of the HFS /tmp/java directory. When this directory exists, any messages written to STDOUT will appear in /tmp/java/server_stdout.txt. Conversely, any messages written to STDERR will appear in /tmp/java/server_stderr.txt. If the Java directory does not physically exist under the /tmp directory, these messages will not be written.

  Alternatively, you may want to isolate the Java sysprint lines more granularly to a specific directory for the Java stored procedures executing in a specific WLM AE. This can be done by including a WORK_DEPT environment variable pointing to a separate HFS directory in the WLM procedure JAVAENV statement.

- Convert your Java stored procedure to a Java application, and debug using the Java perspective in IBM Data Studio. See Chapter 29, "Debugging DB2 V8 Java procedures with Data Studio" on page 785.

28.6 Debugging Java procedures on Linux, UNIX, and Windows

While the main focus of this book is stored procedures on DB2 for z/OS, this next section describes debugging Java stored procedures using the IBM Data Studio Unified Debugger. Some customers prototype their application development on DB2 for Linux, UNIX and Windows (LUW) before porting to DB2 for z/OS. IBM Data Studio allows you to deploy directly to DB2 on z/OS directly from DB2 on LUW. When this environment is possible, then this debugging option can be used.

In IBM Data Studio, we copied the EmpDtlsJ stored procedure from our DB9A project on z/OS, then pasted this stored procedure to our DB2 9 on LUW SAMPLE project. Since our Java stored procedure created on z/OS used the EMP table, which is comparable to the EMPLOYEE table on Windows, the only change needed to the stored procedure on Windows was to change the SQL select statement to point to the Windows EMPLOYEE table instead of the z/OS EMP table.

Our case study for this section performs the following steps:

1. Start IBM Data Studio and create database connections.
   - Create a database connection to DB9A, our DB2 for z/OS V9 location.
   - Create a database connection to SAMPLE, our DB2 9 on LUW database.
2. Create LUW project
   - Create the project ProjectLUW with a target connection to SAMPLE.
3. In the Database Explorer, 28.6.3, "Drag and drop EmpDtlsJ to Windows" on page 780, project PROJECTLUW.
4. Using the Routine Editor, 28.6.4, "Modify the table DEVL7083.EMP to EMPLOYEE" on page 781.
5. Deploy EmpDtlsJ for debug.
6. Run EmpDtlsJ in debug mode.

28.6.1 Start IBM Data Studio and create database connections

Start IBM Data Studio and create connections to DB9A and SAMPLE. In “Create a database connection” on page 764, we created a connection to DB9A. We follow the same steps to create a connection to the SAMPLE database.

1. Click the **New Connection** icon.
2. In the Connection wizard, type:
   – Database: `SAMPLE`
   – Host: `localhost` (or the domain name for the DB2 on LUW server)
   – Port: `50000`
   – Userid and password: your Windows userid and password to connect to this server
3. Click **Finish**.

28.6.2 Create a project to target each server

In “Create a Database Development Project” on page 765, we created a new Data Development Project, DEVL7083. We followed the same steps to create a Data Development project for our DB2 9 on LUW connection.

1. Click **File** → **New** → **Data Development Project.** The New Data Development Project wizard is launched.
2. In the Data Development Project page, type `ProjectLUW`, for the Project Name. Click **Next**.
3. In the Select Connection page, select **SAMPLE**. Click **Finish**.

28.6.3 Drag and drop[4] EmpDtlsJ to Windows

The IBM Data Studio can select a specific stored procedure from one server and drag and drop it to a project for another server. We can do this process between like and unlike servers. That is, we can copy a stored procedure created on DB2 for z/OS, and paste it to a project that targets a DB2 on LUW; modify the stored procedure as necessary; and deploy to the DB2 LUW server. This is the scenario we are performing in this example.

- In Database Explorer, select **DB9A** → **Schemas** → **DEVL7083** → **Stored Procedures**, then select **DEVL7083.EMPDTLSJ**.
- Drag and drop DEVL7083.EMPDTLSJ to ProjectLUW → Stored Procedures folder in the Data Project Explorer.

Alternatively, in the Database Explorer, you can right-click **DEVL7083.EMPDTLSJ**, then click **Copy**. Then in the Data Project Explorer, right-click the Stored Procedures folder and click **Paste.**

[4] Copy and Paste performs the same function as a drag and drop.

Figure 28-33 shows the state of ProjectLUW after the drag and drop. We also opened the routine editor after the drag and drop.

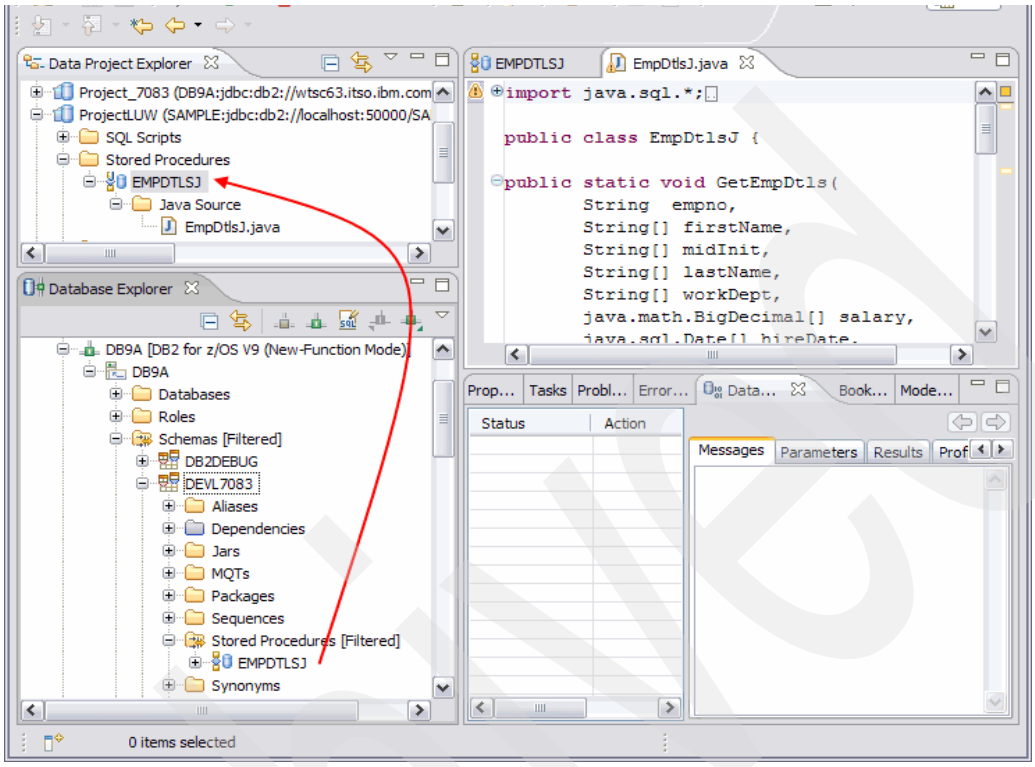

Figure 28-33 Drag and drop stored procedure from z/OS to Windows

28.6.4 Modify the table DEVL7083.EMP to EMPLOYEE

After dragging DEVL7083.EMPDTLSJ, edit the stored procedure source.

- Right-click the stored procedure and click **Open** to display the stored procedure in the routine editor.
- In the Configuration tab of the editor, click **EmpDtlsJ.java**. The Java Editor will be launched on another tab in the Editor view.
- In the EmpDtlsJ.java tab, locate the following statement:

  sql = “SELECT * FROM DEVL7083.EMP WHERE EMPNO = '"+ empno + “'”;

  and change DEVL7083.EMP table to EMPLOYEE.

Finally, save the stored procedure.

28.6.5 Deploy EmpDtlsJ for debug

Right-click **DEVL7083.EMPDTLSJ** and then click **Deploy**. This launches the Deploy wizard. See 27.6, “Deploying a stored procedure” on page 703, for a review of the Deploy wizard and the steps in deploying a stored procedure.

In the Deploy wizard’s Deploy Options page, click the **Enable Debugging** checkbox. The Compile Options field is updated with a **-g**. Click **Finish**.

The message in Example 28-16 is returned in the Data Output view's Messages tab after a successful deploy for debug.

Example 28-16 Deploy Java stored procedure in debug mode on Windows

```
MARICHU.EMPDTLSJ - Deploy for debug started.
DROP SPECIFIC PROCEDURE MARICHU.EMPDTLSJ

MARICHU.EMPDTLSJ - Drop stored procedure completed.

MARICHU.EMPDTLSJ - Created temporary working directory
C:\$ViperII_Stuff\Workspaces\IBMDataStudio11_ITSO\.metadata\.plugins\com.ibm.datatools.db2.
routines.deploy.ui\bld1194560817813.

C:\IBM_JDK15\bin\javac -g -classpath ".;C:\SQLLIB\java\sqlj.zip;C:\Program
Files\IBM\SDP70Shared\plugins\com.ibm.datatools.db2_1.0.100.v200709182330\driver\db2jcc.jar
;C:\Program
Files\IBM\SDP70Shared\plugins\com.ibm.datatools.db2_1.0.100.v200709182330\driver\db2jcc_lic
ense_cisuz.jar" -g EmpDtlsJ.java

MARICHU.EMPDTLSJ - Javac completed.

C:\IBM_JDK15\bin\jar cf spjar.jar  EmpDtlsJ.class

MARICHU.EMPDTLSJ - Jar file created.

Call SQLJ.DB2_REPLACE_JAR
(<<C:\$ViperII_Stuff\Workspaces\IBMDataStudio11_ITSO\.metadata\.plugins\com.ibm.datatools.d
b2.routines.deploy.ui\bld1194560817813\spjar.jar>>, 'MARICHU.SQL7103011562640')

MARICHU.EMPDTLSJ - SQLJ.DB2_REPLACE_JAR using Jar name MARICHU.SQL7103011562640 completed.

Call sqlj.refresh_classes( )

MARICHU.EMPDTLSJ - sqlj.refresh_classes completed.

Call SQLJ.DB2_UPDATEJARINFO ('MARICHU.SQL7103011562640', 'EmpDtlsJ',
<<C:\$ViperII_Stuff\Workspaces\IBMDataStudio11_ITSO\.metadata\.plugins\com.ibm.datatools.db
2.routines.deploy.ui\bld1194560817813\EmpDtlsJ.java>>)

MARICHU.EMPDTLSJ - Source saved to the server.

CREATE PROCEDURE EMPDTLSJ ( IN empno CHAR(32),
                            OUT firstName CHAR(32),
                            OUT midInit CHAR(32),
                            OUT lastName CHAR(32),
                            OUT workDept CHAR(32),
                            OUT salary DECIMAL(5,0),
                            OUT hireDate DATE,
                            OUT outputMessage CHAR(32) )
   SPECIFIC EMPDTLSJ
   NOT DETERMINISTIC
   LANGUAGE JAVA
   EXTERNAL NAME 'MARICHU.SQL7103011562640:EmpDtlsJ.GetEmpDtls'
   FENCED
   PARAMETER STYLE JAVA

MARICHU.EMPDTLSJ - Create stored procedure completed.
```

```
MARICHU.EMPDTLSJ - Removed temporary working directory
C:\$ViperII_Stuff\Workspaces\IBMDataStudio11_ITSO\.metadata\.plugins\com.ibm.datatools.db2.
routines.deploy.ui\bld1194560817813.

MARICHU.EMPDTLSJ - Deploy for debug successful.
```

28.6.6 Run EmpDtlsJ in debug mode

The final step for our Java stored procedure debugging example on Windows is to debug the stored procedure.

- First, we make sure that there is a Session Manager setup. In 28.3.1, "Setting up the Session Manager" on page 747, we discussed the steps for setting up and launching the Session Manager on both the workstation and the server. For this example, we launch the Session Manager on the workstation.

  Example 28-17 shows invoking db2dbgm.bat from the client to start the Session Manager.

  Example 28-17 Start the Session Manager on the client

```
C:\Program Files\IBM\SDP70\dwb\bin>db2dbgm.bat
args[0]: -port
args[1]: 4554
args[2]: -timeout
args[3]: 50
Code Level: 070418
Debug Session Manager started on IP: 9.30.28.118 - port: 4554
idleTimeOut: 50
```

- Update the DB2 Routine Session Manager Location in the **Window** → **Preferences** → **Run / Debug** → **DB2 Stored Procedure Debugger** page with the IP address and port number of the Session Manager. See Figure 28-4, "Preferences for using the client Session Manager" on page 748.
- From the Data Project Explorer → **ProjectLUW** → Stored Procedures folder, select **DEVL7083.EMPDTLSJ.** and right-click **Debug**.

  This starts the Java stored procedure in debug mode running, as shown in Figure 28-34. At this point, the behavior of the Unified Debugger when processing a Java stored procedure is the same as when processing an SQL stored procedure, which we discussed in 28.3.3, "Debugging SQL stored procedures" on page 749.

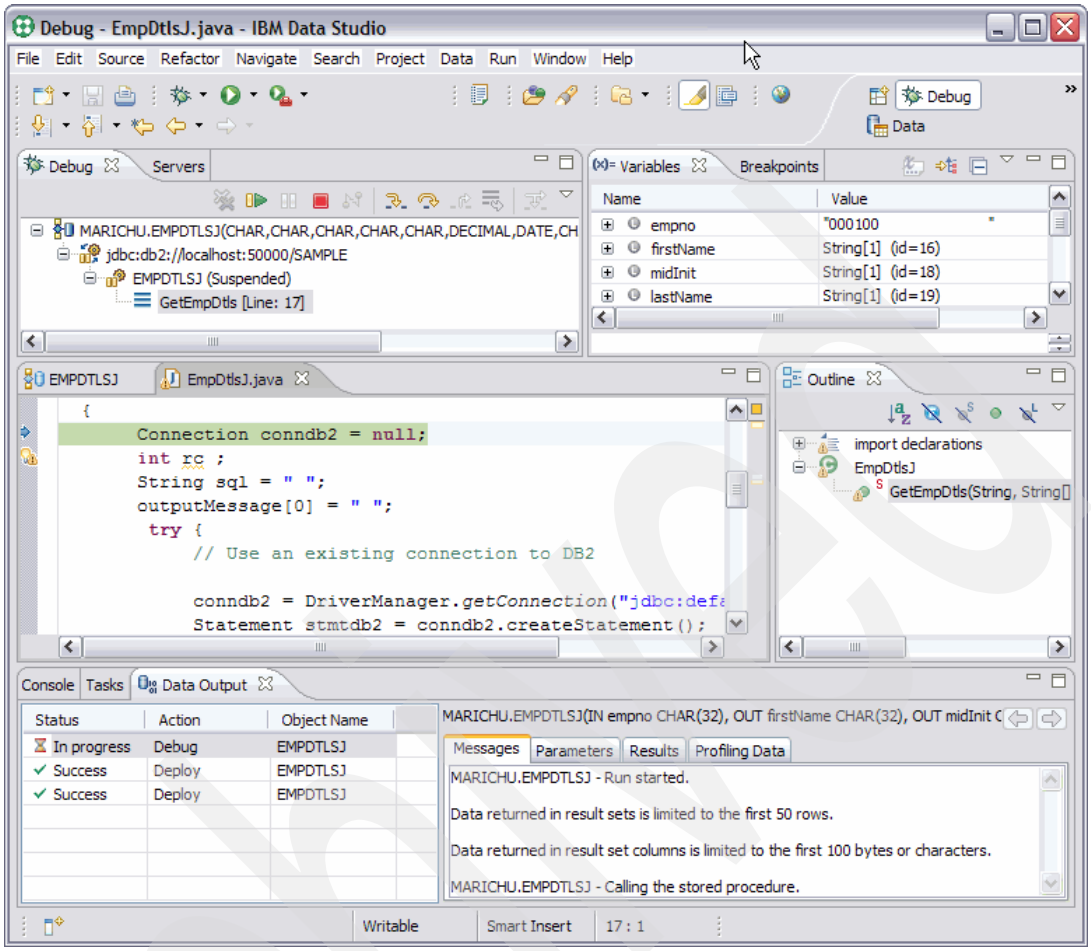

Figure 28-34 Unified Debugger debugging a Java stored procedure on Windows

29

Debugging DB2 V8 Java procedures with Data Studio

Currently, only Java stored procedures on DB2 for z/OS V9 can be debugged using the IBM Data Studio Unified Debugger. Java stored procedures on DB2 for z/OS V8 can be debugged using the methods described in Chapter 28, "Tools for debugging DB2 stored procedures" on page 735[1], namely:

- Create comparable DB2 for z/OS DDL and SQL on a DB2 for Linux, UNIX and Windows; then use the Unified Debugger against the LUW server.
- Write System.out.println, System.err.println lines in your Java code to STDOUT, and STDERR

  or

- Convert the Java stored procedure to a Java application, and use the Java Debugger included in IBM Data Studio.

In this chapter we describe the third option of using IBM Data Studio to debug Java stored procedures converted to Java programs. Our case study uses both JDBC and SQLJ stored procedures.

This chapter contains the following topics:

- Debugging JDBC procedures converted to JDBC applications
- Debugging SQLJ procedures converted to SQLJ applications

[1] In the previous edition of this book, there were additional methods offered involving the WebSphere Application Developer.

29.1 Debugging JDBC procedures converted to JDBC applications

This section describes using IBM Data Studio to debug a JDBC application that is converted from a JDBC stored procedure on DB2 V8 for z/OS.

As we have seen in the last chapter, the IBM Data Studio launches with the Data Perspective. This time, we want to use the Java Perspective and create a Java project to run our test cases.

29.1.1 Switch to the Java Perspective

In the main toolbar, click **Window** → **Open Perspective** → **Java**. You may not see the Java choice at once, in which case you may have to select **Other** and then select **Java** from the list. See Figure 29-1.

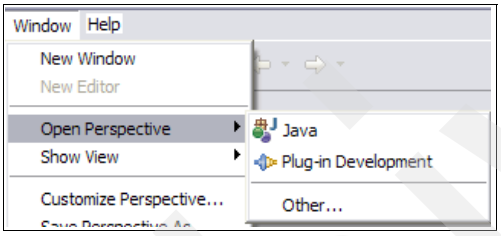

Figure 29-1 Switching perspective to Java Perspective

The Java Perspective is displayed in Figure 29-2 on page 787. This is identified by the word Java in the title bar, and in the top-right tab.

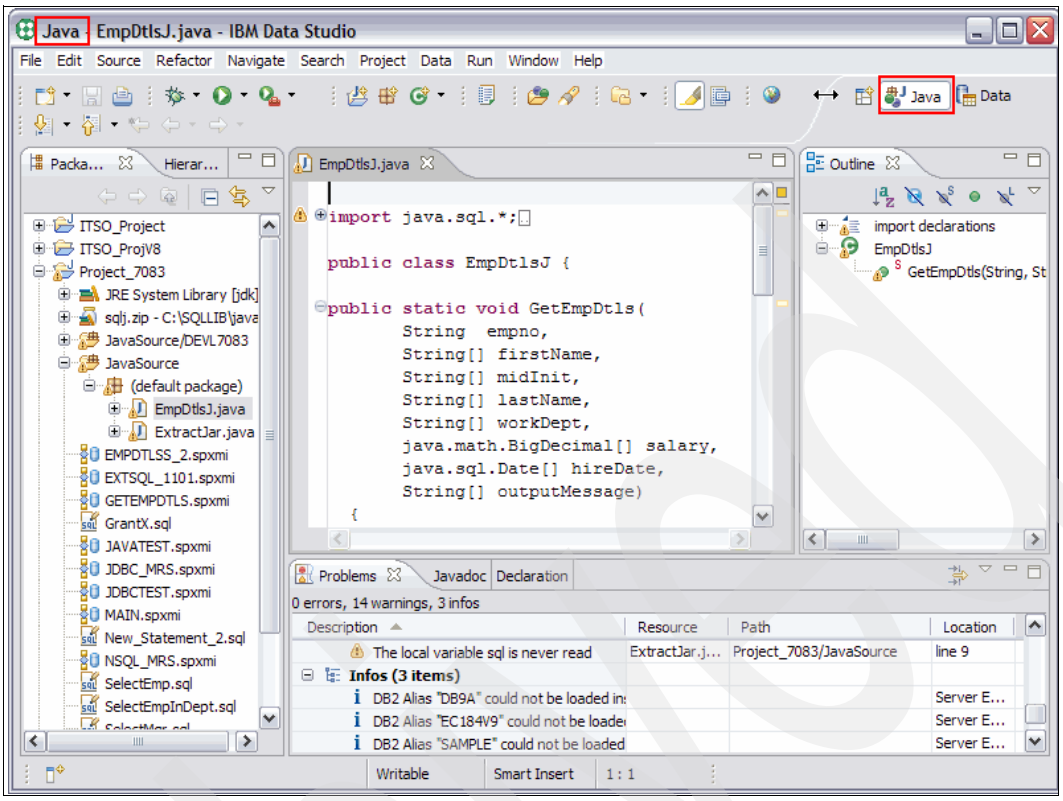

Figure 29-2 The Java Perspective in IBM Data Studio

The default views available in the Java Perspective differ slightly from the Data Perspective. In the Java Perspective, we have:

- Package Explorer - This displays the resources in each project, and displays them as files rather than as database objects. So, for example, the stored procedure EmpDtlsJ is shown as EmpDtlsJ.spxmi.
- Hierarchy - This displays the class hierarchy for a given class. We will not use this view.
- Editor view - This is the middle part of the workspace and displays the current selected resource for edition. This view supports multiple types of editors (for example XML editor, Java Editor). For our case study we use the Java Editor, which shows the source code of our Java stored procedure.
- Outline view - This shows the sections of our Java source, for example import section, variable declarations, methods, inner classes, and so on, in an outline form.
- Output view → **Problems** - This tab in the Output view shows us what Java compilation errors were found.

29.1.2 Create a Java Project

- Click **File** → **New** → **Project** → **Other** → **Java** → **Java Project** as shown in Figure 29-3, to create a new Java Project.

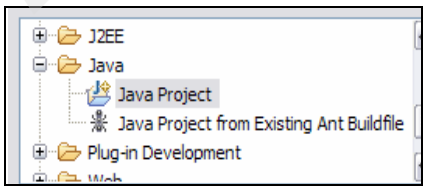

Figure 29-3 Creating a New Java Project

- From the New Project window, select **Java** in the left-hand window, and **Java Project** in the right-hand window, as shown in Figure 29-3, and click **Next.**

  The New Java Project wizard is launched. Type `JAVASPDEBUG` as the Project Name. We use the default settings for everything else. See Figure 29-4. Click **Next**.

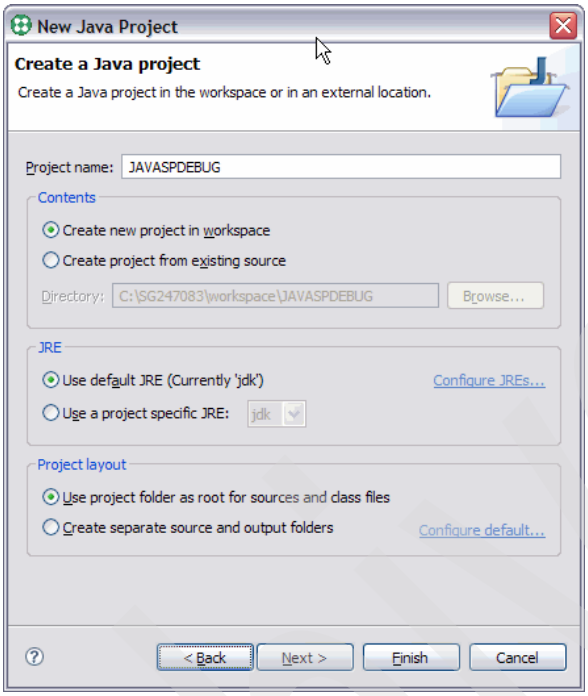

Figure 29-4 New Java Project wizard, Create a Java project

- The next window shown is the Java Settings window, shown in Figure 29-5. Click the **Libraries** tab.

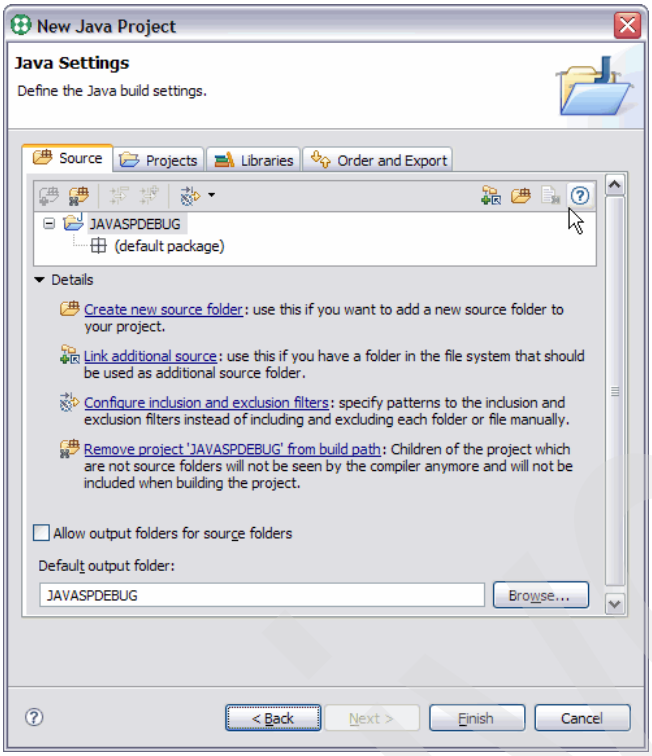

Figure 29-5 Define the Java build settings - Source

- The Libraries window is where we define the Java build settings. We need to add external jars to our project. To add the jars, click **Add External JARS** on the right-hand side of the window.

  In the File Browser dialog, browse the folder in the IBM Data Studio install directory that includes:

  – db2jcc.jar
  – db2jcc_license_cisuz.jar

  If you installed IBM Data Studio using the default settings, this directory will be:

  C:\Program Files\IBM\SDP70Shared\plugins\com.ibm.datatools.db2_<some version #>\driver

  Click **Open**.

  The Libraries window now includes the files shown in Figure 29-6.

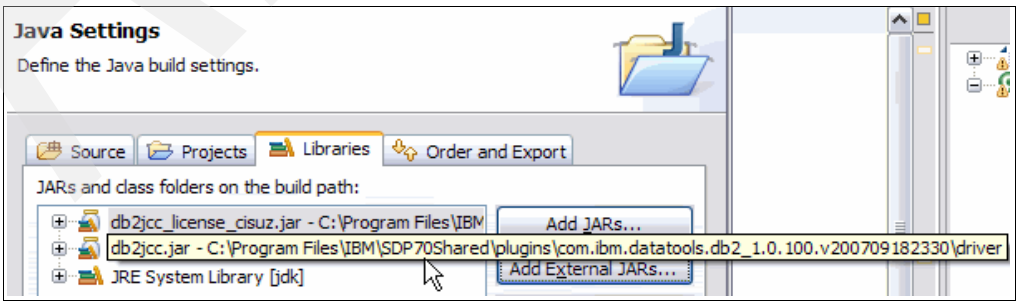

Figure 29-6 Define the Java build settings - Libraries

Click **Finish** at the bottom of the window, and the template for our Java application will be generated.

29.1.3 Copy EmpDtlsJ.java to JAVASPDEBUG project

In the Java Perspective's Package Explorer view, we now see both the Data Development Projects we created in the last two chapters, as well as the new JAVASPDEBUG Java project.

- Open **Project_7083** → **Java Source** → (default package) folder. Right-click **EmpDtlsJ.java** and then click **Copy**. See Figure 29-7.

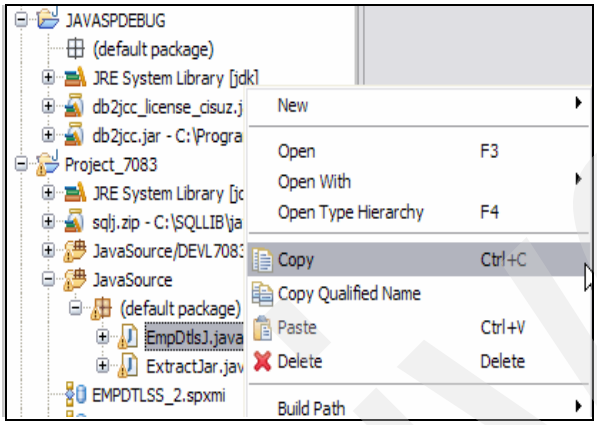

Figure 29-7 Java Perspective, Package Explorer, Copy EmpRsetJ.java

- Now, right-click project **JAVASPDEUG** and click **Paste**, as shown in Figure 29-8.

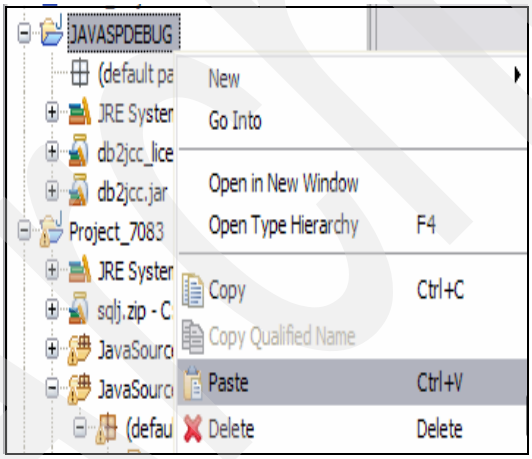

Figure 29-8 Java Perspective, Package Explorer, Paste EmpDtlsJ.java

- The Package Explorer view now has the EmpDtlsJ.java source file. It is located in the default package folder under the JAVASPDEBUG object. Expand this folder and double-click **EmpDtlsJ.java**. This opens the resource in the Editor view using the Java Editor.

29.1.4 Modify the Java stored procedure code

Now, we convert the Java stored procedure code to a Java application.

We discussed the Java Editor in Chapter 27, “The IBM Data Studio” on page 643, “Java Editor” on page 677. The Java Editor reports syntax errors as you do your modifications. When you save the file, it does a full compile of your Java source, and report the problems in the Problems tab of the Output view.

In the Java Editor, do the following:

- Convert the stored procedure method.
  a. Replace the main stored procedure method, GetEmpDtls with a main.
  b. Replace the parameters with `(String args[])`.
  c. Convert the parameter list to declared variables and initialize the input variables to the values passed in as arguments to the application.
  d. Define all output parameters as arrays of one element.

  Table 29-1 summarizes the changes to the stored procedure method.

Table 29-1 Converting the stored procedure method to a main method

Stored procedure code	Converted Java application
Changes to the Stored Procedure Method	

public static void GetEmpDtls(
String empno,
String[] firstName,
String[] midInit,
String[] lastName,
String[] workDept,
java.math.BigDecima[] salary,
String[] outputMessage)
	public static void main (String args[])
	{
	String empno;
	empno=args[0];
	String[] firstName = new String[1];
	String[] midInit = new String[1];
	String[] lastName = new String[1];
	String[] workDept = new String[1];
	java.math.BigDecimal[] salary = new java.math.BigDecimal[1];
	String[] outputMessage = new String[1];

- Modify the DB2 server connection information.
  a. Locate the DB2 server connection information in the Java source.
  b. Create variables for:
     - Connection con
     - String url
     - String userid
     - String pwd
  c. In IBM Data Studio, select **Database Explorer**, click the **DB9A** connection, and then click the **Properties** tab, as shown in Figure 29-9.

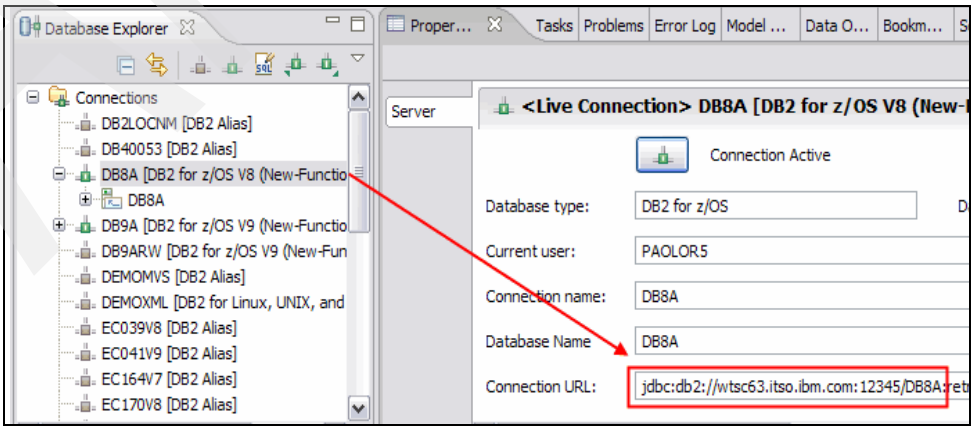

Figure 29-9 Connection Properties ->Connection URL

d. Copy the Connection url value and set the url variable to this value.
e. Set the userid and password variables to the login ID and password we used to connect to DB9A server.
f. Code the Class.forName class to com.ibm.db2.jcc.DB2Driver, the IBM Universal driver class.
g. Set the conndb2 connection to the connection obtained from the given connection info.

Table 29-2 summarizes the changes to the connection string.

Table 29-2 Changes to the connection string

Stored procedure code	Converted Java application
Connection conndb2 = null; conndb2 = DriverManager.getConnection("jdbc:default:connection");	The connection statements in the stored procedure needs to be changed to use the specific driver class, userid and password, as follows: Connection conndb2 = null; String url = "jdbc:db2://wtsc63.itso.ibm.com:12347/DB9A"; String userid = "PAOLOR5"; String pwd = "PUP4SALE"; Class.forName("com.ibm.db2.jcc.DB2Driver").newInstance (); conndb2 = DriverManager.getConnection(url, userid, pwd);

- Report the output values.

  As a Java stored procedure, the output parameters reported the values returned from our query. As a Java application, we need to report the values of the output parameters in some other way. We choose to issue System.out.println statements.

  Add the statements in Figure 29-10after retrieving the values from the result set.

```
System.out.println("First Name: " + firstName[0]);
System.out.println("Middle Initial: " + midInit[0]);
System.out.println("Last Name: " + lastName[0]);
System.out.println("Work Dept: " + workDept[0]);
System.out.println("Salary: " + salary[0]);
System.out.println("Hire Date: " + hireDate[0]);
```

Figure 29-10 System.out.println statements

29.1.5 Set breakpoints

While we have the Java source in edit mode, we are going to set breakpoints. Locate the lines:

```
empno= args[0];
Connection conndb2 = null;
System.out.println("First Name: " + firstName[0]);
```

In the prefix area on the left side of the editor, double-click these lines as shown in Figure 29-11. A breakpoint decorator (shaded dot) is created.

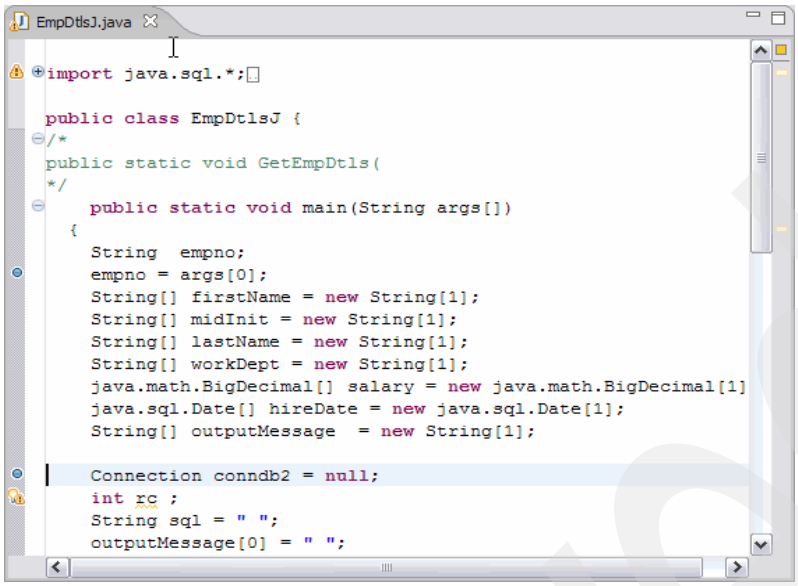

Figure 29-11 Add breakpoint for Java applications

29.1.6 Configure the Debug Launch Settings

1. From the Java Perspective, Package Explorer view, expand the default package folder. With the **EmpDtlsJ.java** file selected, click the **bug** icon in the tool bar on the top of the window to see the drop-down Debug options. Click **Debug,** as shown in Figure 29-12.

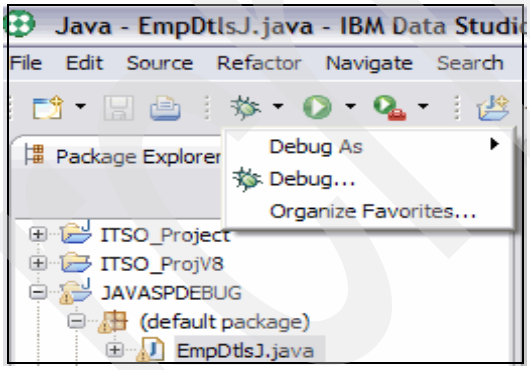

Figure 29-12 Select Debug configuration option

The first time you run or debug a resource in the Java Perspective, IBM Data Studio will ask you to configure your run/debug launch settings. The next time you run or debug a resource, this named launch configuration will be shown as the first entry in the Debug or Run drop-down options.

2. The next window is where we configure the debug options. From the Debug window, select **Java Application** and click the **New launch configuration** icon as shown in Figure 29-13.

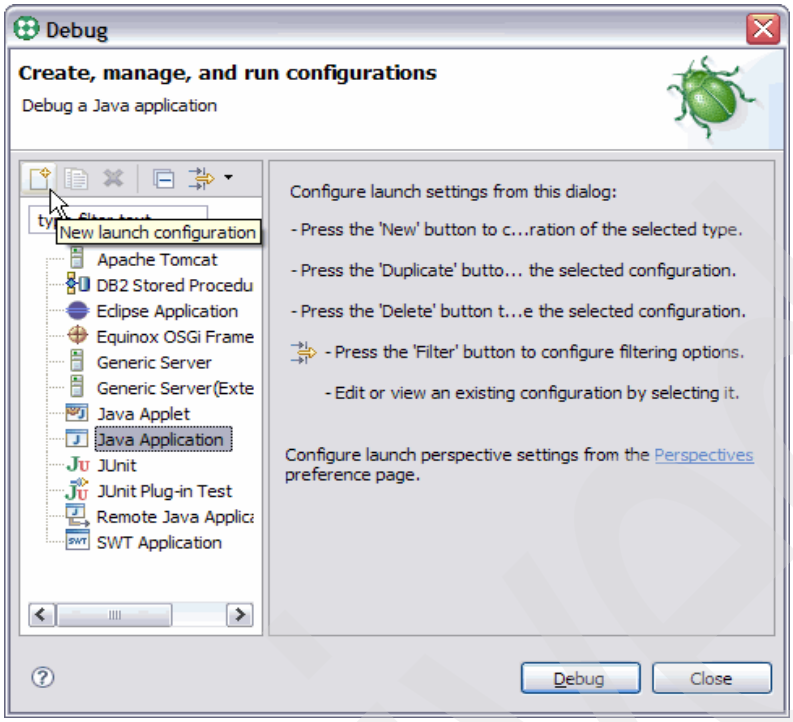

Figure 29-13 Configure Java Application for debug

3. In the Debug Main window, select **EmpDtlsJ** from the Configurations portion on the left. On the right side of the window, enter the information in Table 29-3.

Table 29-3 Debug settings for Java application

Field	Value
Name	Java_Debug
Project	JAVASPDEBUG
Main class	EmpDtlsJ.

The completed Main debug panel is shown in Figure 29-14.

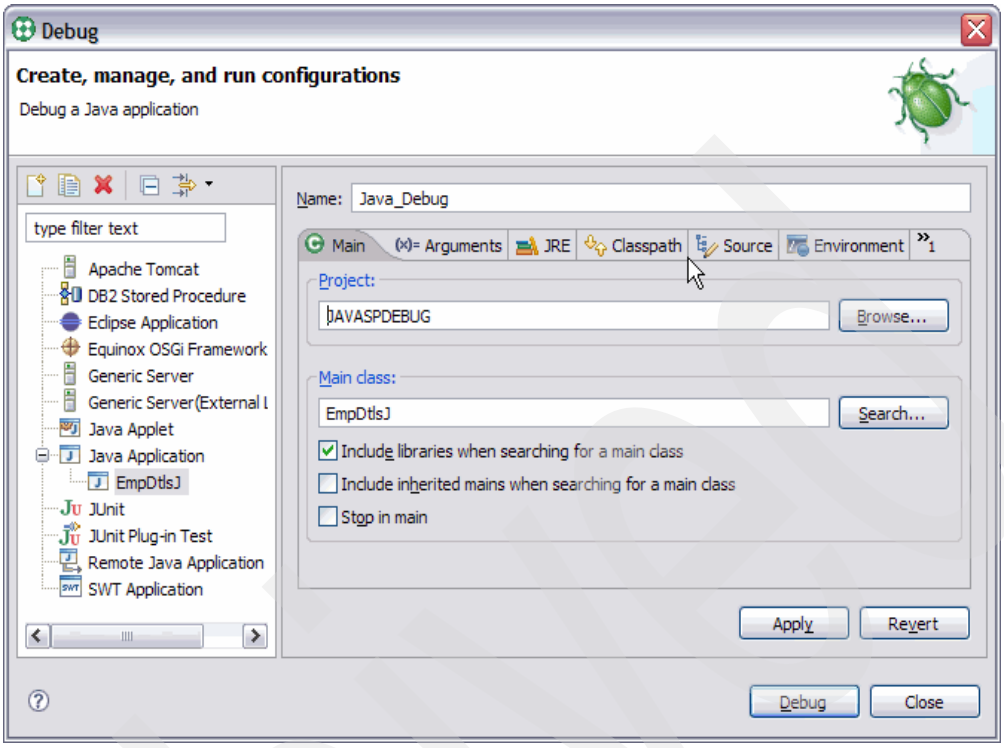

Figure 29-14 Java application Debug Main window definition

4. Click **Apply.**
5. Click the **Arguments** tab next to the Main tab.

   Our stored procedure requires input parameters to execute. These were removed as described when we modified the stored procedure in 29.1.4, “Modify the Java stored procedure code” . The Arguments window is where we specify these parameters. Enter `D11` for Department 11.
6. Click **Debug** to save these changes and start the debug session. Optionally, you can click **Apply**, then **Debug**. However, selecting **Debug** performs an implicit Apply, as shown in Figure 29-15.

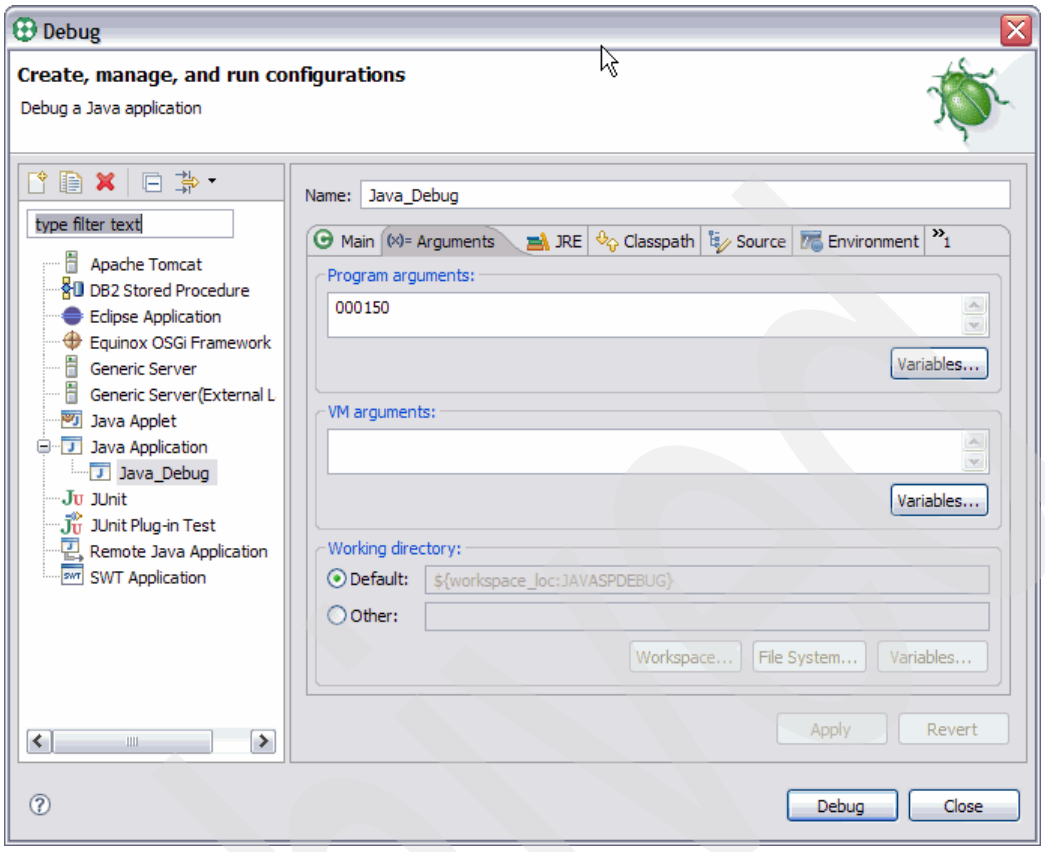

Figure 29-15 Java application Debug Arguments window definition

29.1.7 Debugging the application

A confirmation dialog to switch from the Java Perspective to the Debug Perspective may appear. Click **Yes**.

The Debug Perspective opens. This is the same Debug Perspective we introduced in “28.2, “The Unified Debugger” on page 738”, and in Chapter 29, “Debugging DB2 V8 Java procedures with Data Studio” on page 785”.

You are now ready to debug your Java application. As when using the Unified Debugger, you have the same capabilities for Step In, Step Over, Step Return, Resume, Pause, and Terminate in the execution of this application. You also have the same capabilities for managing breakpoints and variables.

Figure 29-16 shows the Console output from our System.out.println statements.

```
First Name: BRUCE
Middle Initial:
Last Name: ADAMSON
Work Dept: D11
Salary: 25280.00
Hire Date: 1972-02-12
```

Figure 29-16 Console output

Figure 29-17 shows the Debug Perspective for our converted Java stored procedure.

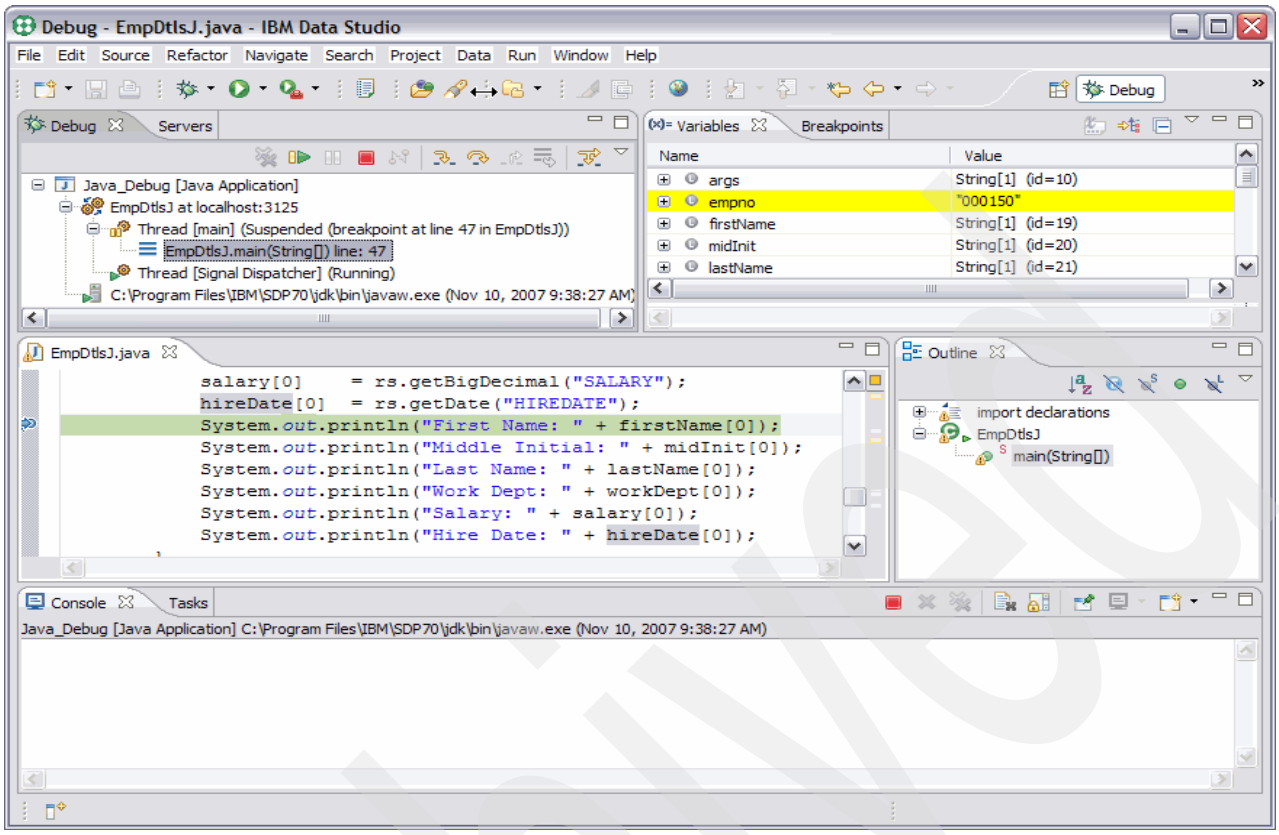

Figure 29-17 Debug Perspective at a breakpoint

29.2 Debugging SQLJ procedures converted to SQLJ applications

This section describes using IBM Data Studio to debug an SQLJ application that was converted from an SQLJ stored procedure on DB2 V8 for z/OS. SQLJ stored procedures targeting a DB2 9 for z/OS server can be debugged, without any conversion, using the Unified Debugger.

The steps to convert an SQLJ stored procedure to an SQLJ application are very similar to the steps described in 29.1, “Debugging JDBC procedures converted to JDBC applications” on page 786.

29.2.1 Create the SQLJSPDEBUG project

Create a new Java Project and name it SQLJSPDEBUG, adding the same external jars from the Libraries window as were added previously. Click **Finish**.

29.2.2 Copy SQLJ stored procedure source

The Java Perspective, Package Explorer view now has two Java projects: SQLJSPDEBUG in addition to JAVASPDEBUG. We will copy our SQLJ stored procedure source like we copied the Java source for our JDBC stored procedure.

Select **Project_7083** → **Java Source** → **EmpDtl1.sqlj**. Then right-click **Copy** as shown in Figure 29-18.

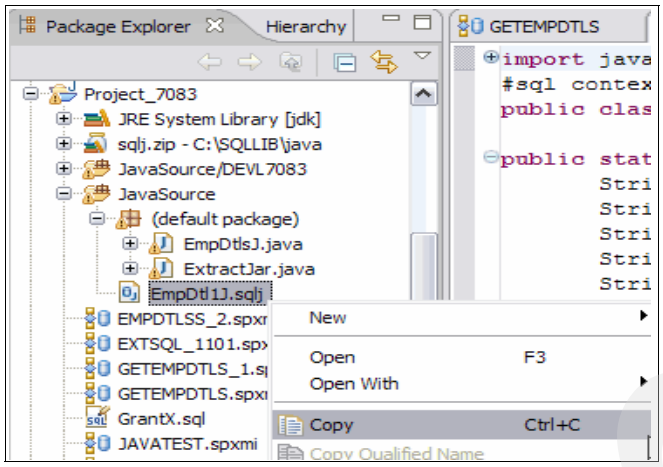

Figure 29-18 Copy SQLJ source

29.2.3 Paste into the SQLJSPDebug project

Select the **SQLJSPDEBUG** project. Right-click **Paste**, as shown in Figure 29-19.

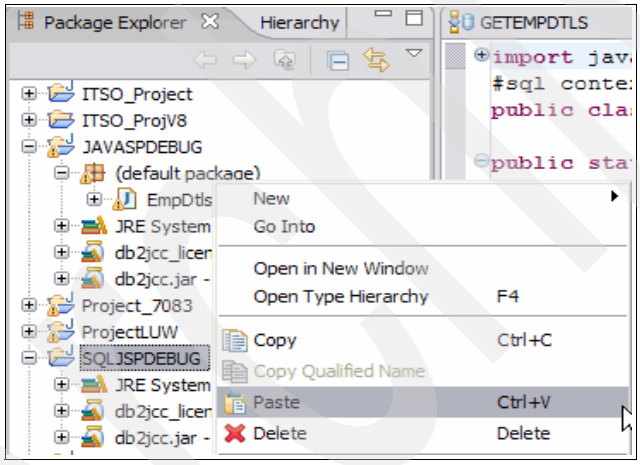

Figure 29-19 Paste into the SQLJSPDebug project

29.2.4 Add SQLJ support

- In the Java Perspective, Package Explorer, select project **SQLJSPDEBUG**. Right-click **Add SQLJ Support** as shown in Figure 29-20.

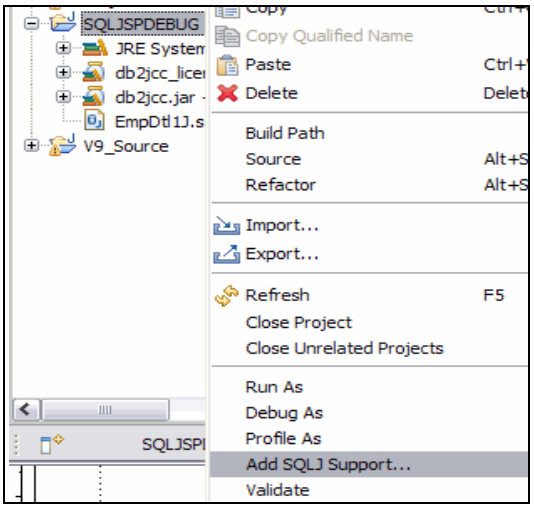

Figure 29-20 Add SQLJ Support

A list of available projects is returned. SQLJSPDEBUG is preselected as shown in Figure 29-21.

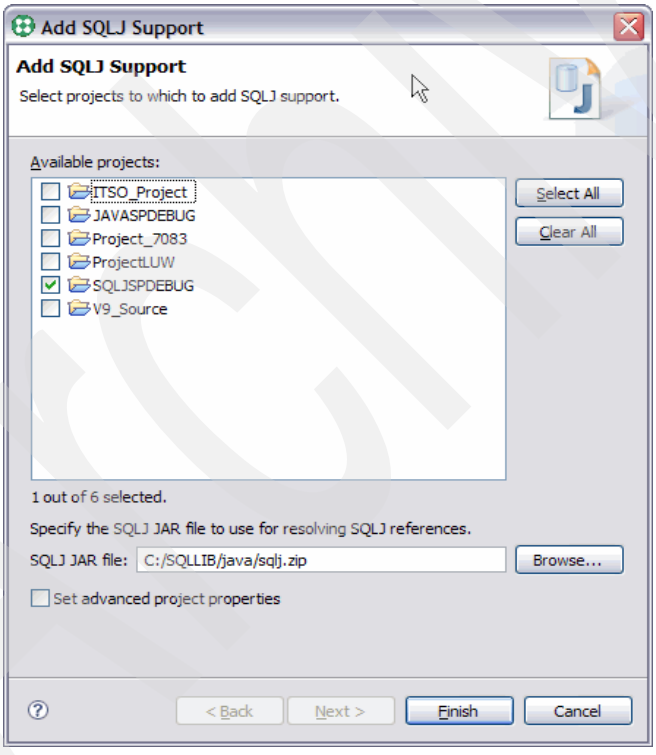

Figure 29-21 Select projects for SQLJ support

- Click **Finish.**

The SQLJSPDEBUG project now has the following objects added, as shown in Figure 29-22:

- SQLJAnt Scripts
- sqlj.zip
- EmpDtlAJ.sqlj

- EmpDtlAJ_SJProfile0.ser

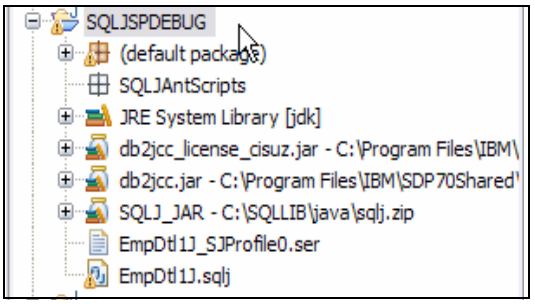

Figure 29-22 SQLJ support added to project

29.2.5 Modify the source code

We make the same changes as in 29.1.4, “Modify the Java stored procedure code” on page 790, to the SQLJ source code.

29.2.6 Set breakpoints

Before closing the editor, we set breakpoints in the prefix area for the following statements:

```
empno= args[0];
Connection conndb2 = null;
System.out.println("First Name: " + firstName[0]);
```

Click the **X** in the upper right corner to close the editor as shown in Figure 29-23. Reply `yes` when prompted to save the updates.

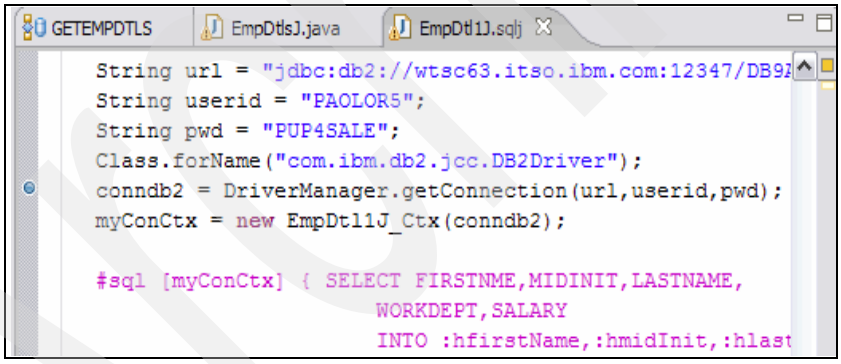

Figure 29-23 SQLJ application source with breakpoint

29.2.7 Configure the debug session

Next we configure a debug session for the SQLJ application.

- From the Java Perspective, Package Explorer, select **EmpDtlAJ.sqlj,** then from the menu, click the **bug** icon to enable the pop-down list and select **Debug,** as shown in Figure 29-24.

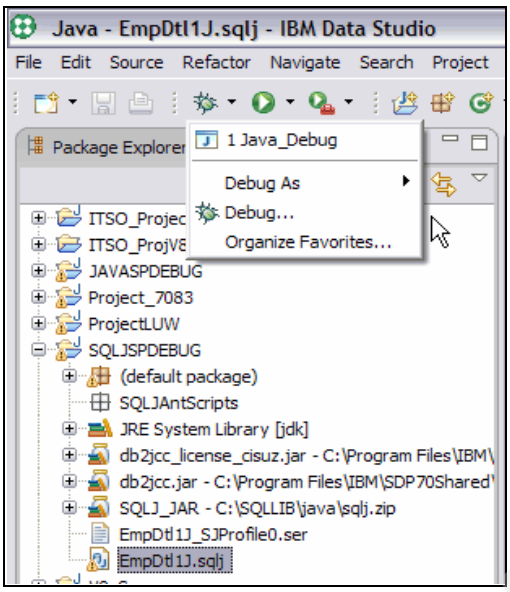

Figure 29-24 Launch the Debug configuration

- Locate **Java Application** in Configurations in the left-hand portion of the window, then click **New launch configuration** as shown in Figure 29-13 on page 794.
- In the Debug Main window, enter the information in Table 29-4. If EmpDtl1J is not the Main class selected, click **Browse** to select this class.

Table 29-4 Debug settings for SQLJ applications

Field	Value
Name	SQLJ_Debug
Project	SQLJSPDebug
Main class	EmpDtl1J

- Click **Apply.**

  The main configuration panel is shown in Figure 29-25 on page 802.

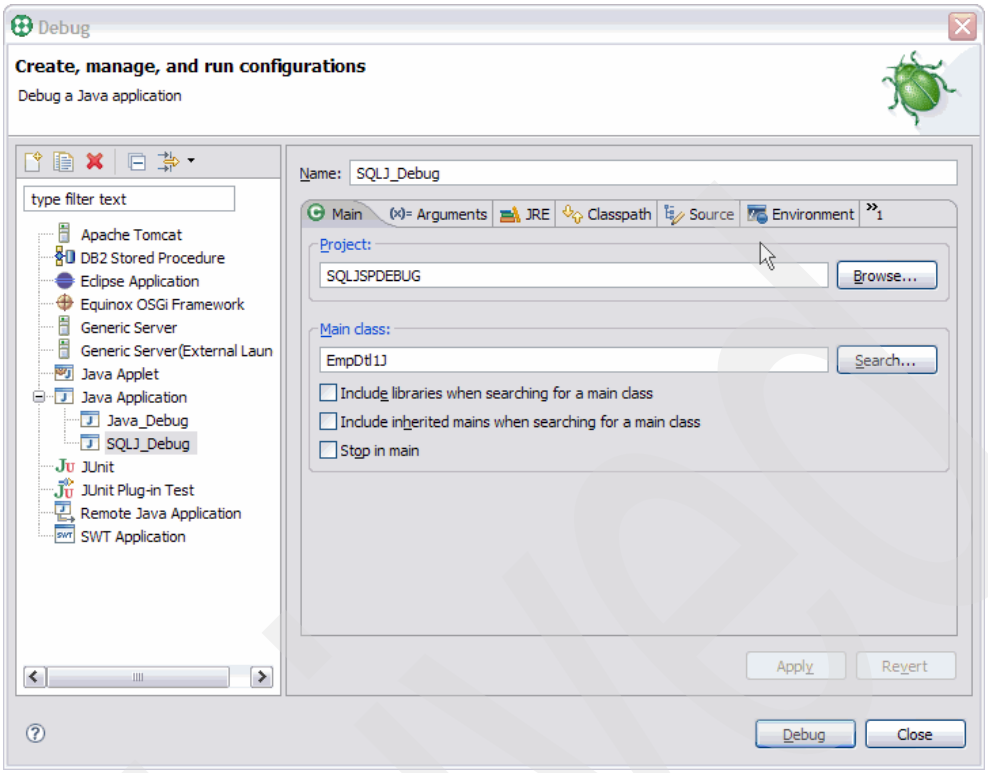

Figure 29-25 Define a new SQLJ Debug configuration

- Our stored procedure requires input parameters to execute. These were removed as described in 29.2.5, “Modify the source code” on page 800. So we now specify the input parameters as input arguments to our SQLJ application.

  Click the **Arguments** tab.
- The Arguments window is where we specify these parameters. Enter `000150` selecting an employee number in the EMP table.
- Click **Debug** to start the debug session.

  You may be asked again to confirm switching from the Java Perspective to the Debug Perspective as shown in Figure 29-26. Click the checkbox **Remember my decision** to suppress this dialog the next time you debug.

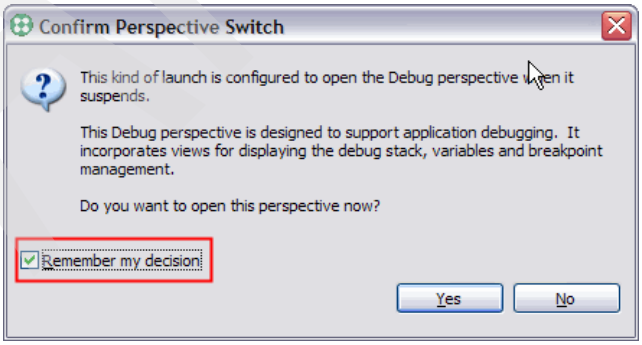

Figure 29-26 Confirm Perspective Switch, Remember my decision

The Debug Perspective is now launched. Once the debug session starts, we can start debugging the SQLJ application, and monitor or change variables, set breakpoints, step through the code, etc., as shown in Figure 29-27.

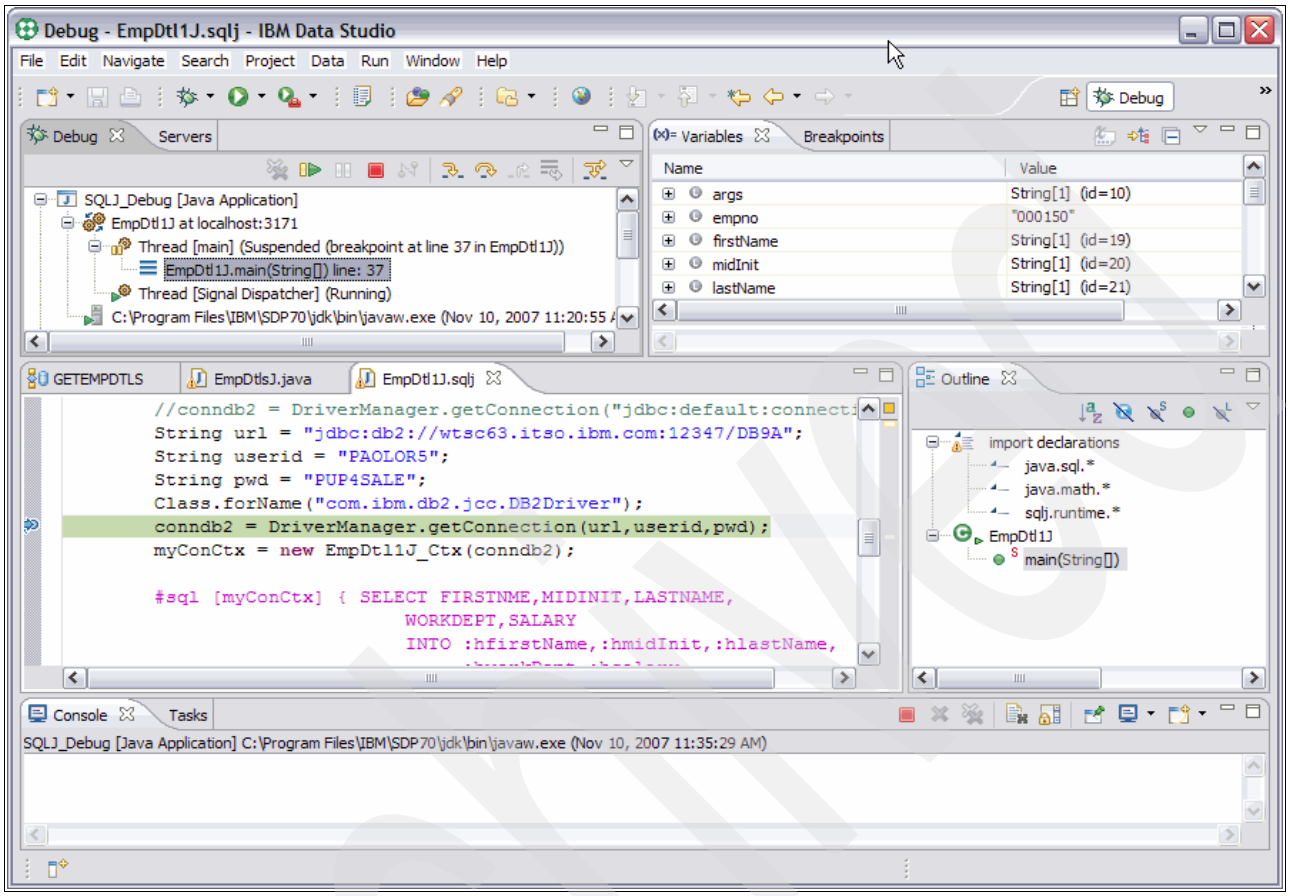

Figure 29-27 Debug Perspective launched for an SQLJ application

Part 7

Appendixes

This part includes the Appendixes defined during this project:

A

Samples for using DB2-supplied stored procedures

In this appendix we provide the source code for invoking several of the DB2-provided stored procedures described in Chapter 24, “DB2-supplied stored procedures” on page 493.

This appendix details the invocation of stored procedures providing the following functions:

- Display DB2 system information with AdminSystemInformation
- Refresh a WLM environment with AdminWLMRefresh
- Issue DB2 commands with AdminDB2Command
- Automate RUNSTATS with AdminUtilityExecution
- Manage data sets with AdminDataSet
- Submit JCL with AdminJob
- Issue USS commands with AdminUNIXCommand
- Issue DSN subcommands with AdminDSNSubcommand
- Task Scheduler Sample Use cases
- Invoking the Common SQL API stored procedures

The structure of the sample programs is very similar across the first seven functions. To familiarize yourself, read through A.1, “Display DB2 system information with AdminSystemInformation” on page 808 where the sample program structure is described in detail. The following six samples focus on calling the DB2-supplied stored procedures and do not repeat this information. The sections about task scheduler and the Common SQL API are very special and should be read in detail to get familiar with this.

A.1 Display DB2 system information with AdminSystemInformation

AdminSystemInformation displays the MVS subsystem name, the fully qualified domain name, the DSNZPARMs, and the licensed utilities of the connected subsystem.

The MVS subsystem name is important if you want to create JCL that will run DB2 commands, DSN subcommands, or DB2 online or offline utilities. While you can find out the subsystem name by querying the DSNZPARM, ADMIN_INFO_SSID is easier to use and does not require any special privileges such as DSNWZP.

The fully qualified domain name of your DB2 server is important if you want to FTP files to your DB2 server. The TCP/IP hostname that you specified in your CATALOG command on the client or in the JDBC URL of your Java application may be a DB2 Connect gateway. While you can find out the fully qualified domain name using the -DIS DDF console command, ADMIN_INFO_HOST is easier to use, and does not require any special privileges such as ADMIN_COMMAND_DB2.

While the DSNZPARMs are primarily of interest to a database or system administrator to view the configuration parameters of a DB2 subsystem, you may need to know certain DSNZPARMs in any application program that you write. For example, if you use dynamic SQL and your database object names need to be enclosed in delimiters, you have to know what the SQLDELI value is. If the SQLDELI value is DEFAULT, you have to use the quotation mark for a delimited identifier like this:

```
CREATE TABLE "MY TABLE" (COL CHAR(1) NOT NULL WITH DEFAULT)
```

If the SQLDELI value is set to the quotation mark, you have to use the apostrophe as the escape character like this:

```
CREATE TABLE ‘MY TABLE’ (COL CHAR(1) NOT NULL WITH DEFAULT)
```

If you are planning to run utilities from a client application or a remote application server, you have to find out which utilities are installed on the connected DB2 subsystem. In our sample application, we use DSNUTILU to run the DIAGNOSE online utility to find out which licensed utilities are installed.

Example A-1 lists the source code for the AdminSystemInformation class where we need to import the required Java packages and classes first.

Example: A-1 AdminSystemInformation class

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminSystemInformation.java
//
// Sample: How to use the DB2 provided system information stored
//         procedures
//
//The user runs the program by issuing:
//java AdminSystemInformation <alias or //server/database> <userid> <password>
//
//The arguments are:
```

```
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************
import java.sql.*;
```

In Example A-2 we define a bitmask for every DB2 utility. We will use these bitmasks later when we display the installed licensed utilities.

Example: A-2 Defining a bitmask for each utility

```
public class AdminSystemInformation
{
  private static final long CATMAINT =   0x0000000000000001L;
  private static final long CHECK =      0x0000000000000002L;
  private static final long COPY =       0x0000000000000004L;
  private static final long DIAGNOSE =   0x0000000000000008L;
  private static final long LISTDEF =    0x0000000000000010L;
  private static final long LOAD =       0x0000000000000020L;
  private static final long MERGECOPY =  0x0000000000000040L;
  private static final long MODIFY =     0x0000000000000080L;
  private static final long OPTIONS =    0x0000000000000100L;
  private static final long QUIESCE =    0x0000000000000200L;
  private static final long REBUILD =    0x0000000000000400L;
  private static final long RECOVER =    0x0000000000000800L;
  private static final long REORG =      0x0000000000001000L;
  private static final long REPAIR =     0x0000000000002000L;
  private static final long REPORT =     0x0000000000004000L;
  private static final long RUNSTATS =   0x0000000000008000L;
  private static final long STOSPACE =   0x0000000000010000L;
  private static final long TEMPLATE =   0x0000000000020000L;
  private static final long UNLOAD =     0x0000000000040000L;
  private static final long COPYTOCOPY = 0x0000000000080000L;
  private static final long EXEC =       0x0000000000100000L;
  private static final long BACKUP =     0x0000000000200000L;
  private static final long RESTORE =    0x0000000000400000L;
```

Most of our sample applications only contain a single main() method and an exception class, which is thrown to indicate a program error rather than a system error such as an SQL error. The main() method checks whether all the required arguments have been passed to the program. In case of an error, a usage message is displayed, as shown in Example A-3.

Example: A-3 Errors on argument verification

```
  public static void main(String[] args)
  {
    Connection con = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminSystemInformation " +
```

```
                    "<alias or //server/database> <userid> <password>");
  System.err.println("where  <alias or //server/database> is " +
                    "DB2 subsystem alias or //server/database " +
                    "for type 4 connectivity");
  System.err.println("       <userid> is user ID to connect as");
  System.err.println("       <password> is password to connect with");
  return;
}
url += args[0];
userid = args[1];
password = args[2];
```

We now load the IBM DB2 Driver for JDBC and SQLJ by invoking the Class.forName method with the following argument: com.ibm.db2.jcc.DB2Driver, as shown in Example A-4. This is a single driver that allows for both type 2 and type 4 JDBC support. An application setting up a connection via a locally registered DB2 subsystem or alias establishes a type 2 connection. Type 4 connectivity is employed when the target server is addressed with the help of server name, port number, and the location name. For more information on using JDBC drivers in your application, see the *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for JAVA,* SC18-9842*.*

A number of exceptions can be thrown here. The most likely problem is that the db2jcc.jar archive that contains the JDBC driver classes is not in your CLASSPATH when you run this application. Now we connect to the database with a URL as specified in the JDBC specification and using db2 as the subprotocol.

Example: A-4 Load and connect with type 2 driver for COM.ibm.db2.jdbc.app.DB2Driver

```
try
{
  int rc = 0;
  String message = null;
  boolean hasResultSet = false;

  // Load the DB2 Universal JDBC Driver
  Class.forName(driver);

  // Connect to database
  con = DriverManager.getConnection(url, userid, password);
  con.setAutoCommit(false);
```

We now prepare a CallableStatement, which is the standard way in JDBC to execute stored procedures. IN parameter values are set using the set methods inherited from PreparedStatement. See Example A-5. The type of all OUT parameters must be registered prior to executing the stored procedure; their values are retrieved after execution through the get methods provided here. ADMIN_INFO_SSID has no IN parameters and three OUT parameters, which we register and then call execute to call the stored procedure.

Example: A-5 Preparing the CallableStatement

```
  // Query SSID
  cs = con.prepareCall("CALL SYSPROC.ADMIN_INFO_SSID(?, ?, ?)");
  cs.registerOutParameter(1, Types.VARCHAR); // SSID
  cs.registerOutParameter(2, Types.INTEGER); // Return code
  cs.registerOutParameter(3, Types.VARCHAR); // Message area

cs.execute();
```

Correct and complete error handling in your applications is very important. Many of the DB2 provided stored procedures follow a similar approach. First you have to check an OUT parameter return code to determine whether the call was successful. If ADMIN_INFO_SSID completes normally, it issues return code 0. It if completes with an error, it issues return code 12. In case of an error we retrieve the error message from the message area and throw an application-defined exception called AdminSystemInformationException with the return code and the error message. If the call was successful, we retrieve the subsystem name, print it, and close the CallableStatement to release associated JDBC and database resources. It is generally good practice to release resources as soon as possible. This is shown in Example A-6.

Example: A-6 Error handling in the procedure

```
      // Obtain the return code
      rc = cs.getInt(2);
      if (rc > 0)
      {
      message = cs.getString(3);
        throw new AdminSystemInformationException(rc, "ADMIN_INFO_SSID execution failed: "
+ message);
      }
      else
      {
        String sSSID = cs.getString(1);
        System.out.println("SSID = " + sSSID);
        cs.close();
      }
```

The call to ADMIN_INFO_HOST to retrieve the fully qualified domain name is shown in Example A-7. This stored procedure has two input parameters that are initialized with the help of the set functions, whereas the actual value of the host name is returned in a result set. If the return code output parameter features a value of 0, the result set is accessed. In the sample the proper result set handling is illustrated, which fetches the TCP/IP hostname for the connected DB2 subsystem.

Example: A-7 Retrieving the domain name

```
      // Query hostname of connected DB2 subsystem
      cs = con.prepareCall("CALL SYSPROC.ADMIN_INFO_HOST(?, ?, ?, ?)");
      cs.setInt(1, 1);                          // Processing option
      cs.setNull(2, Types.VARCHAR);             // DB2 member name
      cs.registerOutParameter(3, Types.INTEGER); // Return code
      cs.registerOutParameter(4, Types.VARCHAR); // Message area
      hasResultSet = cs.execute();

      // Obtain the return code
      rc = cs.getInt(3);
      if (rc > 0)
      {
      message = cs.getString(4);
        throw new AdminSystemInformationException(rc, "ADMIN_INFO_HOST execution failed: "
+ message);
      }
      else
      {
        if (hasResultSet)
        {
          rs = cs.getResultSet();
```

```
System.out.println("----------------------------------------------------------------");
          while (rs.next())
          {
            System.out.println(" DB2 member = " + rs.getString(2) + " Hostname = " +
rs.getString(3).trim());
          }
          rs.close();
        }
        cs.close();
      }
```

We use DSNWZP to retrieve the DSNZPARMs of the connected subsystem. DSNWZP has only one OUT parameter, which we register and then call execute to call the stored procedure. Since DSNWZP does not issue a return code, we can simply retrieve the OUT parameter and tokenize it using the split() method, which is new since Java 1.4. The split() method splits the string around matches of the given regular expression and returns a string array, which we print to the terminal. See Example A-8.

Example: A-8 Calling DSNWZP and handling the output

```
      // Query ZPARM
      cs = con.prepareCall("CALL SYSPROC.DSNWZP(?)");
      cs.registerOutParameter(1, Types.LONGVARCHAR); // ZPARMs
      cs.execute();

      String[] zparms = cs.getString(1).trim().split("[/\n]");

System.out.println("----------------------------------------------------------------");
      for (int i = 0; (i + 7) < zparms.length; i += 7)
      {
        System.out.println("Internal field name       = " + zparms[i]);
        System.out.println("Macro name               = " + zparms[i + 1]);
        System.out.println("Parameter name           = " + zparms[i + 2]);
        System.out.println("Install panel name       = " + zparms[i + 3]);
        System.out.println("Install panel field number = " + zparms[i + 4]);
        System.out.println("Install panel field name   = " + zparms[i + 5]);
        System.out.println("Value                    = " + zparms[i + 6]);
      }
      cs.close();
```

We run the DIAGNOSE online utility through the online utility stored procedure DSNUTILU to determine which licensed utilities are installed on the connected subsystem. This is shown in Example A-9. DSNUTILU does not dynamically allocate data sets. In order to do so, the TEMPLATE utility control statement has to be used. The DIAGNOSE utility does not require any data sets, so the call is very straightforward and thus much simpler than the same call with DSNUTILS. Here only the utility-ID, a utility restart value, as well as the actual utility control statement has to be provided.

Example: A-9 Running DIAGNOSE through DSNUTILU

```
      // Query installed utilities
      String map = null;
      cs = con.prepareCall("CALL SYSPROC.DSNUTILU(?, ?, ?, ?)");
      cs.setString(1, "ADMINSYSINFO");              // Utility ID
      cs.setString(2, "NO");                        // Restart
      cs.setString(3, "DIAGNOSE DISPLAY AVAILABLE"); // Utility statement
      cs.registerOutParameter(4, Types.INTEGER);    // Return code
      hasResultSet = cs.execute();
```

The error handling of stored procedures that return error messages in a result set cursor is different from that of other stored procedures. You should always check if there is a result set and parse it or print it to a log. An error may have occurred while the stored procedure is inserting rows into the result set table and so you do not lose any results. The output from DIAGNOSE DISPLAY AVAILABLE looks similar to Figure A-1.

```
DSNU050I    DSNUGUTC -  DIAGNOSE DISPLAY AVAILABLE
DSNU862I    DSNUDIAG - DISPLAY AVAILABLE UTILITIES.  MAP: 11111111111111111111111000000
-------------------------------------------------------------------------------
!CATMAINT !CHECK    !COPY     !DIAGNOSE !LISTDEF  !LOAD     !MERGECOPY!MODIFY   !

!OPTIONS  !QUIESCE  !REBUILD  !RECOVER  !REORG    !REPAIR   !REPORT   !RUNSTATS !

!STOSPACE !TEMPLATE !UNLOAD   !COPYTOCOP!EXEC     !BACKUP   !RESTORE  !         !

!         !         !         !         !         !         !         !         !
-------------------------------------------------------------------------------
DSNU860I    DSNUGUTC - DIAGNOSE UTILITY COMPLETE
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

Figure A-1 Output from DIAGNOSE DISPLAY AVAILABLE

The map shows a 1 for each installed utility, and each position represents a specific utility. We parse the output of message number DSNU8621 and store the map string for later processing, as shown in Example A-10.

Example: A-10 Parsing DSNU8621

```
      if (hasResultSet)
      {
        rs = cs.getResultSet();

System.out.println("----------------------------------------------------------------");
        while (rs.next())
        {
          String line = rs.getString(2);
          if (line.indexOf("DSNU862I") != -1)
            map = line.substring(line.lastIndexOf(':') + 1).trim();
          System.out.println(line);
        }
        rs.close();
      }
```

After we have processed the result set, we check the return code as usual. For more information on the use of DSNUTILU and the DIAGNOSE utility, refer to *DB2 Version 9.1 for z/OS Utility Guide and Reference,* SC18-9855. If the return code indicates that DIAGNOSE has completed normally, and we found a map string in the output, we string the bitmap values for each utility together and then print it to the console. This is a good way to save the information on which utilities are installed. See Example A-11.

Example: A-11 Displaying the installed utilities

```
      rc = cs.getInt(4);
      if (map == null || rc > 4)
      {
        throw new AdminSystemInformationException(rc, "DSNUTILU execution failed.");
      }
      else
      {
        char[] mapArray = map.toCharArray();
        long base = 1;
        long licensedUtilities = 0;

        // Build utilities bitmap
        for (int i = 0; i < mapArray.length; i++)
        {
          if (mapArray[i] == '1')
            licensedUtilities += base;
          base = base * 2;
        }

        System.out.println("CATMAINT = " + ((CATMAINT & licensedUtilities) != 0));
        System.out.println("CHECK = " + ((CHECK & licensedUtilities) != 0));
        System.out.println("COPY = " + ((COPY & licensedUtilities) != 0));
        System.out.println("DIAGNOSE = " + ((DIAGNOSE & licensedUtilities) != 0));
        System.out.println("LISTDEF = " + ((LISTDEF & licensedUtilities) != 0));
        System.out.println("LOAD = " + ((LOAD & licensedUtilities) != 0));
        System.out.println("MERGECOPY = " + ((MERGECOPY & licensedUtilities) != 0));
        System.out.println("MODIFY = " + ((MODIFY & licensedUtilities) != 0));
        System.out.println("OPTIONS = " + ((OPTIONS & licensedUtilities) != 0));
        System.out.println("QUIESCE = " + ((QUIESCE & licensedUtilities) != 0));
        System.out.println("REBUILD = " + ((REBUILD & licensedUtilities) != 0));
        System.out.println("RECOVER = " + ((RECOVER & licensedUtilities) != 0));
        System.out.println("REORG = " + ((REORG & licensedUtilities) != 0));
        System.out.println("REPAIR = " + ((REPAIR & licensedUtilities) != 0));
        System.out.println("REPORT = " + ((REPORT & licensedUtilities) != 0));
        System.out.println("RUNSTATS = " + ((RUNSTATS & licensedUtilities) != 0));
        System.out.println("STOSPACE = " + ((STOSPACE & licensedUtilities) != 0));
        System.out.println("TEMPLATE = " + ((TEMPLATE & licensedUtilities) != 0));
        System.out.println("UNLOAD = " + ((UNLOAD & licensedUtilities) != 0));
        System.out.println("COPYTOCOPY = " + ((COPYTOCOPY & licensedUtilities) != 0));
        System.out.println("EXEC = " + ((EXEC & licensedUtilities) != 0));
        System.out.println("BACKUP = " + ((BACKUP & licensedUtilities) != 0));
        System.out.println("RESTORE = " + ((RESTORE & licensedUtilities) != 0));
        cs.close();
      }
    }
```

In order to provide meaningful error messages, we catch our application-defined exceptions AdminSystemInformationException and SQLException. This is good practice so that we do not lose any information such as the SQL code if we just caught an exception. Coding a *finally block* is good coding practice to release JDBC and database resources and to disconnect from the database. The final block is always executed, even when an exception is thrown. See Example A-12.

Example: A-12 The finally block code

```
    catch (AdminSystemInformationException asie)
    {
```

```
      System.err.println("Program error: rc=" + asie.getRC() + " message=" +
asie.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());
    }
    catch (Exception e)
    {
      System.err.println("Error: message=" + e.getMessage());
    }
    finally
    {
      // Release resources and disconnect
      try
      {
        rs.close();
      } catch (Exception e)
      {
      }

      try
      {
        cs.close();
      } catch (Exception e)
      {
      }

      try
      {
        con.close();
      } catch (Exception e)
      {
      }
    }
  }
}
```

Our application-defined exception class extends Exception by allowing to set and get a return code, which is typically the stored procedure return code. See Example A-13.

Example: A-13 Setting and getting the return code

```
class AdminSystemInformationException extends Exception
{
  private int rc;

  AdminSystemInformationException(int rc, String message)
  {
    super(message);
    this.rc = rc;
```

```
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile AdminSystemInformation and enter the following command to execute it:

```
java AdminSystemInformation DBALIAS USERID PASSWORD
```

You should get the response shown in Figure A-14.

Example: A-14 Output from AdminSystemInformation

```
SSID = V91A
-------------------------------------------------------------
 DB2 member =          Hostname = w25ec051.svl.ibm.com
-------------------------------------------------------------
Internal field name        = SYSPAUDT
Macro name                 = DSN6SYSP
Parameter name             = AUDITST
Install panel name         = DSNTIPN
Install panel field number =  1
Install panel field name   = AUDIT TRACE
Value                      = 00000000000000000000000000000000
...
ODSNU050I    DSNUGUTC -  DIAGNOSE DISPLAY AVAILABLE
 DSNU862I    DSNUDIAG - DISPLAY AVAILABLE UTILITIES.  MAP: 11111111111111111111110000000000
 ------------------------------------------------------------------------------
 |CATMAINT |CHECK    |COPY     |DIAGNOSE |LISTDEF  |LOAD     |MERGECOPY|MODIFY   |
 |OPTIONS  |QUIESCE  |REBUILD  |RECOVER  |REORG    |REPAIR   |REPORT   |RUNSTATS |
 |STOSPACE |TEMPLATE |UNLOAD   |COPYTOCOP|EXEC     |BACKUP   |RESTORE  |         |
 |         |         |         |         |         |         |         |         |
 ------------------------------------------------------------------------------
 DSNU860I    DSNUGUTC - DIAGNOSE UTILITY COMPLETE
 DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
CATMAINT = true
CHECK = true
COPY = true
DIAGNOSE = true
LISTDEF = true
LOAD = true
MERGECOPY = true
MODIFY = true
OPTIONS = true
QUIESCE = true
REBUILD = true
RECOVER = true
REORG = true
REPAIR = true
REPORT = true
RUNSTATS = true
STOSPACE = true
TEMPLATE = true
UNLOAD = true
COPYTOCOPY = true
EXEC = true
BACKUP = true
```

```
RESTORE = true
```

A.2 Refresh a WLM environment with AdminWLMRefresh

AdminWLMRefresh refreshes a WLM application environment, which is required after redeploying a modified stored procedure or after you have changed the WLM JCL procedure. The *DB2 Version 9.1 for z/OS Application Programming and SQL Guide*, SC18-9841 describes WLM_REFRESH and the *DB2 Version 9.1 for z/OS Installation Guide,* GC18-9846 contains detailed information on how to use it as well as the required SAF resource profiles that need to be defined. WLM_REFRESH has two IN parameters, the WLM application environment name and the name of the associated subsystem. The parameters used in the following example require that an SAF resource profile DSN.WLM_REFRESH.WLMENVR of the class DSNR has been created and the caller has READ access to it. Ensure that the class DSNR is active.

Example A-15 lists the source code for the AdminWLMRefresh class.

Example: A-15 AdminWLMRefresh source code

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminWLMRefresh.java
//
// Sample: How to use the DB2 provided procedure WLM_REFRESH
//
//The user runs the program by issuing:
//java AdminWLMRefresh <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************
import java.sql.*;

public class AdminWLMRefresh
{
  public static void main(String[] args)
  {
    Connection con = null;
    CallableStatement cs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminWLMRefresh <alias or //server/database> <userid>
<password>");
```

```
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
    {
      int rc = 0;
      String message = null;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      // Refresh WLM environment
      cs = con.prepareCall("CALL SYSPROC.WLM_REFRESH(?, ?, ?, ?)");
      cs.setString(1, "WLMENV1");              // WLM environment
      cs.setString(2, "DSN");                  // SSID
      cs.registerOutParameter(3, Types.VARCHAR); // Message area
      cs.registerOutParameter(4, Types.INTEGER); // Return code
      cs.execute();

    // Obtain the return code
      message = cs.getString(3);
      rc = cs.getInt(4);
      if (rc > 0)
        throw new AdminWLMRefreshException(rc, "WLM_REFRESH execution failed: " + message);
      else
      {
        System.out.println("WLM refresh successful:" + message);
        cs.close();
      }
    }

    catch (AdminWLMRefreshException awre)
    {
      System.err.println("Program error: rc=" + awre.getRC() + " message=" +
awre.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());
```

```
    }

    finally
    {
      // Release resources and disconnect
      try
      {
        cs.close();
      } catch (Exception e)
      {
      }

      try
      {
        con.close();
      } catch (Exception e)
      {
      }
    }
  }
}

class AdminWLMRefreshException extends Exception
{
  private int rc;

  AdminWLMRefreshException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile AdminWLMRefresh and enter the following command to execute it:

```
java AdminWLMRefresh DBALIAS USERID PASSWORD
```

You should get the response:

```
WLM refresh successful:DSNT540I  WLMENV1 WAS REFRESHED BY PAOLOR3 USING AUTHORITY FROM
SQL ID PAOLOR3
```

A.3 Issue DB2 commands with AdminDB2Command

AdminDB2Command shows how to use ADMIN_COMMAND_DB2 to issue the following DB2 commands:

```
-DISPLAY ARCHIVE
-DISPLAY BUFFERPOOL
-DISPLAY DATABASE
-DISPLAY THREAD
-DISPLAY UTILITY
-DISPLAY DDF DETAIL
-DISPLAY GROUP DETAIL
```

Except for the -DISPLAY ARCHIVE command, the corresponding parse type is specified. The sample program also shows how to retrieve meta information like the DB2 major version number. This might be useful to programmatically react upon program execution against different versions of the data server.

Example A-16 shows the source code for the AdminDB2Command class.

Example: A-16 AdminDB2Command class

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminDB2Command.java
//
// Sample: How to use the DB2 provided stored procedure to issue DB2
//         commands.
//
//The user runs the program by issuing:
//java AdminDB2Command <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************
import java.sql.*;
import java.util.*;

public class AdminDB2Command
{
  public static final String PARSE_NO = "NO";
  public static final String PARSE_BP = "BP";
  public static final String PARSE_DB = "DB";
  public static final String PARSE_IX = "IX";
  public static final String PARSE_TS = "TS";
  public static final String PARSE_THD = "THD";
  public static final String PARSE_UT = "UT";
  public static final String PARSE_DDF = "DDF";
  public static final String PARSE_GRP = "GRP";

  public static void main(String args[])
  {
    Connection con = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;
    int db2release = 0;

    // Parse arguments
    if (args.length != 3)
    {
```

```
      System.err.println("Usage: AdminDB2Command <alias or //server/database> <userid>
<password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
    {
      int rc = 0;
      String message = null;
      boolean hasResultSet = false;
      ArrayList<String> messageText = new ArrayList<String>();
      String commandArea = null;
      String parseType = null;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      DatabaseMetaData databaseMetaData = con.getMetaData();
      db2release = databaseMetaData.getDatabaseMajorVersion();
      // Display the DB2 major version this value might be useful
      // when temporary tables differ between version numbers.
      // With the help of this version the right one can be chosen
      // for the output.
      System.out.println("DB2 major version " + db2release);
      con.setAutoCommit(false);

      // Issue DB2 commands
      cs = con.prepareCall("CALL SYSPROC.ADMIN_COMMAND_DB2(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?)");
      cs.registerOutParameter(5,  Types.INTEGER); // Number of commands executed
      cs.registerOutParameter(6,  Types.INTEGER); // IFI return code
      cs.registerOutParameter(7,  Types.INTEGER); // IFI reason code
      cs.registerOutParameter(8,  Types.INTEGER); // IFI excess bytes
      cs.registerOutParameter(9,  Types.INTEGER); // IFI reason code (data sharing)
      cs.registerOutParameter(10, Types.INTEGER); // IFI excess bytes (data sharing)
      cs.registerOutParameter(11, Types.INTEGER); // Return code
      cs.registerOutParameter(12, Types.VARCHAR); // Message area

      commandArea = "-DISPLAY ARCHIVE";
      parseType = PARSE_NO;
      cs.setString(1, commandArea);       // DB2 commands
      cs.setInt(2, commandArea.length()); // Commands area length
      cs.setString(3, parseType);         // Parse type
      cs.setNull(4, Types.VARCHAR);       // DB2 member name
      rs = executeCommand(cs, messageText);
      parseResponse(parseType, rs, messageText);
      if (rs != null)
        rs.close();

      commandArea = "-DISPLAY BUFFERPOOL (BP0)";
      parseType = PARSE_BP;
```

```
cs.setString(1, commandArea);
cs.setInt(2, commandArea.length());
cs.setString(3, parseType);
cs.setNull(4, Types.VARCHAR);
rs = executeCommand(cs, messageText);
parseResponse(parseType, rs, messageText);
if (rs != null)
  rs.close();

commandArea = "-DISPLAY DATABASE (DSNDB06) LIMIT(500)";
parseType = PARSE_DB;
cs.setString(1, commandArea);
cs.setInt(2, commandArea.length());
cs.setString(3, parseType);
cs.setNull(4, Types.VARCHAR);
rs = executeCommand(cs, messageText);
parseResponse(parseType, rs, messageText);
if (rs != null)
  rs.close();

commandArea = "-DISPLAY THREAD (*)";
parseType = PARSE_THD;
cs.setString(1, commandArea);
cs.setInt(2, commandArea.length());
cs.setString(3, parseType);
cs.setNull(4, Types.VARCHAR);
rs = executeCommand(cs, messageText);
parseResponse(parseType, rs, messageText);
if (rs != null)
  rs.close();

commandArea = "-DISPLAY UTILITY (*)";
parseType = PARSE_UT;
cs.setString(1, commandArea);
cs.setInt(2, commandArea.length());
cs.setString(3, parseType);
cs.setNull(4, Types.VARCHAR);
rs = executeCommand(cs, messageText);
parseResponse(parseType, rs, messageText);
if (rs != null)
  rs.close();

commandArea = "-DISPLAY DDF DETAIL";
parseType = PARSE_DDF;
cs.setString(1, commandArea);
cs.setInt(2, commandArea.length());
cs.setString(3, parseType);
cs.setNull(4, Types.VARCHAR);
rs = executeCommand(cs, messageText);
parseResponse(parseType, rs, messageText);
if (rs != null)
 rs.close();

commandArea = "-DISPLAY GROUP DETAIL";
parseType = PARSE_GRP;
cs.setString(1, commandArea);
cs.setInt(2, commandArea.length());
cs.setString(3, parseType);
cs.setNull(4, Types.VARCHAR);
rs = executeCommand(cs, messageText);
```

```
      parseResponse(parseType, rs, messageText);
      if (rs != null)
        rs.close();

      cs.close();
    }
    catch (AdminDB2CommandException adce)
    {
      System.err.println("Program error: message=" + adce.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());
    }
    catch (Exception e)
    {
      System.out.println("Error: message=" + e.getMessage());
    }

    finally
    {
      // Release resources and disconnect
      try { rs.close(); }
      catch(Exception e) {}

      try { cs.close(); }
      catch(Exception e) {}

      try { con.close(); }
      catch(Exception e) {}
    }
  }

  private static ResultSet executeCommand(CallableStatement cs, ArrayList<String>
messageText) throws Exception
  {
    int rc = 0;
    ResultSet rs = null;
    boolean hasErrorMessage = false;

    messageText.clear();
    boolean hasResultSet = cs.execute();
    if (hasResultSet)
    {
      rs = cs.getResultSet();

      // Save command execution messages
      while (rs.next())
        messageText.add((String)rs.getString(2).trim());

      if (cs.getMoreResults())
```

```
        rs = cs.getResultSet();
      else
        rs = null;
    }

    int execute_count = cs.getInt(5);
    int IFIReturnCode = cs.getInt(6);
    int IFIReasonCode = cs.getInt(7);
    int IFIExcessBytes = cs.getInt(8);
    int IFIDSReasonCode = cs.getInt(9);
    int IFIDSExcessBytes = cs.getInt(10);
    int ReturnCode = cs.getInt(11);
    String SQLMessage = cs.getString(12);

    if (ReturnCode > 0)
    {
      throw new AdminDB2CommandException(rc, "ADMIN_COMMAND_DB2 execution failed: " +
SQLMessage);
    }
    else if (IFIReturnCode != 0)
    {
      System.out.println("IFI error encountered:");
      System.out.println("IFI return code               = " + IFIReturnCode);
      System.out.println("IFI reason code               = " + IFIReasonCode);
      System.out.println("IFI excess bytes              = " + IFIExcessBytes);
      System.out.println("IFI data sharing reason code  = " + IFIDSReasonCode);
      System.out.println("IFI data sharing excess bytes = " + IFIDSExcessBytes);
    }

    return rs;
  }

  private static void parseResponse(String parseType, ResultSet rs, ArrayList messageText)
throws SQLException
  {
    if (rs != null)
    {
      while (rs.next())
      {
        if (parseType.equals(PARSE_BP))
        {
            System.out.println("BPNAME    = " + rs.getString(2).trim());
            System.out.println("VPSIZE    = " + rs.getInt(3));
            System.out.println("VPSEQT    = " + rs.getInt(4));
            System.out.println("VPPSEQT   = " + rs.getInt(5));
            System.out.println("VPXPSEQT  = " + rs.getInt(6));
            System.out.println("DWQT      = " + rs.getInt(7));
            System.out.println("PCT VDWQT = " + rs.getInt(8));
            System.out.println("ABS VDWQT = " + rs.getInt(9));
            System.out.println("PGSTEAL   = " + rs.getString(10).trim());
            System.out.println("ID        = " + rs.getInt(11));
            System.out.println("USE COUNT = " + rs.getInt(12));
            System.out.println("PGFIX     = " + rs.getString(13).trim());
        }
        else if (parseType.equals(PARSE_DB))
        {
          System.out.println("DBNAME   = " + rs.getString(2).trim());
          System.out.println("SPACENAM = " + rs.getString(3).trim());
          System.out.println("TYPE     = " + rs.getString(4).trim());
          System.out.println("PART     = " + rs.getShort(5));
```

```
            System.out.println("STATUS   = " + rs.getString(6).trim());
          }
          else if (parseType.equals(PARSE_THD))
          {
            System.out.println("TYPE        = " + rs.getInt(2));
            System.out.println("NAME        = " + rs.getString(3).trim());
            System.out.println("STATUS      = " + rs.getString(4).trim());
            System.out.println("ACTIVE      = " + rs.getString(5).trim());
            System.out.println("REQ         = " + rs.getString(6).trim());
            System.out.println("ID          = " + rs.getString(7).trim());
            System.out.println("AUTHID      = " + rs.getString(8).trim());
            System.out.println("PLAN        = " + rs.getString(9).trim());
            System.out.println("ASID        = " + rs.getString(10).trim());
            System.out.println("TOKEN       = " + rs.getString(11).trim());
            System.out.println("COORDINATOR = " + rs.getString(12).trim());
            System.out.println("RESET       = " + rs.getString(13).trim());
            System.out.println("URID        = " + rs.getString(14).trim());
            System.out.println("LUWID       = " + rs.getString(15).trim());
            System.out.println("WORKSTATION = " + rs.getString(16).trim());
            System.out.println("USERID      = " + rs.getString(17).trim());
            System.out.println("APPLICATION = " + rs.getString(18).trim());
            System.out.println("ACCOUNTING  = " + rs.getString(19).trim());
            System.out.println("LOCATION    = " + rs.getString(20).trim());
            System.out.println("DETAIL      = " + rs.getString(21).replace('\0',
'\n').trim());
          }
          else if (parseType.equals(PARSE_UT))
          {
            System.out.println("CSECT     = " + rs.getString(2).trim());
            System.out.println("USER      = " + rs.getString(3).trim());
            System.out.println("MEMBER    = " + rs.getString(4).trim());
            System.out.println("UTILID    = " + rs.getString(5).trim());
            System.out.println("STATEMENT = " + rs.getInt(6));
            System.out.println("UTILITY   = " + rs.getString(7).trim());
            System.out.println("PHASE     = " + rs.getString(8).trim());
            System.out.println("COUNT     = " + rs.getInt(9));
            System.out.println("STATUS    = " + rs.getString(10).trim());
            System.out.println("DETAIL    = " + rs.getString(11).replace('\0', '\n').trim());
            System.out.println("NUM OBJ   = " + rs.getInt(12));
            System.out.println("LAST OBJ  = " + rs.getInt(13));
          }
          else if (parseType.equals(PARSE_DDF))
          {
            System.out.println("STATUS       = " + rs.getString(2).trim());
            System.out.println("LOCATION     = " + rs.getString(3).trim());
            System.out.println("LUNAME       = " + rs.getString(4).trim());
            System.out.println("GENERICLU    = " + rs.getString(5).trim());
            System.out.println("IPV4ADDR     = " + rs.getString(6).trim());
            System.out.println("IPV6ADDR     = " + rs.getString(7).trim());
            System.out.println("TCPPORT      = " + rs.getInt(8));
            System.out.println("RESPORT      = " + rs.getInt(9));
            System.out.println("SQL DOMAIN   = " + rs.getString(10).trim());
            System.out.println("RSYNC DOMAIN = " + rs.getString(11).trim());
          }
          else if (parseType.equals(PARSE_GRP))
          {
            System.out.println("DB2 MEMBER  = " + rs.getString(2).trim());
            System.out.println("ID          = " + rs.getInt(3));
            System.out.println("SUBSYS      = " + rs.getString(4).trim());
            System.out.println("CMDPREF     = " + rs.getString(5).trim());
```

```
          System.out.println("STATUS      = " + rs.getString(6).trim());
          System.out.println("DB2 LVL     = " + rs.getString(7).trim());
          System.out.println("SYSTEM NAME = " + rs.getString(8).trim());
          System.out.println("IRLM SUBSYS = " + rs.getString(9).trim());
          System.out.println("IRLMPROC    = " + rs.getString(10).trim());
        }
      }
    }

    if (messageText != null)
    {
      // Print informational message text
      for (int i = 0; i < messageText.size(); i++)
      {
        if (messageText.get(i) != null)
          System.out.println(messageText.get(i));
      }
    }
  }
}

class AdminDB2CommandException extends Exception
{

  AdminDB2CommandException(int rc, String message)
  {
    super(message);
  }
}
```

Compile AdminDB2Command.java and enter the following command to execute it:

```
java AdminDB2Command DBALIAS USERID PASSWORD
```

You should receive the response shown in Example A-17.

Example: A-17 Response to AdminDB2Command

```
DSNJ322I  = DISPLAY ARCHIVE REPORT FOLLOWS-
COUNT                  TIME
(TAPE UNITS)           (MIN,SEC)
DSNZPARM          3                    0,00
CURRENT           3                    0,00
================================
ADDR STATUS CORR-ID  VOLSER DATASET_NAME
NO ARCHIVE READ ACTIVITY
END OF DISPLAY ARCHIVE REPORT.
DSN9022I  = DSNJC001 '-DISPLAY ARCHIVE' NORMAL COMPLETION
BPNAME    = BP0
VPSIZE    = 2000
VPSEQT    = 80
VPPSEQT   = 50
VPXPSEQT  = 0
DWQT      = 85
PCT VDWQT = 80
ABS VDWQT = 0
PGSTEAL   = LRU
ID        = 0
USE COUNT = 25
PGFIX     = NO
```

```
DBNAME   = DSNDB06
SPACENAM =
TYPE     = DB
PART     = 0
STATUS   = RW
TYPE        = 1
NAME        = SERVER
STATUS      = SP
ACTIVE      = *
REQ         = 38
ID          = db2jccmain
AUTHID      = SYSADM
PLAN        = DISTSERV
ASID        = 0023
TOKEN       = 23
COORDINATOR =
RESET       =
URID        =
LUWID       =
WORKSTATION = BL3TFZVB
USERID      = sysadm
APPLICATION = db2jccmain
ACCOUNTING  =
LOCATION    = ::FFFF:9.152.229.57
DETAIL      = V429 CALLING PROCEDURE=SYSPROC.ADMIN_COMMAND_DB2,
      PROC=V91AWLM1, ASID=0021, WLM_ENV=WLMENV1
 V445-G91EBDD5.G1BE.C1A8A703DE26=23 ACCESSING DATA FOR
  ::FFFF:9.152.229.57
DSNU112I  ) DSNUGDIS - NO AUTHORIZED UTILITY FOUND FOR UTILID = *
STATUS       = STARTD
LOCATION     = STLEC1
LUNAME       = USIBMSY.SYEC1DB2
GENERICLU    = -NONE
IPV4ADDR     = ::9.30.189.213
IPV6ADDR     =
TCPPORT      = 446
RESPORT      = 5001
SQL DOMAIN   = w25ec213.svl.ibm.com
RSYNC DOMAIN = w25ec213.svl.ibm.com
DSNLO90I DT=A  CONDBAT=     64 MDBAT=   64
DSNLO92I ADBAT=    1 QUEDBAT=      0 INADBAT=      0 CONQUED=      0
DSNLO93I DSCDBAT=      0 INACONN=      0
DB2 MEMBER  = ........
ID          = 0
SUBSYS      = V91A
CMDPREF     = )
STATUS      = ACTIVE
DB2 LVL     = 910
SYSTEM NAME = ZS17PD
IRLM SUBSYS = PR21
IRLMPROC    = PRLMPR21
*** BEGIN DISPLAY OF GROUP(........) GROUP LEVEL(...) MODE(N )
PROTOCOL LEVEL(3)  GROUP ATTACH NAME(....)
```

A.4 Automate RUNSTATS with AdminUtilityExecution

AdminUtilityExecution runs RUNSTATS by exception. First, it calls DSNACCOX to list all table spaces that require RUNSTATS to be run, then it steps through the recommendation result

set and eliminates all table spaces belonging to a temporary or workfile database. It also eliminates DB2 Directory table spaces and orphaned or restricted table spaces. After that it calls the parallel utility scheduler ADMIN_UTL_SCHEDULE to execute RUNSTATS in parallel.

Example A-18 shows the source code for invoking RUNSTATS on the recommended table spaces.

Example: A-18 AdminUtilityExecution Invoking RUNSTATS

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
//Source file name: AdminUtilityExecution.java
//
//Sample: How to use the DB2 provided stored procedures DSNACCOX and
//        ADMIN_UTL_SCHEDULE
//
//The user runs the program by issuing:
//java AdminUtilityExecution <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************

import java.sql.*;

public class AdminUtilityExecution
{
  public static void main(String args[])
  {
    Connection con = null;
    PreparedStatement ps = null;
    PreparedStatement ps2 = null;
    PreparedStatement psDBType = null;
    PreparedStatement psTSType = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    ResultSet rsDBType = null;
    ResultSet rsTSType = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminUtilityExecution <alias or //server/database>
<userid> <password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
```

```
  return;
}

url += args[0];
userid = args[1];
password = args[2];

try
{
  String QueryType = "RUNSTATS";
  String ObjectType = "TS";
  String ICType = "B";
  String CatlgSchema = "SYSIBM";
  String LocalSchema = "DSNACC";
  int ChkLvl = 1 + 2 + 4 + 8;
  String Criteria = "";
  String Unused = "";
  int CRUpdatedPagesPct = 20;
  int CRUpdatedPagesAbs = 1000;
  int CRChangesPct = 10;
  int CRDaySncLastCopy = 7;
  int ICRUpdatedPagesPct = 1;
  int ICRUpdatedPagesAbs = 300;
  int ICRChangesPct = 1;
  int CRIndexSize = 1;
  int RRTInsertsPct = 20;
  int RRTInsertsAbs = 1000;
  int RRTDeletesPct = 20;
  int RRTDeletesAbs = 1000;
  int RRTUnclustInsPct = 10;
  int RRTDisorgLOBPct = 10;
  int RRTDataSpaceRat = 10;
  int RRTMassDelLimit = 0;
  int RRTIndRefLimit = 10;
  int RRIInsertsPct = 20;
  int RRIInsertsAbs = 500;
  int RRIDeletesPct = 20;
  int RRIDeletesAbs = 500;
  int RRIAppendInsertPct = 10;
  int RRIPseudoDeletePct = 10;
  int RRIMassDelLimit = 0;
  int RRILeafLimit = 10;
  int RRINumLevelsLimit = 0;
  int SRTInsDelUpdPct = 15;
  int SRTInsDelUpdAbs = 5;
  int SRTMassDelLimit = 0;
  int SRIInsDelUpdPct = 15;
  int SRIInsDelUpdAbs = 5;
  int SRIMassDelLimit = 0;
  int ExtentLimit = 2;
  String lastStatement = null;
  int IFCARetCode = 0;
  int IFCAResCode = 0;
  int XSBytes = 0;
  int rc = 0;
  String resultMessage = null;
  String message = null;
  boolean hasResultSet = false;

  // Load the DB2 Universal JDBC Driver
```

```
Class.forName(driver);

// Connect to the database
con = DriverManager.getConnection(url, userid, password);
con.setAutoCommit(false);

// Get RUNSTATS recommendations
cs = con.prepareCall("CALL SYSPROC.DSNACCOX( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, " +
                                            "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, " +
                                            "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, " +
                                            "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, " +
                                            "?, ?, ?, ?, ?, ?, ?");
cs.setString(1, QueryType);
cs.setString(2, ObjectType);
cs.setString(3, ICType);
cs.setString(4, CatlgSchema);
cs.setString(5, LocalSchema);
cs.setInt(6, ChkLvl);
cs.setString(7, Criteria);
cs.setString(8, Unused);
cs.setInt(9, CRUpdatedPagesPct);
cs.setInt(10, CRUpdatedPagesAbs);
cs.setInt(11, CRChangesPct);
cs.setInt(12, CRDaySncLastCopy);
cs.setInt(13, ICRUpdatedPagesPct);
cs.setInt(14, ICRUpdatedPagesAbs);
cs.setInt(15, ICRChangesPct);
cs.setInt(16, CRIndexSize);
cs.setInt(17, RRTInsertsPct);
cs.setInt(18, RRTInsertsAbs);
cs.setInt(19, RRTDeletesPct);
cs.setInt(20, RRTDeletesAbs);
cs.setInt(21, RRTUnclustInsPct);
cs.setInt(22, RRTDisorgLOBPct);
cs.setInt(23, RRTDataSpaceRat);
cs.setInt(24, RRTMassDelLimit);
cs.setInt(25, RRTIndRefLimit);
cs.setInt(26, RRIInsertsPct);
cs.setInt(27, RRIInsertsAbs);
cs.setInt(28, RRIDeletesPct);
cs.setInt(29, RRIDeletesAbs);
cs.setInt(30, RRIAppendInsertPct);
cs.setInt(31, RRIPseudoDeletePct);
cs.setInt(32, RRIMassDelLimit);
cs.setInt(33, RRILeafLimit);
cs.setInt(34, RRINumLevelsLimit);
cs.setInt(35, SRTInsDelUpdPct);
cs.setInt(36, SRTInsDelUpdAbs);
cs.setInt(37, SRTMassDelLimit);
cs.setInt(38, SRIInsDelUpdPct);
cs.setInt(39, SRIInsDelUpdAbs);
cs.setInt(40, SRIMassDelLimit);
cs.setInt(41, ExtentLimit);
cs.registerOutParameter(42, Types.VARCHAR); // Last statement
cs.registerOutParameter(43, Types.INTEGER); // Return code
cs.registerOutParameter(44, Types.VARCHAR); // Message area
cs.registerOutParameter(45, Types.INTEGER); // IFI return code
cs.registerOutParameter(46, Types.INTEGER); // IFI reason code
cs.registerOutParameter(47, Types.INTEGER); // IFI excess bytes
hasResultSet = cs.execute();
```

```
      con.commit();

      resultMessage = cs.getString(44);
      IFCARetCode = cs.getInt(45);
      IFCAResCode = cs.getInt(46);
      XSBytes = cs.getInt(47);

      // Make sure the return code is valid
      rc = cs.getInt(43);
      if (cs.wasNull())
        throw new AdminUtilityExecutionException(rc, "DSNACCOX execution failed: " +
resultMessage);

      if (rc > 4)
      {
        switch (rc)
        {
          case 8:
            message = "DSNACCOX execution failed: " + resultMessage;
            break;

          case 12:
            lastStatement = cs.getString(42);
            message = "DSNACCOX execution failed: " + resultMessage + " last statement: " +
lastStatement;
            break;

          case 14:
            message = "DSNACCOX cannot access real-time statistics tables: " +
resultMessage;
            break;

          case 15:
            message = "DSNACCOX encountered problem with declared temporary tables: " +
resultMessage;
            break;

          case 16:
            message = "DSNACCOX was unable to define declared temporary tables: " +
resultMessage;
            break;
        }

        throw new AdminUtilityExecutionException(rc, message);
      }

      // Print DSNACCOX message
      System.out.println("DSNACCOX message: " + resultMessage);

      if (hasResultSet)
      {
        rs = cs.getResultSet();
        while (rs.next())
          System.out.println("IFI message: " + rs.getString(2));
        rs.close();

        // Prepare the statements for the ADMIN_UTL_SCHEDULE parameter tables
        ps = con.prepareStatement("INSERT INTO SYSIBM.UTILITY_OBJ VALUES (?, ?, ?, ?, ?, ?,
?, ?)");
```

```
        // Insert the utility commands
        ps2 = con.prepareStatement("INSERT INTO SYSIBM.UTILITY_STMT VALUES (?, ?, ?)");
        ps2.setInt(1, 0);
        ps2.setInt(2, 0);
        ps2.setString(3, "RUNSTATS TABLESPACE &OBJECT. TABLE(ALL) SAMPLE 25 INDEX(ALL)
SHRLEVEL CHANGE");
        ps2.executeUpdate();
        ps2.close();

        // Prepare the statement for the database check
        psDBType = con.prepareStatement("SELECT TYPE FROM SYSIBM.SYSDATABASE WHERE NAME = ?
AND TYPE NOT IN ('W', 'T')");

        // Prepare the statement for the tablespace check
        psTSType = con.prepareStatement("SELECT TYPE FROM SYSIBM.SYSTABLESPACE WHERE DBNAME
= ? AND NAME = ? AND TYPE NOT IN ('O', 'P')");

        if (cs.getMoreResults())
        {
          int objects = 0;

          rs = cs.getResultSet();

          while (rs.next())
          {
            String db = rs.getString(1).trim();
            String name = rs.getString(2).trim();
            String status = rs.getString(9);


            // Verify if we can run RUNSTATS on that tablespace
            // Do not run RUNSTATS on DB2 directory tablespaces
            if (db.equals("DSNDB01"))
              continue;

            // Do not run RUNSTATS on orphaned or restricted tablespaces
            if (status != null)
              continue;

            // Do not run RUNSTATS on WORKFILE or TEMPORARY databases
            psDBType.setString(1, db);
            rsDBType = psDBType.executeQuery();
            if (!rsDBType.next())
              continue;
            rsDBType.close();

            // Do not run RUNSTATS on LOB or XML tablespaces
            psTSType.setString(1, db);
            psTSType.setString(2, name);
            rsTSType = psTSType.executeQuery();
            if (!rsTSType.next())
              continue;
            rsTSType.close();

            // Insert the tablespace
            ps.setInt(1, objects); // Object ID
            ps.setInt(2, 0); // Statement ID
            ps.setString(3, "TABLESPACE"); // Object type
            if (db == null)
            {
```

```
            ps.setNull(4, Types.VARCHAR);
          } else
          {
            ps.setString(4, db); // Object qualifier
          }
          ps.setString(5, name); // Object name
          // m
          System.out.println("Database: " + db + " Object: " + name);


          ps.setNull(6, Types.SMALLINT); // Object partition
          ps.setString(7, "NO"); // Restart
          ps.setString(8, "RUNSTATS TABLESPACE"); // Utility name
          ps.executeUpdate();
          objects++;
        }
        ps.close();
        psDBType.close();
        psTSType.close();
        rs.close();
        cs.close();

        if (objects > 0)
        {
          // Execute utilities in parallel
          cs = con.prepareCall("CALL SYSPROC.ADMIN_UTL_SCHEDULE( ?, ?, ?, ?, ?, ?, " +
                                                                 "?, ?, ?, ?, ?, ?, ?)");
          cs.setShort(1, (short) 4); // Maximum parallel subtasks
          cs.setString(2, "YES"); // Optimize workload
          cs.setString(3, "ERROR"); // Stop condition
          cs.setString(4, "RUNSTATS"); // Utility ID stem
          cs.setNull(5, Types.FLOAT); // Shutdown
          cs.setInt(6, objects); // Number of objects
          cs.registerOutParameter(6, Types.INTEGER); // Number of objects
          cs.registerOutParameter(7, Types.INTEGER); // Utilities executed
          cs.registerOutParameter(8, Types.INTEGER); // Highest DSNUTILU return code
          cs.registerOutParameter(9, Types.SMALLINT); // Actual parallel substasks
          cs.registerOutParameter(10, Types.INTEGER); // Return code
          cs.registerOutParameter(11, Types.VARCHAR); // Message area
          hasResultSet = cs.execute();
          con.commit();

          // ADMIN_UTL_SCHEDULE always has two result sets
          // Display the SYSPRINT result set
          if (hasResultSet)
          {
            rs = cs.getResultSet();
            while (rs.next())
            {
              System.out.println(" OBJECTID = " + rs.getInt(1) + " TEXT = " +
rs.getString(3));
            }
            rs.close();
          }

          // Display the RETCODE result set
          if (cs.getMoreResults())
          {
            rs = cs.getResultSet();
            while (rs.next())
```

```
              {
                System.out.println(" OBJECTID = " + rs.getInt(1) + " RETCODE = " +
rs.getInt(2));
              }
              rs.close();
            }

            int actualNumberOfObjects = cs.getInt(6);
            int utilitiesExecuted = cs.getInt(7);
            int highestDSNUTILUretCode = cs.getInt(8);
            int actualParallel = cs.getInt(9);
            // ADMIN routines always return the RETURN_CODE, the MSG may be null */
            rc = cs.getInt(10);
            resultMessage = cs.getString(11);
            boolean hasMessage = false;
            if (!cs.wasNull())
            {
              hasMessage = true;
            }

            if (rc > 4)
            {
              message = "ADMIN_UTL_SCHEDULE execution failed with RETURN_CODE " + rc;
              if (hasMessage)
              {
                message += " message: " + resultMessage;
              }
              throw new AdminUtilityExecutionException(rc, message);
            }

            // Check if the highest SYSPROC.DSNUTILU return code
            // requires an exception to be thrown
            if (highestDSNUTILUretCode > 4)
            {
              message = "ADMIN_UTL_SCHEDULE execution failed with HIGHEST_RETCODE " +
highestDSNUTILUretCode;
              throw new AdminUtilityExecutionException(rc, message);
            }

            // Print ADMIN_UTL_SCHEDULE message
            message = "ADMIN_UTL_SCHEDULE executed " + utilitiesExecuted + " utility calls
for " + actualNumberOfObjects + " objects in " + actualParallel
                + " subtasks";
            if (hasMessage)
            {
              message += " with the message: " + resultMessage;
            }
            System.out.println(message);
          }
          cs.close();
          System.out.println("AdminUtilityExecution successful.");
        } else
          throw new AdminUtilityExecutionException(rc, "DSNACCOX execution failed: no
result set.");
      } else
        throw new AdminUtilityExecutionException(rc, "DSNACCOX execution failed: no IFI
command result set.");
    }
    catch (AdminUtilityExecutionException re)
    {
```

```
      System.err.println("Program error: rc=" + re.getRC() + " message=" +
re.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException ex)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + ex.getMessage());
      System.err.println("SQLSTATE: " + ex.getSQLState());
      System.err.println("Error code: " + ex.getErrorCode());
    }
    finally
    {
      // Release resources and disconnect
      try
      {
        ps.close();
      } catch (Exception e)
      {
      }

      try
      {
        ps2.close();
      } catch (Exception e)
      {
      }

      try
      {
        psDBType.close();
      } catch (Exception e)
      {
      }

      try
      {
        psTSType.close();
      } catch (Exception e)
      {
      }

      try
      {
        rs.close();
      } catch (Exception e)
      {
      }

      try
      {
        rsDBType.close();
      } catch (Exception e)
      {
      }
```

```
      try
      {
        rsTSType.close();
      } catch (Exception e)
      {
      }

      try
      {
        cs.close();
      } catch (Exception e)
      {
      }

      try
      {
        con.close();
      } catch (Exception e)
      {
      }
    }
  }
}

class AdminUtilityExecutionException extends Exception
{
  private int rc;

  AdminUtilityExecutionException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile AdminUtilityExecution.java and enter the following command to execute it:

```
java AdminUtilityExecution DBALIAS USERID PASSWORD
```

Example A-19 shows a sample output of the AdminUtilityExecution java driver. Due to the provided threshold values, DSNACCOX recommended to run the RUNSTATS utility on two table spaces, DSNDB06.SYSRTSTS and DSN00005.TABRWLMR.

Example: A-19 AdminUtilityExecution output

```
DSNACCOX message: EVALUATED:  26.45 % of Tablespaces;   0.00 % of Indexspaces
Database: DSNDB06 Object: SYSRTSTS
Database: DSN00005 Object: TABRWLMR
 OBJECTID = 0 TEXT = 1DSNU000I    357 05:08:56.54 DSNUGUTC - OUTPUT START FOR UTILITY,
UTILID = RUNSTATS01000001
 OBJECTID = 0 TEXT =  DSNU1045I   357 05:08:56.65 DSNUGTIS - PROCESSING SYSIN AS UNICODE
UTF-8
 OBJECTID = 0 TEXT = 0DSNU050I    357 05:08:56.66 DSNUGUTC -  RUNSTATS TABLESPACE
DSNDB06.SYSRTSTS TABLE(ALL) SAMPLE 25 INDEX(ALL)
```

```
 OBJECTID = 0 TEXT =  SHRLEVEL CHANGE
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.72 DSNUSUTP - SYSTABLEPART CATALOG UPDATE
FOR DSNDB06.SYSRTSTS SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.73 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR
SYSIBM.SYSTABLESPACESTATS SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.80 DSNUSUCO - SYSCOLUMNS CATALOG UPDATE FOR
SYSIBM.SYSTABLESPACESTATS SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.81 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR
SYSIBM.SYSINDEXSPACESTATS SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.91 DSNUSUCO - SYSCOLUMNS CATALOG UPDATE FOR
SYSIBM.SYSINDEXSPACESTATS SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.92 DSNUSUTS - SYSTABLESPACE CATALOG UPDATE
FOR DSNDB06.SYSRTSTS SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.93 DSNUSUIP - SYSINDEXPART CATALOG UPDATE
FOR SYSIBM.DSNRTX01 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.93 DSNUSUIP - SYSINDEXPART CATALOG UPDATE
FOR SYSIBM.DSNRTX02 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.94 DSNUSUIP - SYSINDEXPART CATALOG UPDATE
FOR SYSIBM.DSNRTX03 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.95 DSNUSUCO - SYSCOLUMNS CATALOG UPDATE FOR
SYSIBM.DSNRTX01 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.95 DSNUSUIX - SYSINDEXES CATALOG UPDATE FOR
SYSIBM.DSNRTX01 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.95 DSNUSUCO - SYSCOLUMNS CATALOG UPDATE FOR
SYSIBM.DSNRTX02 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.95 DSNUSUIX - SYSINDEXES CATALOG UPDATE FOR
SYSIBM.DSNRTX02 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.95 DSNUSUCO - SYSCOLUMNS CATALOG UPDATE FOR
SYSIBM.DSNRTX03 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.95 DSNUSUIX - SYSINDEXES CATALOG UPDATE FOR
SYSIBM.DSNRTX03 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.96 DSNUSUCD - SYSCOLDIST CATALOG UPDATE FOR
SYSIBM.DSNRTX01 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.99 DSNUSUCD - SYSCOLDIST CATALOG UPDATE FOR
SYSIBM.DSNRTX02 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU610I  ) 357 05:08:56.99 DSNUSUCD - SYSCOLDIST CATALOG UPDATE FOR
SYSIBM.DSNRTX03 SUCCESSFUL
 OBJECTID = 0 TEXT =  DSNU620I  ) 357 05:08:56.99 DSNUSEOF - RUNSTATS CATALOG TIMESTAMP =
2007-12-23-05.08.56.673183
 OBJECTID = 0 TEXT =  DSNU010I    357 05:08:57.02 DSNUGBAC - UTILITY EXECUTION COMPLETE,
HIGHEST RETURN CODE=0
 OBJECTID = 1 TEXT = 1DSNU000I    357 05:08:57.13 DSNUGUTC - OUTPUT START FOR UTILITY,
UTILID = RUNSTATS02000001
 OBJECTID = 1 TEXT =  DSNU1045I   357 05:08:57.23 DSNUGTIS - PROCESSING SYSIN AS UNICODE
UTF-8
 OBJECTID = 1 TEXT = 0DSNU050I    357 05:08:57.24 DSNUGUTC -  RUNSTATS TABLESPACE
DSN00005.TABRWLMR TABLE(ALL) SAMPLE 25 INDEX(ALL
)
 OBJECTID = 1 TEXT =  SHRLEVEL CHANGE
 OBJECTID = 1 TEXT =  DSNU718I  ) 357 05:08:57.28 DSNUSIIX - NO INDEXES FOUND FOR
TABLESPACE 'DSN00005.TABRWLMR'
 OBJECTID = 1 TEXT =  DSNU610I  ) 357 05:08:57.30 DSNUSUTP - SYSTABLEPART CATALOG UPDATE
FOR DSN00005.TABRWLMR SUCCESSFUL
 OBJECTID = 1 TEXT =  DSNU610I  ) 357 05:08:57.30 DSNUSUPT - SYSTABSTATS CATALOG UPDATE FOR
USER.TAB_WLM_REFR SUCCESSFUL
 OBJECTID = 1 TEXT =  DSNU610I  ) 357 05:08:57.31 DSNUSUPC - SYSCOLSTATS CATALOG UPDATE FOR
USER.TAB_WLM_REFR SUCCESSFUL
 OBJECTID = 1 TEXT =  DSNU610I  ) 357 05:08:57.32 DSNUSUTB - SYSTABLES CATALOG UPDATE FOR
USER.TAB_WLM_REFR SUCCESSFUL
```

```
 OBJECTID = 1 TEXT =  DSNU610I  ) 357 05:08:57.33 DSNUSUCO - SYSCOLUMNS CATALOG UPDATE FOR
USER.TAB_WLM_REFR SUCCESSFUL
 OBJECTID = 1 TEXT =  DSNU610I  ) 357 05:08:57.33 DSNUSUTS - SYSTABLESPACE CATALOG UPDATE
FOR DSN00005.TABRWLMR SUCCESSFUL
 OBJECTID = 1 TEXT =  DSNU620I  ) 357 05:08:57.34 DSNUSEF2 - RUNSTATS CATALOG TIMESTAMP =
2007-12-23-05.08.57.250092
 OBJECTID = 1 TEXT =  DSNU010I    357 05:08:57.35 DSNUGBAC - UTILITY EXECUTION COMPLETE,
HIGHEST RETURN CODE=4
 OBJECTID = 0 RETCODE = 0
 OBJECTID = 1 RETCODE = 4
ADMIN_UTL_SCHEDULE executed 2 utility calls for 2 objects in 2 subtasks
AdminUtilityExecution successful.
```

A.5 Manage data sets with AdminDataSet

AdminDataSet uses ADMIN_DS_WRITE to create a PS data set named DSTEST using the caller's user ID as HLQ. It then browses the data set employing ADMIN_DS_BROWSE and renames it to DSTEST2 using ADMIN_DS_RENAME. Next, it checks whether it is cataloged using ADMIN_DS_SEARCH, lists it using ADMIN_DS_LIST, and finally deletes it using ADMIN_DS_DELETE.

Example A-20 shows how to use the stored procedures for data set manipulation.

Example: A-20 AdminDataSet

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminDataSet.java
//
// Sample: How to use the DB2 provided data set manipulation stored
//         procedures
//
//The user runs the program by issuing:
//java AdminDataSet <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************
import java.sql.*;

public class AdminDataSet
{
  public static void main(String args[])
  {
    Connection con = null;
    PreparedStatement ps = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
```

```
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminDataSet <alias or //server/database> <userid>
<password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
    {
      String dsName = userid + ".DSTEST";
      String dsName2 = userid + ".DSTEST2";
      int rc = 0;
      String message = null;
      boolean hasResultSet = false;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      // Create a data set
      ps = con.prepareStatement("INSERT INTO SYSIBM.TEXT_REC_INPUT(ROWNUM, TEXT_REC)
VALUES(?,?)");
      ps.setInt(1, 1);
      ps.setString(2, "This is the first row of my LRECL=80 data set.");
      ps.execute();
      ps.setInt(1, 2);
      ps.setString(2, "This is the second row.");
      ps.execute();

      cs = con.prepareCall("CALL SYSPROC.ADMIN_DS_WRITE(?, ?, ?, ?, ?, ?, ?)");
      cs.setInt(1, 1);                              // Data type (1, 2)
      cs.setString(2, dsName);                      // Data set name or GDG
      cs.setString(3, "");                          // Member name, generation # (+1, -1,
0, +2) or blank
      cs.setString(4, "ND");                        // Option (ND, NM, A, R)
      cs.setString(5, "N");                         // Dump option (Y, N)
      cs.registerOutParameter(6,Types.INTEGER);     // Return code
      cs.registerOutParameter(7,Types.LONGVARCHAR); // Message area

      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(6);
      if (rc > 0)
      {
```

```
          message = cs.getString(7);
          throw new AdminDataSetException(rc, "ADMIN_DS_WRITE execution failed: " + message);
        }
        else
        {
        ps.close();
        cs.close();
          System.out.println(dsName + " created and has records to be browsed.");
        }

        // Browse the data set
        cs = con.prepareCall("CALL SYSPROC.ADMIN_DS_BROWSE(?, ?, ?, ?, ?, ?)");
        cs.setInt(1, 1);                              // Data type (1, 2)
        cs.setString(2, dsName);                      // Data set name
        cs.setString(3, "");                          // Member name
        cs.setString(4, "N");                         // Dump option (Y, N)
        cs.registerOutParameter(5,Types.INTEGER);     // Return code
        cs.registerOutParameter(6,Types.LONGVARCHAR); // Message area
        cs.execute();
        con.commit();

        hasResultSet = cs.execute();

        // Obtain the return code
        rc = cs.getInt(5);
        if (rc > 0)
        {
          message = cs.getString(6);
          throw new AdminDataSetException(rc, "ADMIN_DS_BROWSE execution failed: " +
message);
        }
        else
        {
          if (hasResultSet)
          {
            rs = cs.getResultSet();
            while (rs.next())
            {
              System.out.println(rs.getString(2).trim());
            }
          }
          rs.close();
          cs.close();
        }

        // Rename the data set
        cs = con.prepareCall("CALL SYSPROC.ADMIN_DS_RENAME(?, ?, ?, ?, ?, ?, ?)");
        cs.setInt(1, 4);                              // Data set type
(1-pds,2-pdse,3-mbr,4-ps)
        cs.setString(2, dsName);                      // Data set or member name
        cs.setString(3, "");                          // Parent data set name or blank
        cs.setString(4, dsName2);                     // New data set or member name
        cs.setString(5, "N");                         // Dump option (Y, N)
        cs.registerOutParameter(6,Types.INTEGER);     // Return code
        cs.registerOutParameter(7,Types.LONGVARCHAR); // Message area
        cs.execute();
        con.commit();

        // Obtain the return code
        rc = cs.getInt(6);
```

```
      if (rc > 0)
      {
        message = cs.getString(7);
        throw new AdminDataSetException(rc, "ADMIN_DS_RENAME execution failed: " +
message);
      }
      else
      {
      cs.close();
        System.out.println(dsName + " renamed to " + dsName2 + ".");
      }

      // Check if the data set exists
      cs = con.prepareCall("CALL SYSPROC.ADMIN_DS_SEARCH(?, ?, ?, ?, ?, ?)");
      cs.setString(1, dsName2);                      // Data set name or GDG
      cs.setString(2, "");                           // Member name
      cs.setString(3, "N");                          // Dump option (Y, N)
      cs.registerOutParameter(4,Types.INTEGER);      // Exist
      cs.registerOutParameter(5,Types.INTEGER);      // Return code
      cs.registerOutParameter(6,Types.LONGVARCHAR); // Message area
      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(5);
      if (rc > 0)
      {
        message = cs.getString(6);
        throw new AdminDataSetException(rc, "ADMIN_DS_SEARCH execution failed: " +
message);
      }
      else
      {
      int exist = cs.getInt(4);
        cs.close();
        if (exist == 0)
          System.out.println(dsName2 + " exists.");
        else if (exist == 1)
          System.out.println(dsName2 + " does not exist.");
      }

      // List everything under the user HLQ
      cs = con.prepareCall("CALL SYSPROC.ADMIN_DS_LIST(?, ?, ?, ?, ?, ?, ?)");
      cs.setString(1, userid + ".**");               // Data set name or filter
      cs.setString(2, "N");                          // List members (Y, N)
      cs.setString(3, "N");                          // List generations (Y, N)
      cs.setInt(4, 50);                              // Maximum number of data sets returned
      cs.setString(5, "N");                          // Dump option (Y, N)
      cs.registerOutParameter(6,Types.INTEGER);      // Return code
      cs.registerOutParameter(7,Types.LONGVARCHAR); // Message area

      hasResultSet = cs.execute();

      // Obtain the return code
      rc = cs.getInt(6);
      if (rc > 0)
      {
        message = cs.getString(7);
        throw new AdminDataSetException(rc, "ADMIN_DS_LIST execution failed: " + message);
      }
```

```
      else
      {
        if (hasResultSet)
        {
          rs = cs.getResultSet();
          while (rs.next())
          {

System.out.println("----------------------------------------------------------------");
            System.out.println("DSNAME           = " + rs.getString(1).trim());
            System.out.println("CREATE_YEAR      = " + rs.getInt(2));
            System.out.println("CREATE_DAY       = " + rs.getInt(3));
            System.out.println("TYPE             = " + rs.getInt(4));
            System.out.println("VOLUME           = " + rs.getString(5).trim());
            System.out.println("PRIMARY_EXTENT   = " + rs.getInt(6));
            System.out.println("SECONDARY_EXTENT = " + rs.getInt(7));
            System.out.println("MEASUREMENT_UNIT = " + rs.getString(8).trim());
            System.out.println("EXTENTS_IN_USE   = " + rs.getInt(9));
            System.out.println("DASD_USAGE       = " + rs.getString(10));
            System.out.println("HARBA            = " + rs.getString(11));
            System.out.println("HURBA            = " + rs.getString(12));

System.out.println("----------------------------------------------------------------");
          }
        }
        rs.close();
        cs.close();
      }

      // Delete the data set
      cs = con.prepareCall("CALL SYSPROC.ADMIN_DS_DELETE(?, ?, ?, ?, ?, ?)");
      cs.setInt(1, 4);                                  // Data set type
(1-pds,2-pdse,3-mbr,4-ps,6-gds)
      cs.setString(2, dsName2);                         // Data set name, member name or
generation # (G0001V00)
      cs.setString(3, "");                              // Parent data set name, GDG or blank
      cs.setString(4, "N");                             // Dump option (Y, N)
      cs.registerOutParameter(5,Types.INTEGER);     // Return code
      cs.registerOutParameter(6,Types.LONGVARCHAR); // Message area

      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(5);
      if (rc > 0)
      {
        message = cs.getString(6);
        throw new AdminDataSetException(rc, "ADMIN_DS_DELETE Execution failed: " +
message);
      }
      else
      {
      cs.close();
        System.out.println(dsName2 + " deleted.");
      }
    }
    catch (AdminDataSetException adse)
    {
```

```
      System.err.println("Program error: rc=" + adse.getRC() + " message=" +
adse.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());

    }
    finally
    {
      // Release resources and disconnect
      try
      {
        ps.close();
      } catch (Exception e)
      {
      }

      try
      {
        rs.close();
      } catch (Exception e)
      {
      }

      try
      {
        cs.close();
      } catch (Exception e)
      {
      }

      try
      {
        con.close();
      } catch (Exception e)
      {
      }
    }
  }
}

class AdminDataSetException extends Exception
{
  private int rc;

  AdminDataSetException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }
```

```
  public int getRC()
  {
    return rc;
  }
}
```

Compile AdminDataSet.java and enter the following command to execute it:

```
java AdminDataSet DBALIAS USERID PASSWORD
```

You receive the response shown in Example A-21.

Example: A-21 Response to AdminDataSet

```
PAOLOR3.DSTEST created and has records to be browsed.
This is the first row of my LRECL=80 data set.
This is the second row.
PAOLOR3.DSTEST renamed to PAOLOR3.DSTEST2.
PAOLOR3.DSTEST2 exists.
------------------------------------------------------------------
DSNAME           = PAOLOR3.DSNCLIST
CREATE_YEAR      = 1996
CREATE_DAY       = 275
TYPE             = 1
VOLUME           = USER01
PRIMARY_EXTENT   = 1
SECONDARY_EXTENT = 3
MEASUREMENT_UNIT = TRACKS
EXTENTS_IN_USE   = 1
DASD_USAGE       = ffffffffffffffff
HARBA            = fffffffffff
HURBA            = fffffffffff
------------------------------------------------------------------
------------------------------------------------------------------
DSNAME           = PAOLOR3.DSNSPFM
CREATE_YEAR      = 1996
CREATE_DAY       = 275
TYPE             = 1
VOLUME           = USER01
PRIMARY_EXTENT   = 1
SECONDARY_EXTENT = 2
MEASUREMENT_UNIT = TRACKS
EXTENTS_IN_USE   = 1
DASD_USAGE       = ffffffffffffffff
HARBA            = fffffffffff
HURBA            = fffffffffff
------------------------------------------------------------------
------------------------------------------------------------------
DSNAME           = PAOLOR3.DSTEST2
CREATE_YEAR      = 2007
CREATE_DAY       = 351
TYPE             = 4
VOLUME           = SCR03
PRIMARY_EXTENT   = 2
SECONDARY_EXTENT = 10
MEASUREMENT_UNIT = BLOCKS
EXTENTS_IN_USE   = 1
DASD_USAGE       = ffffffffffffffff
HARBA            = fffffffffff
```

```
HURBA            = fffffffffff
------------------------------------------------------------
------------------------------------------------------------
PAOLOR3.DSTEST2 deleted.
```

A.6 Submit JCL with AdminJob

AdminJob uses ADMIN_JOB_SUBMIT to submit JCL to compress an existing PDS. Then it uses ADMIN_JOB_QUERY to poll the job status until the job is in the OUT queue. It then uses ADMIN_JOB_FETCH to fetch the job output and prints it. Finally, it calls ADMIN_JOB_CANCEL to purge the job output.

Example A-22 shows how to submit JCL with AdminJob.

Example: A-22 AdminJob

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminJob.java
//
// Sample: How to use the DB2 provided JCL administration stored procedures
//
//The user runs the program by issuing:
//java AdminJob <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************
import java.sql.*;

public class AdminJob
{
  public static void main(String[] args)
  {
    Connection con = null;
    PreparedStatement ps = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminJob <alias or //server/database> <userid>
<password>");
```

```
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
    {
      String jobid = null;
      String[] jclstmt =
        {
          "//IEBCOPY  JOB ,CLASS=K,MSGCLASS=H,MSGLEVEL=(1,1)",
          "//COPY    EXEC PGM=IEBCOPY,DYNAMNBR=20",
          "//SYSUT1   DD  DSN=SG247083.PROD.SOURCE,DISP=SHR",
          "//SYSUT2   DD  DSN=SG247083.PROD.SOURCE,DISP=SHR",
          "//SYSPRINT DD  SYSOUT=*",
          "//SYSIN    DD  *"
        };
      int jobstatus = 0;
      int retrycount = 0;
      int rc = 0;
      String message = null;
      boolean hasResultSet = false;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      // Submit JCL
      ps = con.prepareStatement("INSERT INTO SYSIBM.JOB_JCL(ROWNUM, STMT) VALUES(?, ?)");
      for (int i = 0; i < jclstmt.length; i++)
      {
        ps.setInt(1, i + 1);
        ps.setString(2, jclstmt[i]);
        ps.execute();
      }

      cs = con.prepareCall("CALL SYSPROC.ADMIN_JOB_SUBMIT(?, ?, ?, ?, ?)");
      cs.setString(1, userid);                      // User ID
      cs.setString(2, password);                    // Password
      cs.registerOutParameter(3,Types.VARCHAR);     // Job ID
      cs.registerOutParameter(4,Types.INTEGER);     // Return code
      cs.registerOutParameter(5,Types.LONGVARCHAR); // Message area
      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(4);
      if (rc > 0)
      {
        message = cs.getString(5);
        throw new AdminJobException(rc, "SYSPROC.ADMIN_JOB_SUBMIT execution failed: " +
message);
```

```
      }
      else
      {
        jobid = cs.getString(3);
        System.out.println("Job " + jobid + " submitted successfully.");
      ps.close();
      cs.close();
      }

      /* Query job status */
      cs = con.prepareCall("CALL SYSPROC.ADMIN_JOB_QUERY(?, ?, ?, ?, ?, ?, ?, ?, ?, ?)" );
      cs.setString(1, userid);                        // User ID
      cs.setString(2, password);                      // Password
      cs.setString(3, jobid);                         // Job ID
      cs.registerOutParameter(4,Types.INTEGER);       // Job status
      cs.registerOutParameter(5,Types.INTEGER);       // Max RC
      cs.registerOutParameter(6,Types.INTEGER);       // Completion type
      cs.registerOutParameter(7,Types.INTEGER);       // System abend code
      cs.registerOutParameter(8,Types.INTEGER);       // User abend code
      cs.registerOutParameter(9,Types.INTEGER);       // Return code
      cs.registerOutParameter(10,Types.LONGVARCHAR); // Message area

      while (true)
      {
        cs.execute();
        con.commit();

        // Obtain return code
        rc = cs.getInt(9);
        if (rc > 4)
        {
          message = cs.getString(10);
          throw new AdminJobException(rc, "SYSPROC.ADMIN_JOB_QUERY execution failed:" +
message);
        }
        else
        {
          jobstatus = cs.getInt(4);

          if (rc == 0)
          {
            // The job is in the OUTPUT queue
            if (jobstatus == 3)
            {
              System.out.println("Job " + jobid + " finished execution. Job completion
information: ");
              System.out.println("   Max RC:            " + cs.getInt(5));
              System.out.println("   Completion type:   " + cs.getInt(6));
              System.out.println("   System abend code: " + cs.getInt(7));
              System.out.println("   User abend code:   " + cs.getInt(8));
              break;
            }
            else if (jobstatus == 1 || jobstatus == 2)
            {
              // The job is in the INPUT or ACTIVE queue
              System.out.println("Job " + jobid + " is in the " + (jobstatus == 1 ? "INPUT"
: "ACTIVE") + " queue. Waiting for job to finish...");
              Thread.sleep(1000);
              continue;
            }
```

```
          }
          else if (rc == 4)
          {
            if (jobstatus == 5)
            {
              // The job is in an unknown phase
              System.out.println("Job " + jobid + " is in an unknown phase. Waiting for job
to finish...");
              Thread.sleep(1000);
              continue;
            }
            else if (jobstatus == 4)
            {
              if (retrycount == 10)
                throw new AdminJobException(rc, "Job " + jobid + " not found:" + message);
              else
              {
                System.out.println("Job " + jobid + " not found. Waiting for job...");
                Thread.sleep(1000);
                retrycount++;
                continue;
              }
            }
          }
        }
      }

      cs.close();
      System.out.println("Job " + jobid + " has finished and has output to be fetched.");

      // Fetch job output
      cs = con.prepareCall("CALL SYSPROC.ADMIN_JOB_FETCH(?, ?, ?, ?, ?)");
      cs.setString(1, userid);                      // User ID
      cs.setString(2, password);                    // Password
      cs.setString(3, jobid);                       // Job ID
      cs.registerOutParameter(4,Types.INTEGER);     // Return code
      cs.registerOutParameter(5,Types.LONGVARCHAR); // Message area
      hasResultSet = cs.execute();
      con.commit();

      // Obtain return code
      rc = cs.getInt(4);
      if (rc > 0)
      {
        message = cs.getString(5);
        throw new AdminJobException(rc, "SYSPROC.ADMIN_JOB_FETCH execution failed: " +
message);
      }
      else
      {
        if (hasResultSet)
        {
          rs = cs.getResultSet();
          while (rs.next())
          {
            System.out.println(rs.getString(2));
          }
          rs.close();
        }
        cs.close();
```

```
      }

      // Purge job output
      cs = con.prepareCall("CALL SYSPROC.ADMIN_JOB_CANCEL(?, ?, ?, ?, ?, ?)");
      cs.setString(1, userid);                      // User ID
      cs.setString(2, password);                    // Password
      cs.setInt(3, 2);                              // Processing option (Cancel=1,
Purge=2)
      cs.setString(4, jobid);                       // Job ID
      cs.registerOutParameter(5,Types.INTEGER);     // Return code
      cs.registerOutParameter(6,Types.LONGVARCHAR); // Message area
      cs.execute();
      con.commit();

      // Obtain return code
      rc = cs.getInt(5);

      if (rc > 0)
      {
        message = cs.getString(6);
        throw new AdminJobException(rc, "SYSPROC.ADMIN_JOB_CANCEL execution failed: " +
message);
      }
      else
      {
        System.out.println("Job " + jobid + " has been purged.");
        cs.close();
      }
    }
    catch (AdminJobException aje)
    {
      System.err.println("Program error: rc=" + aje.getRC() + " message=" +
aje.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());
    }
    catch (Exception e)
    {
      System.err.println("Error: message=" + e.getMessage());
    }
    finally
    {
      // Release resources and disconnect
      try
      {
        ps.close();
      } catch (Exception e)
      {
      }
```

```
      try
      {
        rs.close();
      } catch (Exception e)
      {
      }

      try
      {
        cs.close();
      } catch (Exception e)
      {
      }

      try
      {
        con.close();
      } catch (Exception e)
      {
      }
    }
  }
}

class AdminJobException extends Exception
{
  private int rc;

  AdminJobException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile AdminJob.java and enter the following command to execute it:

```
java AdminJob DBALIAS USERID PASSWORD
```

You should get the response shown in Example A-23.

Example: A-23 Response to AdminJob

```
Job JOB00087 submitted successfully.
Job JOB00087 is in the ACTIVE queue. Waiting for job to finish...
Job JOB00087 is in the ACTIVE queue. Waiting for job to finish...
Job JOB00087 is in the ACTIVE queue. Waiting for job to finish...
Job JOB00087 is in the ACTIVE queue. Waiting for job to finish...
Job JOB00087 finished execution. Job completion information:
   Max RC:            0
   Completion type:   1
   System abend code: 0
   User abend code:   0
Job JOB00087 has finished and has output to be fetched.
```

```
                J E S 2  J O B  L O G  --  S Y S T E M  S T L 0  --  N O D E  S T L V
M 3

JOB00087 ---- MONDAY,    17 DEC 2007 ----
JOB00087  IRR010I  USERID SYSADM   IS ASSIGNED TO THIS JOB.
JOB00087  ICH70001I SYSADM   LAST ACCESS AT 10:37:04 ON MONDAY, DECEMBER 17, 2007
JOB00087  $HASP373 IEBCOPY  STARTED - INIT 4    - CLASS K - SYS STL0
JOB00087  SMF000I  IEBCOPY     COPY        IEBCOPY     0000
JOB00087  $HASP395 IEBCOPY  ENDED
------ JES2 JOB STATISTICS ------
  17 DEC 2007 JOB EXECUTION DATE
            6 CARDS READ
           49 SYSOUT PRINT RECORDS
            0 SYSOUT PUNCH RECORDS
            3 SYSOUT SPOOL KBYTES
         0.00 MINUTES EXECUTION TIME
        1 //IEBCOPY  JOB ,CLASS=K,MSGCLASS=H,MSGLEVEL=(1,1)                      JOB00087
        2 //COPY    EXEC PGM=IEBCOPY,DYNAMNBR=20
        3 //SYSUT1   DD  DSN=SG247083.PROD.SOURCE,DISP=SHR
        4 //SYSUT2   DD  DSN=SG247083.PROD.SOURCE,DISP=SHR
        5 //SYSPRINT DD  SYSOUT=*
        6 //SYSIN    DD  *
ICH70001I SYSADM   LAST ACCESS AT 10:37:04 ON MONDAY, DECEMBER 17, 2007
IEF236I ALLOC. FOR IEBCOPY COPY
IEF237I 04B0 ALLOCATED TO SYSUT1
IEF237I 04B0 ALLOCATED TO SYSUT2
IEF237I JES2 ALLOCATED TO SYSPRINT
IEF237I JES2 ALLOCATED TO SYSIN
IEF142I IEBCOPY COPY - STEP WAS EXECUTED - COND CODE 0000
IEF285I   SG247083.PROD.SOURCE                           KEPT
IEF285I   VOL SER NOS= USER01.
IEF285I   SG247083.PROD.SOURCE                           KEPT
IEF285I   VOL SER NOS= USER01.
IEF285I   SYSADM.IEBCOPY.JOB00087.D0000102.?             SYSOUT
IEF285I   SYSADM.IEBCOPY.JOB00087.D0000101.?             SYSIN
IEF373I STEP/COPY    /START 2007351.1038
IEF374I STEP/COPY    /STOP  2007351.1038 CPU    0MIN 00.01SEC SRB    0MIN 00.00SEC VIRT
1024K SYS   248K EXT       4K SYS   11312K
IEF375I  JOB/IEBCOPY /START 2007351.1038
IEF376I  JOB/IEBCOPY /STOP  2007351.1038 CPU    0MIN 00.01SEC SRB    0MIN 00.00SEC
                           IEBCOPY MESSAGES AND CONTROL STATEMENTS
PAGE     1
IEB1135I IEBCOPY  FMID HDZ11G0  SERVICE LEVEL UA23763  DATED 20060119 DFSMS 01.03.00 z/OS
01.04.00 HBB7707  CPU 4381
IEB1035I IEBCOPY   COPY     10:38:26 MON 17 DEC 2007 PARM=''
COPY     COPY      INDD=SYSUT1,OUTDD=SYSUT2         GENERATED STATEMENT
IEB1018I COMPRESSING  PDS  OUTDD=SYSUT2   VOL=USER01 DSN=SG247083.PROD.SOURCE
IEB1097I FOLLOWING MEMBER(S) MOVED IN DATA SET REFERENCED BY SYSUT2
IEB154I SQL      HAS BEEN SUCCESSFULLY MOVED
IEB1098I 1 OF 1 MEMBERS MOVED IN DATA SET REFERENCED BY SYSUT1
IEB144I THERE ARE 4 UNUSED TRACKS IN OUTPUT DATA SET REFERENCED BY SYSUT2
IEB149I THERE ARE 0 UNUSED DIRECTORY BLOCKS IN OUTPUT DIRECTORY
IEB147I END OF JOB - 0 WAS HIGHEST SEVERITY CODE
Job JOB00087 has been purged.
```

A.7 Issue USS commands with AdminUNIXCommand

AdminUNIXCommand uses ADMIN_COMMAND_UNIX to issue the following command:

```
ls -lat
```

Example A-24 shows how to submit USS commands through stored procedures.

Example: A-24 AdminUNIXCommand

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminUNIXCommand.java
//
// Sample: How to use the DB2 provided stored procedures to issue
//         a USS command.
//
//The user runs the program by issuing:
//java AdminUNIXCommand <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************
import java.sql.*;

public class AdminUNIXCommand
{
  public static void main(String[] args)
  {
    Connection con = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminUNIXCommand <alias or //server/database> <userid>
<password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
```

```
    {
      int rc = 0;
      String message = null;
      boolean hasResultSet = false;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      // Execute UNIX command
      cs = con.prepareCall("CALL SYSPROC.ADMIN_COMMAND_UNIX(?, ?, ?, ?, ?, ?)");
      cs.setString(1, userid);                        // User ID
      cs.setString(2, password);                      // Password
      cs.setString(3, "ls -lat");                     // USS command
      cs.setString(4, "OUTMODE=LINE");                // Outmode (LINE or BLK)
      cs.registerOutParameter(5,Types.INTEGER);       // Return code
      cs.registerOutParameter(6,Types.LONGVARCHAR);  // Message area
      hasResultSet = cs.execute();

      // Obtain the return code
      rc = cs.getInt(5);
      if (rc > 0)
      {
        message = cs.getString(6);
        throw new AdminUNIXCommandException(rc, "ADMIN_COMMAND_UNIX execution failed: " +
message);
      }
      else
      {
        if (hasResultSet)
        {
          rs = cs.getResultSet();
          while (rs.next())
          {
            System.out.println(rs.getString(2).trim());
          }
        }
        rs.close();
        cs.close();
      }
    }

    catch (AdminUNIXCommandException auce)
    {
      System.err.println("Program error: rc=" + auce.getRC() + " message=" +
auce.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
```

```
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());

    }
    finally
    {
      // Release resources and disconnect
      try
      {
        rs.close();
      } catch (Exception e)
      {
      }

      try
      {
        cs.close();
      } catch (Exception e)
      {
      }

      try
      {
        con.close();
      } catch (Exception e)
      {
      }
    }
  }
}

class AdminUNIXCommandException extends Exception
{
  private int rc;

  AdminUNIXCommandException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile AdminUNIXCommand.java and enter the following command to execute it:

```
java AdminUNIXCommand DBALIAS USERID PASSWORD
```

You should get the response shown in Example A-25.

Example: A-25 Response to AdminUNIXCommand

```
total 32
drwxr-xr-x   2 OEDFLTU  SYS1        8192 Dec 17 18:34 .
-rwxr-xr-x   1 OEDFLTU  SYS1           0 Dec 17 18:34 dsnadmcu351183437
drwxr-xr-x  30 OMVSKERN OMVSGRP     8192 Dec 17 14:29 ..
```

A.8 Issue DSN subcommands with AdminDSNSubcommand

AdminDSNSubcommand uses the DB2-supplied stored procedure ADMIN_COMMAND_DSN to issue a REBIND PACKAGE command.

Example A-26 shows how to submit DSN subcommands through stored procedures.

Example: A-26 AdminDSNSubcommand

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminDSNSubcommand.java
//
// Sample: How to use the DB2 provided procedure ADMIN_COMMAND_DSN to issue
//         DSN subcommands.
//
//The user runs the program by issuing:
//java AdminDSNSubcommand <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*************************************************************************
import java.sql.*;

public class AdminDSNSubcommand
{
  public static void main(String[] args)
  {
    Connection con = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminDSNSubcommand <alias or //server/database> <userid>
<password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
```

```
    {
      String message = null;
      boolean hasResultSet = false;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      // Execute DSN subcommand
      cs = con.prepareCall("CALL SYSPROC.ADMIN_COMMAND_DSN(?, ?)");
      cs.setString(1, "REBIND PACKAGE (DSNUTILU.DSNUTILU.(V9R1))");  // DSN subcommand
      cs.registerOutParameter(2,Types.LONGVARCHAR);                  // Message area
      hasResultSet = cs.execute();

      if (hasResultSet)
      {
        rs = cs.getResultSet();
        while (rs.next())
          {
            System.out.println(rs.getString(2).trim());
          }
        rs.close();
      }

      // Obtain the message
      message = cs.getString(2).trim();
      if (message != null &&
          message.length() != 0)
        throw new AdminDSNSubcommandException("ADMIN_COMMAND_DSN execution failed: " +
message);
    }

    catch (AdminDSNSubcommandException adse)
    {
      System.err.println("Program error: message=" + adse.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());
    }
    finally
    {
      // Release resources and disconnect
      try
      {
        rs.close();
      } catch (Exception e)
      {
```

```
      }

      try
      {
        cs.close();
      } catch (Exception e)
      {
      }

      try
      {
        con.close();
      } catch (Exception e)
      {
      }
    }
  }
}

class AdminDSNSubcommandException extends Exception
{

  AdminDSNSubcommandException(String message)
  {
    super(message);
  }
}
```

Compile AdminDSNSubcommand.java and enter the following command to execute it:

```
java AdminDSNSubcommand DBALIAS USERID PASSWORD
```

You get the response shown in Example A-25.

Example: A-27 Response to AdminDSNSubcommand

```
DSNE932I WARNING, ONLY IBM-SUPPLIED COLLECTION-IDS SHOULD BEGIN WITH  "DSN"
DSNE932I WARNING, ONLY IBM-SUPPLIED PACKAGE-IDS SHOULD BEGIN WITH  "DSN"
DSNT254I  = DSNTBRB2 REBIND OPTIONS FOR
PACKAGE = STLEC1.DSNUTILU.DSNUTILU.(V9R1)
ACTION
OWNER         SYSADM
QUALIFIER     SYSADM
VALIDATE      BIND
EXPLAIN       NO
ISOLATION     CS
RELEASE
COPY
DSNT255I  = DSNTBRB2 REBIND OPTIONS FOR
PACKAGE = STLEC1.DSNUTILU.DSNUTILU.(V9R1)
SQLERROR      NOPACKAGE
CURRENTDATA   NO
DEGREE        1
DYNAMICRULES
DEFER
REOPT         NONE
KEEPDYNAMIC   NO
IMMEDWRITE    NO
DBPROTOCOL    DRDA
OPTHINT
```

```
ENCODING      EBCDIC(00037)
PATH
DSNT232I  = SUCCESSFUL REBIND FOR
PACKAGE = STLEC1.DSNUTILU.DSNUTILU.(V9R1)
```

A.9 Task Scheduler Sample Use cases

The following section shows some use cases that employ the DB2-provided task scheduler to schedule the execution of stored procedures. These use cases are enriched with the respective Java source code that performs the CALL to the ADMIN_TASK_ADD stored procedure and therefore provide a good way to get started working with the scheduler.

A.9.1 Use case - 1

Schedule a one-time stored procedure with static input parameters

An administrator seeks to run a single REORG job on a table space. Since the daily business should not be impacted by the utility execution, the job invocation should occur during a nightly maintenance window starting at 1a.m. in the morning. Therefore, he employs the DB2-provided scheduler to schedule a CALL to the DB2-supplied stored procedure SYSPROC.DSNUTILU that eventually executes the REORG utility.

The scenario requires that the stored procedure DSNUTILU is executed only once at a later time. The main scheduling parameters involved in this scenario are BEGIN_TIMESTAMP and MAX_INVOCATIONS. Because there is no END_TIMESTAMP, the stored procedures validity time window would never expire. It is the value of the MAX_INVOCATIONS parameter that constraints the task to be executed only once. The DSNUTILU input parameters are known at scheduling time and the stored procedure is only executed once, therefore the PROCEDURE_INPUT parameter is initialized in a static way. Furthermore, the task name is set to REORG_JOB.

The Java sample in Example A-28 shows how the parameters of ADMIN_TASK_ADD are properly initialized to schedule a CALL to DSNUTILU at 01:00AM on the October 30. The created task is named REORG_JOB. Note that the stored procedure parameters USERID and PASSWORD are initialized with some dummy values (MYUSERID, MYPASSWD). When using this sample, substitute these values with valid credentials.

Example: A-28 ADMIN_TASK_ADD parm initialization

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: AdminSchedule1.java
//
// Sample: How to use the DB2 provided scheduler to schedule a one time
//         execution of DSNUTILU with static input parameters.
//         This implements use case 1 of the scheduling chapter.
//
// The user runs the program by issuing:
// java AdminSchedule1 <alias or //server/database> <userid> <password>
//
// The arguments are:
```

```
// <alias> - DB2 subsystem alias for type 2 or //server/database for
//           type 4
//  connectivity
// <userid> - user ID to connect as
// <password> - password to connect with
//*************************************************************************
import java.sql.*;

public class AdminSchedule1
{
  public static void main(String args[])
  {
    Connection con = null;
    CallableStatement cs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminSchedule1 <alias or //server/database> <userid>
<password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
    {
      int rc = 0;
      String message = null;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      cs = con.prepareCall( "CALL SYSPROC.ADMIN_TASK_ADD("
         + "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? )" );

      /* provide the authid */
      cs.setString(1, "MYUSERID");                  // USERID
      /* provide the password */
      cs.setString(2, "MYPASSWD");                  // PASSWORD
      /* Start the task on the 30th of October */
      cs.setTimestamp( 3, Timestamp.valueOf( "2007-10-30 01:00:00.000000000") ); //
BEGIN_TIMESTAMP

      /* This is a one time task, no end timestamp required */
      cs.setNull( 4, Types.TIMESTAMP);              // END_TIMESTAMP
      /* Only one invocation */
```

```
      cs.setInt( 5, 1);                              // MAX_INVOCATIONS
      /* No interval required */
      cs.setNull( 6, Types.INTEGER);                 // INTERVAL
      /* No point in time required */
      cs.setNull( 7, Types.VARCHAR);                 // POINT_IN_TIME

      /* This is a triggering task, not a triggered task */
      cs.setNull( 8, Types.VARCHAR);                 // TRIGGER_TASK_NAME
      cs.setNull( 9, Types.CHAR);                    // TRIGGER_TASK_COND
      cs.setNull(10, Types.INTEGER);                 // TRIGGER_TASK_CODE

      /* No member affinity required */
      cs.setNull(11, Types.VARCHAR);                 // DB2_SSID

      /* This scheduled stored procedure invokes DSNUTILU to execute a REORG */
      cs.setString(12, "SYSPROC");                  // PROCEDURE_SCHEMA
      cs.setString(13, "DSNUTILU");                 // PROCEDURE_NAME
      cs.setString(14, "SELECT 'REORG', 'NO', 'REORG TABLESPACE DSN8D81A.DSN8S81D NOSYSREC
LOG NO', 1 FROM SYSIBM.SYSDUMMY1");

      /* JCL parameter not required */
      cs.setNull(15, Types.VARCHAR);                 // JCL_LIBRARY
      cs.setNull(16, Types.VARCHAR);                 // JCL_MEMBER
      cs.setNull(17, Types.VARCHAR);                 // JOB_WAIT

      cs.setString(18, "REORG_JOB");                 // TASK_NAME
      cs.registerOutParameter(18, Types.VARCHAR);    // TASK_NAME

      /* Provide a task description */
      cs.setString(19, "Scheduled reorg job");       // DESCRIPTION

      /* Register out parameters */
      cs.registerOutParameter(20, Types.INTEGER);    // RETURN CODE
      cs.registerOutParameter(21, Types.VARCHAR);    // MESSAGE

      /* Schedule task */
      cs.execute();

      // Obtain the return code
      rc = cs.getInt(20);
      if (rc == 0)
      {
         /* Task successfully scheduled */
         System.out.print("\nTask " + cs.getString(18) + " successfully added");
      }
      else
      {
         /* Error during task scheduling */
         message = cs.getString(21);
        throw new AdminScheduleException(rc, "ADMIN_TASK_ADD execution failed: " +
message);
      }
    }
    catch (AdminScheduleException ase)
    {
      System.err.println("Program error: rc=" + ase.getRC() + " message=" +
ase.getMessage());
    }
    catch (ClassNotFoundException e)
    {
```

```
      System.err.println("Could not load JDBC driver");
      System.err.println("Exception: " + e);
      e.printStackTrace();
    }
    catch (SQLException sqle)
    {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());
      System.err.println("SQLSTATE: " + sqle.getSQLState());
      System.err.println("Error code: " + sqle.getErrorCode());
    }
    finally
    {
      // Release resources and disconnect
      try {
        if(cs != null)
           cs.close();
      }
      catch (Exception e) {
      }
      try {
        if(con != null) {
           con.commit();
           con.close();
        }
      }
      catch (Exception e) {
      }
    }
  }
}

class AdminScheduleException extends Exception
{
  private int rc;

  AdminScheduleException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

A.9.2 Use case - 2

Schedule a one-time stored procedure with static input parameter out of a TRIGGER

An administrator wants a certain WLM environment to be automatically refreshed at 06:00 AM the next morning, in case that, for example, the WLM JCL procedure changed. He therefore creates a BEFORE TRIGGER that schedules a CALL to the DB2-supplied WLM_REFRESH stored procedure. The triggering event is an INSERT to a table that contains columns for the WLM environment and the DB2 subsystem ID that needs to be

refreshed. The triggered event is a CALL to the ADMIN_TASK_ADD stored procedure, which uses the POINT_IN_TIME and MAX_INVOCATIONS scheduling parameter to schedule an execution at 6:00 a.m. the next morning. Although ADMIN_TASK_ADD uses static input parameters for the scheduled stored procedure call, the use of transition variables in the trigger ensures that WLM_REFRESH is scheduled with the INSERTED WLM environment and DB2 subsystem ID. The required table that creates the triggering event is shown in Example A-29.

Example: A-29 DDL for the table for the trigger

```
CREATE TABLE USER.TAB_WLM_REFR
  ( ID             INTEGER GENERATED ALWAYS AS IDENTITY,
    WLM_ENV        VARCHAR(32),
    SSID           VARCHAR(4),
    TASKNAME       VARCHAR(128),
    TASK_RC        INTEGER,
    TASK_MSG       VARCHAR(1331))%
```

Only the two columns WLM_ENV and SSID are relevant for the WLM_REFRESH call. The TASKNAME column is populated by the trigger with the generic name WLM_REF_X, and is also used for the TASK_NAME parameter when the WLM_REFRESH call is scheduled. The additional columns with the prefix TASK_ are just employed for the output parameter handling of the called ADMIN_TASK_ADD stored procedure.

The TRIGGER that schedules a CALL to WLM_REFRESH could be set up as in Example A-30.

Example: A-30 Scheduling trigger

```
CREATE TRIGGER USER.TR_WLM_REFRESH
  NO CASCADE BEFORE INSERT ON USER.TAB_WLM_REFR
  REFERENCING NEW AS NEWWLM
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC

    SET NEWWLM.TASKNAME = CONCAT('WLM_REF_',STRIP(CHAR(NEWWLM.ID)));
    CALL SYSPROC.ADMIN_TASK_ADD('MYUSERID',
                                'MYPASSWD',
                                NULL,
                                NULL,
                                1,
                                NULL,
                                '0 6 * * *',
                                NULL,
                                NULL,
                                NULL,
                                NULL,
                                'SYSPROC',
                                'WLM_REFRESH',
                                'SELECT ''' || NEWWLM.WLM_ENV || ''',
                                        ''' || NEWWLM.SSID    || ''',
                                        ''OUT1'', 1 FROM SYSIBM.SYSDUMMY1',
                                NULL,
                                NULL,
                                NULL,
                                NEWWLM.TASKNAME,
                                'SCHEDULE WLM REFRESH JOB',
                                NEWWLM.TASK_RC,
                                NEWWLM.TASK_MSG);
```

```
END%
```

Note that the USERID and PASSWORD scheduling parameters need to be populated with valid credentials before ultimately employing this example. Furthermore, be aware that the creator of the scheduled task is the ID that executes the triggering insert event.

The following INSERT statement triggers the CALL to ADMIN_TASK_ADD to schedule a refresh of WLM environment WLMENV1 on the DB2 subsystem V91A at 6:00 a.m. the next morning.

```
INSERT INTO USER.TAB_WLM_REFR (WLM_ENV, SSID)
            VALUES( 'WLMENV1', 'V91A');
```

The first row that is inserted into the table schedules a task, with the task name WLM_REF_1, where the identity column of the table is employed to create unique task names.

Generally, it is good practice to remove tasks from the scheduler task lists as soon as they are no longer required. The trigger shown in Example A-31 calls ADMIN_TASK_REMOVE to remove the WLM_REFRESH task from the scheduler that is associated with the deleted row in the table.

Example: A-31 Trigger calling ADMIN_TAK_REMOVE

```
CREATE TRIGGER USER.TR_WLM_REFR_REMOVE
   NO CASCADE BEFORE DELETE ON USER.TAB_WLM_REFR
   REFERENCING OLD AS REMWLM
   FOR EACH ROW MODE DB2SQL
   BEGIN ATOMIC
     CALL SYSPROC.ADMIN_TASK_REMOVE(
                          REMWLM.TASKNAME,
                          REMWLM.TASK_RC,
                          REMWLM.TASK_MSG);
END%
```

Due to the 1-1 mapping of the TASKNAME column in the USER.TAB_WLM_REFR and the TASK_NAME in the scheduler, the associated scheduler task can be removed with the help of the transition variable REMWLM.TASKNAME. An SQL statement like the following would remove the task WLM_REF_1 from the scheduler:

```
DELETE FROM USER.TAB_WLM_REFR WHERE TASKNAME = 'WLM_REF_1';
```

A.9.3 Use case - 3

Schedule a non-regularly recurring stored procedure with dynamic parameters

In this use case an administrator seeks to refresh a WLM environment every day at 7:00 a.m. except for Saturdays and Sundays. Here a scheduler can be employed to schedule a non-regularly but recurring invocation of the DB2-supplied WLM_REFRESH stored procedure. In the following example, the POINT_IN_TIME scheduling parameter is used because it provides the flexibility to define the sought schedule. The END_TIMESTAMP parameter is set to December 31 to automatically invalidate the task at the end of the year. The sample furthermore schedules the stored procedure with dynamic input parameters, because the WLM environment that needs to be refreshed might change during the lifetime of the task. This ensures that WLM_REFRESH is always called with the latest content of the predefined USER.INPUT_PARMS table. The string provided in the PROCEDURE_INPUT scheduling parameter actually is a hybrid of dynamic and static input parameters. Whereas

the input parameters for WLM_REFRESH are dynamic, the output parameters are provided in a static way by making use of literals. The sample invocation for the described use case is listed in Example A-32.

Example: A-32 Sample invocation of non-regularly recurring procedure with dynamic parameters

```
      cs = con.prepareCall( "CALL SYSPROC.ADMIN_TASK_ADD("
             + "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? )" );

      /* provide the authid */
      cs.setString(1, "MYUSERID");                  // USERID
      /* provide the password */
      cs.setString(2, "MYPASSWD");                  // PASSWORD
      /* Task is immediately valid */
      cs.setNull(3, Types.TIMESTAMP);               // BEGIN_TIMESTAMP

      /* The task should be invalidated at end of the year */
      cs.setTimestamp( 4, Timestamp.valueOf( "2007-12-31 23:59:00.000000000") ); //
END_TIMESTAMP
      /* No limit for number of invocations required */
      cs.setNull( 5, Types.INTEGER);                // MAX_INVOCATIONS
      /* No interval required */
      cs.setNull( 6, Types.INTEGER);                // INTERVAL
      /* Every morning at 7AM except for Saturdays and Sundays */
      cs.setString( 7, "0 7 * * 1-5");              // POINT_IN_TIME

      /* No task triggering */
      cs.setNull( 8, Types.VARCHAR);                // TRIGGER_TASK_NAME
      cs.setNull( 9, Types.CHAR);                   // TRIGGER_TASK_COND
      cs.setNull(10, Types.INTEGER);                // TRIGGER_TASK_CODE

      /* No member affinity required */
      cs.setNull(11, Types.VARCHAR);                // DB2_SSID

      /* provide stored procedure information */
      cs.setString(12, "SYSPROC");                 // PROCEDURE_SCHEMA
      cs.setString(13, "WLM_REFRESH");             // PROCEDURE_NAME
      cs.setString(14, "SELECT WLM_ENV, SSID, 'OUT1', 1 FROM USER.INPUT_PARMS");

      /* JCL parameter not required */
      cs.setNull(15, Types.VARCHAR);                // JCL_LIBRARY
      cs.setNull(16, Types.VARCHAR);                // JCL_MEMBER
      cs.setNull(17, Types.VARCHAR);                // JOB_WAIT

      /* Provide a unique task name */
      cs.setString(18, "REFRESH_WLM");              // TASK_NAME
      cs.registerOutParameter(18, Types.VARCHAR);   // TASK_NAME

      /* Provide a task description */
      cs.setString(19, "Scheduled WLMENV refresh"); // DESCRIPTION

      /* Register out parameters */
      cs.registerOutParameter(20, Types.INTEGER);   // RETURN CODE
      cs.registerOutParameter(21, Types.VARCHAR);   // MESSAGE

      /* Schedule task */
      cs.execute();
```

The schema of the table referenced by the PROCEDURE_INPUT scheduling parameter could be similar like this:

```
CREATE TABLE USER.INPUT_PARMS
   ( ID             INTEGER GENERATED ALWAYS AS IDENTITY,
     WLM_ENV        VARCHAR(32),
     SSID           VARCHAR(4));
```

If the SELECT statement returns multiple rows, the scheduler will use the first row retrieved to populate the WLM_REFRESH input parameters. In order to invoke the scheduled stored procedure with different input parameter values the next time it is executed, the first qualifying row of the SELECT statement has to be updated. To avoid confusion, the SELECT statement should generally return only one row or alternatively employ the ORDER BY clause, for example:

```
SELECT WLM_ENV, SSID from USER.INPUT_PARMS where ID = 1;
```

A.9.4 Use case - 4

Stored procedure triggered by another stored procedure event

The REORG job created in use case 1 is defined with the LOG NO option. After execution, the sample table space DSN8D81A.DSN8S81D is therefore put into a COPY PENDING status. To remove this status, an administrator would have to run, for example, the COPY utility on the table space. Running this utility should also happen in the nightly maintenance window. It thus would make sense to run the COPY job immediately after the REORG job finishes. An administrator therefore employs the scheduler to schedule a CALL to DSNUTILU, which is triggered by the REORG job created in use case 1.

The scenario requires that TRIGGER_TASK_NAME is properly set to the existing REORG job. All other scheduling parameters can be initialized with NULL. Furthermore, DSNUTILU should be executed even if the triggering REORG job did not run successfully. Therefore, the TRIGGER_TASK_COND and TRIGGER_TASK_CODE scheduling parameters are set to NULL as well. Similar to use case 1, PROCEDURE_INPUT contains static values for the parameters.

The sample code in Example A-33 shows the proper initialization of the ADMIN_TASK_ADD stored procedure to schedule a DSNUTILU call that invokes the COPY utility. This call is triggered by the execution of the existing REORG job REORG_JOB.

Example: A-33 initialization of the ADMIN_TASK_ADD stored procedure

```
cs = con.prepareCall( "CALL SYSPROC.ADMIN_TASK_ADD("
            + "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? )" );

    /* provide the authid */
    cs.setString(1, "MYUSERID");                  // USERID
    /* provide the password */
    cs.setString(2, "MYPASSWD");                  // PASSWORD
    /* No begin_timestamp, this task is triggered by another task */
    cs.setNull(3, Types.TIMESTAMP);               // BEGIN_TIMESTAMP

    /* This is a one time task, no end timestamp required */
    cs.setNull( 4, Types.TIMESTAMP);              // END_TIMESTAMP
    /* Triggered by every invocation of the triggering task */
    cs.setNull( 5, Types.INTEGER);                // MAX_INVOCATIONS
    /* No interval required */
    cs.setNull( 6, Types.INTEGER);                // INTERVAL
    /* No point in time required */
    cs.setNull( 7, Types.VARCHAR);                // POINT_IN_TIME
```

```
      /* The triggering task is the REORG job scheduled earlier */
      cs.setString( 8, "REORG_JOB");                // TRIGGER_TASK_NAME
      /* This task should be run after every invocation of the triggering task,
       * no matter what the returned SQLCODE is */
      cs.setNull( 9, Types.CHAR);                   // TRIGGER_TASK_COND
      cs.setNull(10, Types.INTEGER);                // TRIGGER_TASK_CODE

      /* No member affinity required */
      cs.setNull(11, Types.VARCHAR);                // DB2_SSID

      /* provide stored procedure information */
      cs.setString(12, "SYSPROC");                 // PROCEDURE_SCHEMA
      cs.setString(13, "DSNUTILU");                // PROCEDURE_NAME
      cs.setString(14,"SELECT 'COPY', 'NO', 'COPY TABLESPACE
                 DSN8D81A.DSN8S81D', 1 FROM SYSIBM.SYSDUMMY1");

      /* JCL parameter not required */
      cs.setNull(15, Types.VARCHAR);                // JCL_LIBRARY
      cs.setNull(16, Types.VARCHAR);                // JCL_MEMBER
      cs.setNull(17, Types.VARCHAR);                // JOB_WAIT

      /* Provide a unique task name */
      cs.setString(18, "COPY_JOB");                 // TASK_NAME
      cs.registerOutParameter(18, Types.VARCHAR);   // TASK_NAME

      /* Provide a task description */
      cs.setString(19, "Scheduled copy job");      // DESCRIPTION

      /* Register out parameters */
      cs.registerOutParameter(20, Types.INTEGER);   // RETURN CODE
      cs.registerOutParameter(21, Types.VARCHAR);   // MESSAGE

      /* Schedule task */

cs.execute();
```

A.9.5 Housekeeping with the scheduler

The code in Example A-34 illustrates a way to perform housekeeping with the scheduler. The program first joins the two table functions ADMIN_TASK_LIST and ADMIN_TASK_STATUS to get the names of the tasks that are no longer valid, that is, tasks with an END_TIMESTAMP value that is in the past or with a NUM_INVOCATIONS value that is equal to MAX_INVOCATIONS. In the fetch loop, the procedure ADMIN_TASK_REMOVE is then invoked on every TASK_NAME returned by the query.

Note the additional CREATOR predicate that is set to the calling user ID. This is necessary in the application program, because the table functions would otherwise return tasks that cannot be deleted by the following ADMIN_TASK_REMOVE call, because the calling user ID and the user ID that added the task do not match. This would return an error and the fetch loop would be aborted.

Example A-34 only returns and removes expired tasks that have been added with the user ID, which is also used for establishing the JDBC connection to the server.

Example: A-34 Task scheduler housekeeping

```
//*********************************************************************
```

```
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//******************************************************************************
// Source file name: AdminScheduleR.java
//
// Sample: How to use the scheduler to remove tasks from the task list that
//         are no longer valid
//
// The user runs the program by issuing:
// java AdminScheduleR <alias or //server/database> <userid> <password>
//
// The arguments are:
// <alias> - DB2 subsystem alias for type 2 or //server/database for type 4
//  connectivity
// <userid> - user ID to connect as
// <password> - password to connect with
//******************************************************************************
import java.sql.*;

public class AdminScheduleR
{
  public static void main(String args[])
  {
    Connection con = null;
    PreparedStatement ps = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;


    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: AdminScheduleR  <alias or //server/database> <userid>
<password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try
    {
      int rc = 0;
      String message = null;
      String task_name = null;

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);
```

```
        // Connect to database
        con = DriverManager.getConnection(url, userid, password);
        con.setAutoCommit(false);

        cs = con.prepareCall( "CALL SYSPROC.ADMIN_TASK_REMOVE(?, ?, ?)");
        ps = con.prepareStatement(
         "SELECT T.TASK_NAME "                                  +
         " FROM TABLE(DSNADM.ADMIN_TASK_LIST()) T, "            +
         "      TABLE(DSNADM.ADMIN_TASK_STATUS()) S "           +
         " WHERE T.TASK_NAME = S.TASK_NAME AND "                +
         "       (S.NUM_INVOCATIONS = T.MAX_INVOCATIONS OR " +
         "        T.END_TIMESTAMP < CURRENT TIMESTAMP) AND " +
         "       S.STATUS <> 'RUNNING' AND T.CREATOR = ?; ");

        ps.setString(1, userid);
        rs = ps.executeQuery();

        while ( rs.next() ) {
          task_name = rs.getString(1);
          System.out.println("Remove Task_Name: " + task_name);

          //ADMIN_TASK_REMOVE parameters
          cs.setString(1, task_name);                  // TASK_NAME
          cs.registerOutParameter(2, Types.INTEGER);   // RETURN CODE
          cs.registerOutParameter(3, Types.VARCHAR);   // MESSAGE
          // Remove task
          cs.execute();
          con.commit();

          rc = cs.getInt(2);
          if (rc == 0)
          {
             /* Task successfully scheduled */
             System.out.println("Task " + task_name + " successfully removed");
          }
          else
          {
             /* Error during task scheduling */
             message = cs.getString(3);
             throw new AdminScheduleException(rc,
                   "ADMIN_TASK_REMOVE execution failed: " + message);
          }
        }
      }
      catch (AdminScheduleException ase)
      {
        System.err.println("Program error: rc=" + ase.getRC() + " message=" +
ase.getMessage());
      }
      catch (ClassNotFoundException e)
      {
        System.err.println("Could not load JDBC driver");
        System.err.println("Exception: " + e);
        e.printStackTrace();
      }
      catch (SQLException sqle)
      {
        System.err.println("SQLException information");
        System.err.println("Error msg: " + sqle.getMessage());
        System.err.println("SQLSTATE: " + sqle.getSQLState());
```

```
      System.err.println("Error code: " + sqle.getErrorCode());
    }
    finally
    {
      // Release resources and disconnect
      try
      {
        cs.close();
      } catch (Exception e){}
      try
      {
        con.close();
      } catch (Exception e){}
    }
  }
}

class AdminScheduleException extends Exception
{
  private int rc;

  AdminScheduleException(int rc, String message){
    super(message);
    this.rc = rc;
  }

  public int getRC(){
    return rc;
  }
}
```

Compile AdminScheduleR.java and enter the following command to execute it:

```
java AdminScheduleR DBALIAS USERID PASSWORD
```

Example A-35 shows the output of the ADMIN_TASK_LIST table function. All listed tasks feature an execution status different than RUNNING.

Example: A-35 ADMIN_TASK_LIST output

```
---------+--------------+---------+---------+---------+---------+--
TASK_NAME                      END_TIMESTAMP               CREATOR
---------+--------------+---------+---------+---------+---------+--
TASK_ID_0001                   2008-12-22-10.35.00.000000  SYSADM
TASK_ID_0002                   2008-09-22-10.35.00.000000  SYSADM
TASK_ID_0003                   2008-12-22-10.35.00.000000  SYSADM
TASK_ID_0012                   2007-12-29-12.55.00.000000  PAOLOR3
Reorg Job 1                    2007-12-29-12.55.00.000000  PAOLOR2
Reorg Job                      2007-12-29-04.10.00.000000  PAOLOR2
Runstats Job                   2007-12-29-04.15.00.000000  PAOLOR3
DSNE610I NUMBER OF ROWS DISPLAYED IS 7
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
```

Invoking AdminScheduleR on December,29, 2007 at 05:00 with a user ID = PAOLOR3 would have resulted in the following output:

```
Remove Task_Name: Runstats Job
Task Runstats Job successfully removed
```

Although the task Reorg Job can be considered as expired, the CREATOR predicate is not satisfied here, thus this task is not removed.

A.10 Invoking the Common SQL API stored procedures

The following comprehensive Java sample calls the Common SQL API stored procedures SYSPROC.GET_SYSTEM_INFO, GET_CONFIG, and GET_MESSAGE. It illustrates how parameter handling is properly done and employs the Jakarta Commons Configuration implementation of an XML-PropertyList to easily interface with the XML documents that are employed as stored procedure parameters.

The program basically consists of two classes: The class SPDriver.java contains the main method and instantiates an object of the class SPWrapper.java that provides the service routines to eventually CALL the desired stored procedure.

Example A-37 shows the first part of the class SPDriver. It contains the definition of the XML documents that are involved and also a method that gets a Connection object, which will be used to CALL the respective stored procedures. Be aware that the stored procedure GET_SYSTEM_INFO is invoked first in Complete Mode. The program assumes that the XML_COMPLETE_MODE document exists in the search path. Therefore, create a file named CompleteMode.xml and insert the following valid XML document. See Example A-36.

Example: A-36 CompleteMode.xml document

```
<?xml version="1.0"  encoding="UTF-8"?>
<plist version="1.0">
<dict>
   <key>Complete</key><true/>
</dict>
</plist>
```

Furthermore, create the XML_FILTER_DOC XML document and insert a dummy string, for example `this is not a valid XPATH`.

Example: A-37 SPDriver.java - part 1

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

//****************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//****************************************************************************
// Source file name: SPDriver
//
// Sample: This sample calls the stored procedure SYSPROC.GET_SYSTEM_INFO
//         first in Complete-Mode to retrieve a template XML input document.
//         Then the XMLPropertyListConfiguration classes are used to verify
//         the returned XML document and to insert the required input data.
//         A second CALL employs then the augmented XML document to obtain
//         the desired data.
//         A third CALL is issued against SYSPROC.GET_CONFIG with an invalid
//         XML_FILTER, the returned positive SQLCODE warning and message
//         tokens are then wrapped into an XML_INPUT document for a
```

```
//         following GET_MESSAGE CALL to obtain the respective short
//         message text.
//         This class instantiates an object of class SPWrapper.
//
//The user runs the program by issuing:
//java SPDriver <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
//<password> - password to connect with
//*********************************************************************

public class SPDriver {

   // Connection information
   private static String url = "jdbc:db2:";
   private static final String driver = "com.ibm.db2.jcc.DB2Driver";
   // Stored procedure information
   private static final String GS = "SYSPROC.GET_SYSTEM_INFO";
   private static final String GM = "SYSPROC.GET_MESSAGE";
   private static final String GC = "SYSPROC.GET_CONFIG";
   // Document information
   private static final String XML_COMPLETE_MODE = "CompleteMode.xml";
   private static final String XML_INPUT_DOC_GS = "XML_Input_gs.xml";
   private static final String XML_INPUT_TEMPLATE_GS = "XML_TemplateIN_gs.xml";
   private static final String XML_OUTPUT_DOC_GS = "XML_Output_gs.xml";
   private static final String XML_INPUT_DOC_GM = "XML_Input_gm.xml";
   private static final String XML_OUTPUT_DOC_GM = "XML_Output_gm.xml";
   private static final String XML_OUTPUT_DOC_GC = "XML_Output_gc.xml";
   private static final String XML_FILTER_DOC = "XML_Filter.xml";
   private static final String XML_MESSAGE_DOC = "XML_Message.xml";


   private static Connection getConnection(String userid, String password) {
      // Setup connection
      Connection con = null;
      //Load the DB2 Universal JDBC Driver
      try {
         Class.forName(driver);

         // Connect to database
         con = DriverManager.getConnection(url, userid, password);
         con.setAutoCommit(false);
      }
      catch (ClassNotFoundException cnfe) {
         System.err.println("Could not load JDBC driver");
         System.err.println("Exception: " + cnfe);
         cnfe.printStackTrace();
         return null;
      } catch (SQLException sqle) {
         System.err.println("SQLException information");
         System.err.println("Error msg: " + sqle.getMessage());
         System.err.println("SQLSTATE: " + sqle.getSQLState());
         System.err.println("Error code: " + sqle.getErrorCode());
         return null;
      }
      return con;
```

```
}
```

The main method first of all parses the provided arguments, obtains a Connection object, and then instantiates an Object of the class SPWrapper. The calling sequence which is driven by the main method is as follows:

- CALL SYSPROC.GET_SYSTEM_INFO in Complete Mode and materialize the XML template input document returned in the XML_OUTPUT parameter in `XML_INPUT_TEMPLATE_GS`.
- Augment the returned XML template input document with a valid SMPCSI data set name and add two SYSMODs to the respective sections.

  In the sample provided, the SMPCSI data set name is `IEL350.GLOBAL.CSI`, the names of the SYSMODs are: `AK16335, HOA4220.` The name of the new XML input document is specified in `XML_INPUT_DOC_GS.`
- CALL SYSPROC.GET_SYSTEM_INFO a second time with the augmented and valid XML input document.
- Materialize the complete XML_OUTPUT document that now contains the gathered system-related data and additionally information about the queried SYSMODs. The XML output document is written to `XML_OUTPUT_DOC_GS.`
- CALL SYSPROC.GET_CONFIG with an invalid XPath in the XML_FILTER input parameter document. As expected, the CALL will fail with a SQLCODE of +20458. The document containing the filter is defined in `XML_FILTER_DOC.`
- To obtain the short message text associated to +20458, the stored procedure GET_MESSAGE is invoked with an XML_INPUT document specifying the +20458 SQLCODE as well as the message tokens returned in the SQLERRMC structure. The input document is materialized in `XML_INPUT_DOC_GM`:

  ```
  CALL SYSPROC.GET_MESSAGE
  ```

  The short message returned in the GET_MESSAGE XML output document text lists the following explanation:

  ```
  Short Message Text for SQLCODE: 20458:

  DSNT404I SQLCODE = 20458, WARNING: THE PROCEDURE GET_CONFIG HAS ENCOUNTERED AN
  INTERNAL PARAMETER PROCESSING ERROR IN PARAMETER 5. THE VALUE FOR PARAMETER 7
  CONTAINS FURTHER INFORMATION ABOUT THE ERROR.
  ```

- According to this message text, the CALL to SYSPROC.GET_CONFIG returned a XML_MESSAGE output document which should contain further debug information. Taking a look at the materialized `XML_MESSAGE_DOC` shows the following reasonable explanation:

  ```
  DSNA630I  DSNADMGC A PARAMETER FORMAT OR CONTENT ERROR WAS FOUND. Invalid XPath
  expression for filtering the output document. Found string beginning with 'this
  is not a valid XPATH' at position 1.
  ```

Example A-38 contains the remainder of the class SPDriver.java.

Example: A-38 SPDriver.java - part 2

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
public static void main(String[] args) {
```

```
      SPWrapper spDriver = null;
      Connection dbCon = null;
      String userid = null;
      String password = null;
      int callRC = 0;
      int sqlcode = 0;

      // Parse arguments
      if (args.length != 3) {
         System.err
               .println("Usage: Get_Message <alias or //server/database> <userid>
<password>");
         System.err
               .println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
         System.err.println("       <userid> is user ID to connect as");
         System.err.println("       <password> is password to connect with");
         return;
      }
      url += args[0];
      userid = args[1];
      password = args[2];

      // Get a connection object
      dbCon = getConnection(userid, password);
      if (dbCon == null)
         return;

      // Create class object
      spDriver = new SPWrapper(dbCon);
      // Define intput parameters
      spDriver.setMajorMinorEn("1", "0", "en_US");
      spDriver.setInputDocs(XML_COMPLETE_MODE, "null");

      // CALL stored procedure
      callRC = spDriver.callSP(GS, XML_INPUT_TEMPLATE_GS, XML_MESSAGE_DOC);
      sqlcode = spDriver.getSQLCODE();
      if ( callRC < 0)
         System.err.println("A program error occurred");
      else if (sqlcode < 0)
         System.err.println("An error occurred during the 1st CALL");
      else if (sqlcode == 20458 || sqlcode == 20459)
        System.out.println("2nd SP CALL returned a XML_MESSAGE document");
      else if (sqlcode == 0|| sqlcode == 20460 || sqlcode == 20461)
      {
         System.out.println("1st SP CAll successful\n");

         // Augment template input document
         spDriver.defineInputData(XML_INPUT_TEMPLATE_GS, XML_INPUT_DOC_GS);

         spDriver.setMajorMinorEn("1", "0", "en_US");
         spDriver.setInputDocs(XML_INPUT_DOC_GS, "null");

         // Call GET_SYSTEM_INFO with new XML_INPUT document
         callRC = spDriver.callSP(GS, XML_OUTPUT_DOC_GS, XML_MESSAGE_DOC);
         sqlcode = spDriver.getSQLCODE();
         if ( callRC < 0)
            System.err.println("A program error occurred");
         else if (sqlcode < 0)
            System.err.println("An error occurred during the 2nd CALL");
```

```
            else if (sqlcode == 20458 || sqlcode == 20459)
               System.out.println("2nd SP CALL returned a XML_MESSAGE document");
            else if (sqlcode == 0 || sqlcode == 20460 || sqlcode == 20461) {
               System.out.println("2nd Call successful\n");
            }
         }

         // Construct a CALL to GET_CONFIG that result in a negative SQLCODE,
         // invoke GET_MESSAGE on the SQLCODE and the returned message tokens
         spDriver.setMajorMinorEn("1", "0", "en_US");
         spDriver.setInputDocs("null", XML_FILTER_DOC);

         // CALL will fail due to invalid major/minor version combination
         callRC = spDriver.callSP(GC, XML_OUTPUT_DOC_GC, XML_MESSAGE_DOC);
         sqlcode = spDriver.getSQLCODE();
         if ( callRC < 0)
            System.err.println("A program error occurred");
         else if (sqlcode != 0)
         {
            System.err.println("An error occurred during the 3rd CALL");

            // Invoke GET_MESSAGE on the SQLCODE and the message tokens
            // returend. Construct an input document from scratch
            if(!spDriver.create_Msg_Input(XML_INPUT_DOC_GM))
               System.out.println("A program error occurred");

            spDriver.setMajorMinorEn("1", "0", "en_US");
            spDriver.setInputDocs(XML_INPUT_DOC_GM, "null");
            callRC = spDriver.callSP(GM, XML_OUTPUT_DOC_GM, XML_MESSAGE_DOC);
            if ( callRC < 0)
               System.err.println("A program error occurred");
            else if (spDriver.getSQLCODE() != 0)
               System.err.println("An error occurred during the 4th CALL");
            else
               System.out.println("Short Message Text for SQLCODE: " + sqlcode +
                  ": \n" + spDriver.getShortMessageText(XML_OUTPUT_DOC_GM));
         }
         try {
            dbCon.close();
         } catch (Exception e) {}
      }
   }
```

The class SPWrapper.java provides the functionality to eventually CALL the stored procedures and process the input and output parameters.

All XML documents are passed as BLOB to and from the stored procedures. The input documents are read from a file and the respective InputStream is then directly used as input parameter, for invoking cstmt.setBinaryStream(). Refer to the function callSP(). The output BLOBs are materialized to files as well, by using byte arrays. Refer to function saveBlob2File() as a sample.

As mentioned above, the program employs the Jakarta Commons Configuration. It makes use of the XMLPropertyListConfiguration interface implementation. This requires the following JAR files to be contained in the search path:

- commons-configuration.jar
- commons-codec.jar
- commons-beanutils.jar

- commons-digester.jar
- commons-collections.jar
- commons-logging.jar
- commons-lang.jar

For detailed information, refer to:

http://commons.apache.org/configuration/

The Java sample employs the XMLPropertyListConfiguration interface in multiple XML_INPUT and XML_OUTPUT processing scenarios. The following functions show sample implementations:

`verificationPropertyList()` - uses XMLPropertyListConfiguration to verify if the returned XML output document is valid in terms of the PList DTD.

`defineInputData()` - this function augments the returned GET_SYSTEM_INFO XML template input document with the SMPCSI data set and SYSMOD names. It therefore creates an XMLPropertyListConfiguration object from an existing XML PList document and uses the provided interface to update existing key / value pairs. This function also demonstrates how to create <array> elements in the XML document.

`create_Msg_Input()` - here a valid XML input document for a GET_MESSAGE call is created from scratch. A new XMLPropertyListConfiguration object is created and the required key / value pairs are manually added. This function also illustrates how to create key / values pair groupings with different nesting levels.

`getShortMessageText()` - shows how easily the value of a distinct key / value pair can be accessed by employing the XMLPropertyListConfiguration interface.

Example A-39 shows the SPWrapper code.

Example: A-39 SPWrapper.java

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: SPWrapper
//
// Sample: This class is instantiated by the class SPDriver and provides
//         multiple services to work with the CSA stored procedures.
//
//*************************************************************************

import java.io.*;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.plist.XMLPropertyListConfiguration;

public class SPWrapper {

   // XMLPropertyList
   private final String CSI_KEY_NAME = "Optional Parameters.SMPCSI Data Set.Value";
   private final String CSI_KEY_VALUE = "IEL350.GLOBAL.CSI";
   private final String SYSMOD_KEY_NAME = "Optional Parameters.SYSMOD.Value";
```

```
private final String SYSMOD_KEY_VALUE1 = "AK16335";
private final String SYSMOD_KEY_VALUE2 = "HOA4220";

// CALL / SP input information
private String major_version = "1";
private String minor_version = "0";
private String requested_local = "en_US";
private String xml_Input = null;
private String xml_Filter = null;

private int SQLCODE = 0;
private String SQLSTATE = "";
private ArrayList msgTokens = null;

private Connection dbCon = null;

// Constructor
public SPWrapper(Connection con) {
   dbCon = con;
   msgTokens = new ArrayList();
}

public void setMajorMinorEn(String ma, String mi, String en) {
   major_version = ma;
   minor_version = mi;
   requested_local = en;
}

public void setInputDocs(String in, String fil) {
   xml_Input = in;
   xml_Filter = fil;
}

public int getSQLCODE() {
   return SQLCODE;
}

private static InputStream getInputStream(String filename) {
   InputStream xmlFile = null;
   try {
      xmlFile = new FileInputStream(filename);
   } catch (FileNotFoundException fne) {
      System.err.println("File " + filename
            + " could not be found");
      System.err.println("Exception: " + fne);
      fne.printStackTrace();
      return null;
   }
   return xmlFile;
}

private void chopTokens(String messageTxt)
{
   // index of first token:
   int index = messageTxt.indexOf("SQLERRMC:") + 10;
   String tokens[] = messageTxt.substring(index, messageTxt.length()).split(";");
   msgTokens.clear();
   for(int i = 0; i < tokens.length; i++)
      msgTokens.add(tokens[i]);
}
```

```
private void saveBlob2File(Blob blob, String fn) {
   PrintWriter pw = null;
   String xmlInString = null;
   long xmlInLength = 0;
   byte xmlInByte[] = null;
   try {
      // Open OutputStream to save the XML document
      pw = new PrintWriter(new File(fn), "utf-8");

      // Get String representation of the returned BLOB
      xmlInLength = blob.length();
      xmlInByte = blob.getBytes((long) 1, (int) xmlInLength);
      xmlInString = new String(xmlInByte, "utf-8");

      // Write to OutputStream
      pw.write(xmlInString);
      pw.close();
   } catch (IOException ioe) {
      System.err.println("IOException: " + ioe);
      ioe.printStackTrace();
   }
   catch (SQLException sqle) {
      System.err.println("SQLException information");
      System.err.println("Error msg: " + sqle.getMessage());

      // init global debug variables
      SQLCODE = sqle.getErrorCode();
      SQLSTATE = sqle.getSQLState();
      chopTokens(sqle.getMessage());
   }
   finally {
      try {
         pw.close();
      } catch (Exception e) {}
   }
}

public boolean create_Msg_Input(String inputFn) {
   PrintWriter pw = null;
   // Manually create a XML_INOUT document for GET_MESSAGE
   try {
      XMLPropertyListConfiguration configPlist =
         new XMLPropertyListConfiguration();

      configPlist.addProperty("Document Type Name",
                              "Data Server Message Input");
      configPlist.addProperty("Document Type Major Version",
            new Integer(1));
      configPlist.addProperty("Document Type Minor Version",
            new Integer(0));
      configPlist.addProperty("Document Locale",
                              "en_US");

      configPlist.addProperty("Required Parameters.Display Name",
                              "Required Parameters");
      configPlist.addProperty("Required Parameters.SQLCODE.Display Name",
                              "SQLCODE");
      configPlist.addProperty("Required Parameters.SQLCODE.Value",
                              new Integer(SQLCODE));
```

```
      configPlist.addProperty("Optional Parameters.Display Name",
                              "Optional Parameters");
      configPlist.addProperty("Optional Parameters.Message Tokens.Display Name",
      "Message Tokens");

      // Add the array section to the XML input document
      configPlist.addProperty("Optional Parameters.Message Tokens.Value",
      msgTokens);

      pw = new PrintWriter(new File(inputFn),"utf-8");
      configPlist.save(pw);

   }catch (ConfigurationException ce) {
      ce.printStackTrace();
      return false;
   } catch (FileNotFoundException fne) {
      fne.printStackTrace();
   } catch (UnsupportedEncodingException uee) {
      uee.printStackTrace();
   }
   return true;
}

public String getShortMessageText(String fn) {
   String shortMSGTxt = null;
   try {

      // Verify the returned PropertyList
     XMLPropertyListConfiguration configPlist =
           new XMLPropertyListConfiguration(fn);

     shortMSGTxt = (String)configPlist.getProperty(
                   "Short Message Text.Value");

   } catch (ConfigurationException ce) {
      ce.printStackTrace();
   }
   return shortMSGTxt;
}

private boolean verificationPropertyList(String filepath) {
   try {
      // Verify the returned PropertyList
     XMLPropertyListConfiguration configPlist =
           new XMLPropertyListConfiguration(filepath);
   } catch (ConfigurationException ce) {
      ce.printStackTrace();
      return false;
   }
   return true;
}

public void defineInputData(String template, String newInput)
{
   PrintWriter pw = null;
   try {
      XMLPropertyListConfiguration configPlist =
         new XMLPropertyListConfiguration(template);
      // Insert the SMPCSI data set name into the template
```

```
            configPlist.setProperty(CSI_KEY_NAME, CSI_KEY_VALUE);

            // Delete template array element
            configPlist.clearProperty(SYSMOD_KEY_NAME);

            // Construct ArrayList for SYSMOD section
            List sysmods = new ArrayList();
            sysmods.add(SYSMOD_KEY_VALUE1);
            sysmods.add(SYSMOD_KEY_VALUE2);

            // Add the array section to the XML input document
            configPlist.addProperty(SYSMOD_KEY_NAME, sysmods);

            //Construct a local PrintWriter object
            pw = new PrintWriter(new File(newInput),"utf-8");
            // Write Plist to file
            configPlist.save(pw);
         } catch (UnsupportedEncodingException uee) {
            uee.printStackTrace();
         } catch (FileNotFoundException fne) {
            fne.printStackTrace();
         } catch (ConfigurationException ce) {
            ce.printStackTrace();
         }
         finally {
            try {
               pw.close();
            } catch (Exception e) {}
         }
      }

      public int callSP(String proc_name,
                        String outputFn,
                        String messageFn) {
         // Prepare stored procedure CALL
         CallableStatement cstmt = null;
         InputStream xmlInput = null;
         InputStream xmlFilter = null;
         Blob output_Blob = null;
         Blob message_Blob = null;
         String call_statement = "CALL " + proc_name +
         " (?,?,?,?,?,?,?)";

         try {
            cstmt = dbCon.prepareCall(call_statement);

            // Register output parameters
            cstmt.registerOutParameter(1, java.sql.Types.INTEGER); // MAJOR_VERSION
            cstmt.registerOutParameter(2, java.sql.Types.INTEGER); // MINOR_VERSION
            cstmt.registerOutParameter(6, java.sql.Types.BLOB); // XML_OUTPUT
            cstmt.registerOutParameter(7, java.sql.Types.BLOB); // XML_MESSAGE

            // Set input parameters
            if (major_version.trim().equalsIgnoreCase("null")) {
               cstmt.setNull(1, java.sql.Types.INTEGER);
            } else {
               cstmt.setInt(1, Integer.parseInt(major_version));
            }
            if (minor_version.trim().equalsIgnoreCase("null")) {
               cstmt.setNull(2, java.sql.Types.INTEGER);
```

```
} else {
   cstmt.setInt(2, Integer.parseInt(minor_version));
}
if (requested_local.trim().equalsIgnoreCase("null")) {
   cstmt.setNull(3, java.sql.Types.VARCHAR);
} else {
   cstmt.setString(3, requested_local);
}
if (xml_Input.trim().equalsIgnoreCase("null")) {
   cstmt.setBinaryStream(4, null, 0);
}
else {
   // Read XML input document from file
   xmlInput = getInputStream(xml_Input);
   if (xmlInput == null)
      return -1;
   int xmlInputFileSize = xmlInput.available();
   cstmt.setBinaryStream(4, xmlInput, xmlInputFileSize);
}
if (xml_Filter.trim().equalsIgnoreCase("null")) {
   cstmt.setBinaryStream(5, null, 0);
}
else {
   // Read XML input document from file
   xmlFilter = getInputStream(xml_Filter);
   if (xmlFilter == null)
      return -1;
   int xmlFilterFileSize = xmlFilter.available();
   cstmt.setBinaryStream(5, xmlFilter, xmlFilterFileSize);
}
// Trace input parameters
System.out.println("Calling " + proc_name + " with parameters:");
System.out.println("MAJOR_VERSION - " + major_version);
System.out.println("MINOR_VERSION - " + minor_version);
System.out.println("REQUESTED_LOCALE - " + requested_local);
System.out.println("XML_INPUT file - " + xml_Input);
System.out.println("XML_FILTER file - " + xml_Filter);

// CALL stored procedure
cstmt.execute();
SQLWarning cstmt_warning = cstmt.getWarnings();
if (cstmt_warning != null) {
   System.err.println("SQL Warning: " + cstmt_warning.getMessage());

   // init global debug variables
   SQLCODE = cstmt_warning.getErrorCode();
   SQLSTATE = cstmt_warning.getSQLState();
   chopTokens(cstmt_warning.getMessage());
} else {
   System.out.println("SQL Warning: None\n");
   // Reset debug variables
   SQLCODE = 0;
   SQLSTATE = "";
   msgTokens.clear();
}

// Get output parameters
System.out.println("Output parameters: ");
System.out.println("Major Version returned: " + cstmt.getInt(1));
System.out.println("Minor Version returned: " + cstmt.getInt(2));
```

```
   /* XML_OUTPUT document */
   if (!outputFn.trim().equalsIgnoreCase("null")) {
      output_Blob = cstmt.getBlob(6);
      if (output_Blob == null) {
         System.out.println("XML_OUTPUT: NULL");
      } else {
         saveBlob2File(output_Blob, outputFn);
         if (output_Blob.length() == 0) {
            System.out.println("XML_OUTPUT: Empty\n");
         } else if (xml_Filter.trim().equalsIgnoreCase("null")) {
            // Validate only if XML_OUTPUT not null and XML_FILTER null
            System.out.println("Common Configuration validation result: ");
            // Verify the returned Plist XML document
            if (verificationPropertyList(outputFn)) {
               System.out.println(
                 "XML_OUTPUT complies with plist version 1.0.\n");
            } else {
               System.err.println(
                     "XML_OUTPUT does not comply with plist version 1.0.\n");
               return -1;
            }
         }
      }
   }
   // XML_MESSAGE document
   if(!messageFn.trim().equalsIgnoreCase("null")) {
      message_Blob = cstmt.getBlob(7);
      if (message_Blob == null) {
         System.out.println("XML_MESSAGE: NULL");
      } else {
         saveBlob2File(message_Blob, messageFn);
         if (message_Blob.length() == 0) {
            System.out.println("XML_MESSAGE: Empty\n");
         } else {
            System.out.println("Common Configuration validation result: ");
            // Verify the returned Plist XML document
            if (verificationPropertyList(messageFn)) {
               System.out.println(
                     "XML_MESSAGE complies with plist version 1.0.\n");
            }
            else {
               System.err.println(
                     "XML_MESSAGE does not comply with plist version 1.0.\n");
               return -1;
            }
         }
      }
   }
}
catch (SQLException sqle) {
   System.err.println("SQLException information");
   System.err.println("Error msg: " + sqle.getMessage());
   // init global debug variables
   SQLCODE = sqle.getErrorCode();
   SQLSTATE = sqle.getSQLState();
   chopTokens(sqle.getMessage());
} catch (IOException ioe) {
   System.err.println("IOException: " + ioe);
   ioe.printStackTrace();
```

```
            return -1;
         }
         finally {
            try {
               cstmt.close();
               xmlInput.close();
               xmlFilter.close();
            } catch (Exception e) {}
         }
         return 0;
      }
}
```

Compile SPDriver.java and SPWrapper.java then enter the following command to execute it:

```
java SPDriver DBALIAS USERID PASSWORD
```

The output is shown in Example A-40.

Example: A-40 SPDriver.java output traces

```
Calling SYSPROC.GET_SYSTEM_INFO with parameters:
MAJOR_VERSION - 1
MINOR_VERSION - 0
REQUESTED_LOCALE - en_US
XML_INPUT file - CompleteMode.xml
XML_FILTER file - null
SQL Warning: None

Output parameters:
Major Version returned: 1
Minor Version returned: 0
Common Configuration validation result:
XML_OUTPUT complies with plist version 1.0.

XML_MESSAGE: NULL
1st SP CAll successful

Calling SYSPROC.GET_SYSTEM_INFO with parameters:
MAJOR_VERSION - 1
MINOR_VERSION - 0
REQUESTED_LOCALE - en_US
XML_INPUT file - XML_Input_gs.xml
XML_FILTER file - null
SQL Warning: None

Output parameters:
Major Version returned: 1
Minor Version returned: 0
Common Configuration validation result:
XML_OUTPUT complies with plist version 1.0.

XML_MESSAGE: NULL
2nd Call successful

Calling SYSPROC.GET_CONFIG with parameters:
MAJOR_VERSION - 1
MINOR_VERSION - 0
REQUESTED_LOCALE - en_US
XML_INPUT file - null
XML_FILTER file - XML_Filter.xml
```

```
SQL Warning: DB2 SQL error: SQLCODE: 20458, SQLSTATE: 01H54, SQLERRMC: GET_CONFIG;5;7
Output parameters:
Major Version returned: 1
Minor Version returned: 0
XML_OUTPUT: NULL
An error occurred during the 3rd CALL
Common Configuration validation result:
XML_MESSAGE complies with plist version 1.0.

Calling SYSPROC.GET_MESSAGE with parameters:
MAJOR_VERSION - 1
MINOR_VERSION - 0
REQUESTED_LOCALE - en_US
XML_INPUT file - XML_Input_gm.xml
XML_FILTER file - null
SQL Warning: None

Output parameters:
Major Version returned: 1
Minor Version returned: 0
Common Configuration validation result:
XML_OUTPUT complies with plist version 1.0.

XML_MESSAGE: NULL
Short Message Text for SQLCODE: 20458:
DSNT404I SQLCODE = 20458, WARNING: THE PROCEDURE GET_CONFIG HAS ENCOUNTERED AN INTERNAL
PARAMETER PROCESSING ERROR IN PARAMETER 5. THE VALUE FOR PARAMETER 7 CONTAINS FURTHER
INFORMATION ABOUT THE ERROR.
```

A.10.1 Simple GET_CONFIG invocation with a valid XPath

Example A-41 illustrates a very simple and static Java program that CALLs the stored procedure GET_CONFIG with an XPath that queries the value of the data server's IP address. The XPath is statically created as a String object by the program and then converted to a BLOB to serve as XML_FILTER input. After the CALL, XML_OUTPUT only contains a single string and no XML document. This output is materialized as a file called xml_output.xml in the same directory where the GetConfDriver class resides.

Example: A-41 GetConfigDriver.java

```
//*************************************************************************
// Licensed Materials - Property of IBM
// 5635-DB2
// (C) COPYRIGHT 1982, 2006 IBM Corp.  All Rights Reserved.
//
// STATUS = Version 9
//*************************************************************************
// Source file name: GetConfDriver.java
//
// Sample: How to call SYSPROC.GET_CONFIG with a valid XPath to extract the
//         IP Address.
//
//The user runs the program by issuing:
//java GetConfDriver <alias or //server/database> <userid> <password>
//
//The arguments are:
//<alias> - DB2 subsystem alias for type 2 or //server/database for type 4
// connectivity
//<userid> - user ID to connect as
```

```
//<password> - password to connect with
//*************************************************************************
import java.lang.*;
import java.io.*;
import java.sql.*;

public class GetConfDriver
{

   public static void main (String[] args)
  {
    Connection con = null;
    CallableStatement cstmt = null;
    String driver = "com.ibm.db2.jcc.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: GetConfDriver <alias or //server/database> <userid>
<password>");
      System.err.println("where  <alias or //server/database> is DB2 subsystem alias or
//server/database for type 4 connectivity");
      System.err.println("       <userid> is user ID to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    try {

      byte[] xml_input;
      String str_xmlfilter = new String(
       "/plist/dict/key[.='DB2 Subsystem Specific Information']/following-sibling::dict[1]"
+
       "/key[.='V91A']/following-sibling::dict[1]" +
       "/key[.='DB2 Distributed Access Information']/following-sibling::dict[1]" +
       "/key[.='IP Address']/following-sibling::dict[1]" +
       "/key[.='Value']/following-sibling::string[1]");


      /* Convert XML_FILTER to byte array to pass as BLOB */
      byte[] xml_filter = str_xmlfilter.getBytes("UTF-8");

      // Load the DB2 Universal JDBC Driver
      Class.forName(driver);

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      cstmt = con.prepareCall("CALL SYSPROC.GET_CONFIG(?,?,?,?,?,?,?)");

      // Major / Minor Version / Requested Locale
      cstmt.setInt(1, 1);
      cstmt.setInt(2, 0);
```

```
cstmt.setString(3, "en_US");
// No Input document
cstmt.setObject(4, null, Types.BLOB);
cstmt.setObject(5, xml_filter, Types.BLOB);

// Output Parms
cstmt.registerOutParameter(1, Types.INTEGER);
cstmt.registerOutParameter(2, Types.INTEGER);
cstmt.registerOutParameter(6, Types.BLOB);
cstmt.registerOutParameter(7, Types.BLOB);

cstmt.execute();
con.commit();

SQLWarning ctstmt_warning = cstmt.getWarnings();
if (ctstmt_warning != null) {
  System.out.println("SQL Warning: " + ctstmt_warning.getMessage());
}
else {
  System.out.println("SQL Warning: None\r\n");
}

System.out.println("Major Version returned " + cstmt.getInt(1) );
System.out.println("Minor Version returned " + cstmt.getInt(2) );

/* get output BLOBs */
Blob b_out = cstmt.getBlob(6);

if(b_out != null)
{
  int out_length = (int)b_out.length();
  byte[] bxml_output = new byte[out_length];

  /* open an inputstream on BLOB data */
  InputStream instr_out = b_out.getBinaryStream();

  /* copy from inputstream into byte array */
  int out_len = instr_out.read(bxml_output, 0, out_length);

  /* write byte array into FileOutputStream */
  FileOutputStream fxml_out = new FileOutputStream("xml_output.xml");

  /* write byte array content into FileOutputStream */
  fxml_out.write(bxml_output, 0, out_length );

  //Close streams
  instr_out.close();
  fxml_out.close();
}

Blob b_msg = cstmt.getBlob(7);
if(b_msg != null)
{
  int msg_length = (int)b_msg.length();
  byte[] bxml_message = new byte[msg_length];

  /* open an inputstream on BLOB data */
  InputStream instr_msg = b_msg.getBinaryStream();

  /* copy from inputstream into byte array */
```

```
        int msg_len = instr_msg.read(bxml_message, 0, msg_length);

        /* write byte array content into FileOutputStream */
      FileOutputStream fxml_msg  = new FileOutputStream(new File("xml_message.xml"));
        fxml_msg.write(bxml_message, 0, msg_length);

        //Close streams
        instr_msg.close();
        fxml_msg.close();
      }
    }
    catch (SQLException sqle) {
       System.out.println("Error during CALL "
             + " SQLSTATE = " + sqle.getSQLState()
             + " SQLCODE = "  + sqle.getErrorCode()
             + " : " + sqle.getMessage());
    }
    catch (Exception e) {
       System.out.println("Internal Error " + e.toString());
    }
    finally
    {
       if(cstmt != null)
          try { cstmt.close(); } catch ( SQLException sqle) { sqle.printStackTrace(); }
       if(con != null)
          try { con.close(); } catch ( SQLException sqle) { sqle.printStackTrace(); }
    }
  }
}
```

Compile GetConfDriver.java and enter the following command to execute it:

```
java GetConfDriver DBALIAS USERID PASSWORD
```

Verify the created file ml_output.xml to obtain your data server's IP address.

B

Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

B.1 Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

```
ftp://www.redbooks.ibm.com/redbooks/SG247604
```

Alternatively, you can go to the IBM Redbooks Web site at:

```
ibm.com/redbooks
```

Select the **Additional materials** and open the directory that corresponds with this book form number, SG247604.

The zip files that accompany this book contain all of the sample files referenced in the book.

Important: Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 24 to decide what components are applicable to your environment.

Note: The additional Web material that accompanies this book includes the files described in the following sections.

B.1.1 Sample DB2 table DCLGEN files

This file contains DCLGEN output for the DEPT and EMP tables used in the case studies.

The following files can be found in \Samples\SG247604-DCLGEN.ZIP.

File name	*Description*
DEPT.TXT	Sample DCLGEN for DEPT table
EMP.TXT	Sample DCLGEN for EMP table

B.1.2 Sample COBOL programs

This file contains DDL, source code and program preparation JCL for all COBOL stored procedures, calling programs and called modules used in the case studies. All source code files are denoted by .SRC file extensions. The CREATE PROCEDURE statements can be found in the .DDL files. Jobs to prepare the programs can be found in the .JCL files.

The following files can be found in \Samples\SG247604-COBOL.ZIP.

CALDTL1C.JCL
CALDTL1C.SRC
CALDTL2C.JCL
CALDTL2C.SRC
CALDTL3C.JCL
CALDTL3C.SRC
CALDTL4C.JCL
CALDTL4C.SRC
CALRSETC.JCL
CALRSETC.SRC
EMPAUDTS.DDL
EMPAUDTS.JCL
EMPAUDTS.SRC
EMPAUDTU.DDL
EMPAUDTU.JCL
EMPAUDTU.SRC
EMPAUDTX.DDL
EMPAUDTX.JCL
EMPAUDTX.SRC
EMPDTL1C.DDL
EMPDTL1C.JCL
EMPDTL1C.SRC
EMPDTL2C.DDL
EMPDTL2C.JCL
EMPDTL2C.SRC
EMPDTL3C.DDL
EMPDTL3C.JCL
EMPDTL3C.SRC
EMPDTL4C.DDL
EMPDTL4C.JCL
EMPDTL4C.SRC
EMPEXC1C.DDL
EMPEXC1C.JCL
EMPEXC1C.SRC
EMPEXC2C.JCL
EMPEXC2C.SRC
EMPEXC3C.DDL
EMPEXC3C.JCL
EMPEXC3C.SRC
EMPEXC4C.JCL
EMPEXC4C.SRC
EMPODB1C.DDL
EMPODB1C.JCL
EMPODB1C.SRC
EMPRSETC.DDL
EMPRSETC.JCL
EMPRSETC.SRC

In the CALDTLnC JCL examples, the last DD card

```
//CARDIN   DD DSN=SG247604.CALDTL1C.CARDIN,DISP=SHR
```

simply indicates a sequential file to be allocated and used to pass a valid employee number to the stored procedure.

B.1.3 Sample C programs

This file contains DDL, source code and program preparation JCL for all C stored procedures and calling programs used in the case studies. All source code files are denoted by .SRC file extensions. The CREATE PROCEDURE statements can be found in the .DDL files. Jobs to prepare the programs can be found in the .JCL files.

The following files can be found in \Samples\SG247604-C.ZIP.

CALDTL1P.JCL
CALDTL1P.SRC
CALDTL2P.JCL
CALDTL2P.SRC
CALRSETP.JCL
CALRSETP.SRC
EMPDTL1P.DDL
EMPDTL1P.JCL
EMPDTL1P.SRC
EMPDTL2P.DDL
EMPDTL2P.JCL
EMPDTL2P.SRC
EMPRSETP.DDL
EMPRSETP.JCL
EMPRSETP.SRC

B.1.4 Sample Java programs

This file contains DDL, source code and program preparation JCL for all Java stored procedures, calling programs and called modules used in the case studies. All source code files are denoted by .SRC file extensions. The CREATE PROCEDURE statements can be found in the .DDL files. Jobs to prepare the programs can be found in the .JCL files. Note that many of the file names used for the Java examples are case sensitive.

The following files can be found in \Samples\SG247604-JAVA.ZIP.

CalDtlsj.java
EMPCLOBJ.ddl
EmpClobJ.java
EMPCLOBJ.jcl
EmpClobSpServlet.java
EMPDTL1J.ddl
EMPDTL1J.jcc.jcl
EMPDTL1J.jcl
EmpDtl1J.sqlj
EmpDtls2.java
EmpDtls3.java
EmpDtlsJ.class
EMPDTLSJ.ddl
EmpDtlsJ.java
EMPDTLSJ.jcl

EmpDtlsMJ.java
Employee.jar
EMPPHOTJ.ddl
EmpPhotJ.java
EMPPHOTJ.jcl
EmpPhotoSpServlet.java
EMPRMTEJ.ddl
EmpRmteJ.java
EMPRSETJ.ddl
EmpRsetJ.java
EMPRSETJ.jcl
EMPRST1J.ddl
EMPRST1J.jcl
EmpRst1J.sqlj
EMPRST2J.ddl
EMPRST2J.jcl
EmpRst2J.sqlj
EmpRst2J_UpdByPos.sqlj
EXTRACTJ.ddl
EXTRACTJ.jcl
ExtractJar.java
ExtractJarSp.java
Getters_staff.class
Getters_staff.jar
Getters_staff.java
SimpleInstallJar.class
SimpleInstallJar.java

B.1.5 Sample REXX stored procedures

This file contains DDL and source code for all REXX stored procedures used in the case studies. All source code files are denoted by .SRC file extensions. The CREATE PROCEDURE statements can be found in the .DDL files. There are no program preparation jobs necessary for REXX stored procedures.

The following files can be found in \Samples\SG247604-REXXSP.ZIP.

EMPDTLSR.DDL
EMPDTLSR.SRC
EMPRSETR.DDL
EMPRSETR.SRC

B.1.6 Sample External SQL language stored procedures

This file contains DDL and program preparation JCL for all external SQL language stored procedures used in the case studies. Since the source code for external SQL language stored procedures is embedded in the DDL, there are no separate files for the source code. The CREATE PROCEDURE statements, which include the source code, can be found in the .DDL files. Jobs to prepare the programs can be found in the .JCL files.

The following files can be found in \Samples\SG247604-External-SQL.ZIP.

EMPDTLSS.DDL
EMPDTLSS.JCL
EMPDTLV8.DDL
EMPDTLV8.JCL

EMPRSETS.DDL
EMPRSETS.JCL
SQLSPCUS.JCL

B.1.7 Sample Native SQL language stored procedures

This file contains DDL and JCL for all native SQL language stored procedures used in the case studies, as well as one calling program. Since the source code for native SQL language stored procedures is embedded in the DDL, there are no separate files for the source code. In this case the DDL and the JCL are in the same file.

The following files can be found in \Samples\SG247604-Native-SQL.ZIP.

CALCSAL.DDL
DIVIDEPR.DDL
GOTO.DDL
Median_RS.java
MEDIANV1.DDL
MEDIANV2.DDL
NODIFF.DDL
REBNDPCK.DDL
SCLABEL2.DDL
SCLABELN.DDL
SCOPECU2.DDL
SCOPEHND.DDL
SCPCHNDL.DDL
SCPCURS.DDL
SCVARS.DDL
TYPES.DDL

B.1.8 Sample multi-threaded stored procedure programs

This file contains DDL, source code and program preparation JCL for the multi-threaded C stored procedure used in the case studies. The stored procedures are introduced in Chapter 22, “Multi-threaded stored procedures in the C language” on page 441. The source code file for the stored procedure is denoted by an .SRC file extension, while the source code for the calling program is denoted by a .java file extension. The CREATE PROCEDURE statement can be found in the .DDL file. The job to prepare the stored procedure is denoted by a .JCL file extension.

The following files can be found in \Samples\SG247604-MULTITHD.ZIP.

RunstatPDriver.java
RUNSTATP.DDL
RUNSTATP.JCL
RUNSTATP.SRC

B.1.9 Sample code to invoke DB2-supplied stored procedures

This file contains source code for Java programs that were used for invoking DB2-supplied stored procedures. The stored procedures are introduced in Chapter 24, “DB2-supplied stored procedures” on page 493 and described in Appendix A, “Samples for using DB2-supplied stored procedures” on page 807.

The following files can be found in \Samples\SG247604-DB2SUPPLIED.ZIP.

AdminDB2Command.java

AdminDataSet.java
AdminJob.java
AdminUtilityExecution.java
AdminSystemInformation.java
AdminUNIXCommand.java
AdminWLMRefresh.java
AdminDSNSubcommand.java

B.1.10 Sample code for using the DB2-supplied task scheduler

This file contains source code for Java programs and DDL for tables and triggers that were used to show how to employ the DB2 provided task scheduler to schedule the execution of stored procedures. Specific use cases for employing the task scheduler are described in Appendix A.9, “Task Scheduler Sample Use cases” on page 858.

The following files can be found in \Samples\SG247604-TASKSCHED.ZIP.

AdminSchedule1.java
AdminSchedule3.java
AdminSchedule4.java
AdminScheduleR.java
INPTPARM.DDL
SCHEDTRI.DDL

B.1.11 Sample code for invoking the Common SQL API stored procedures

This file contains source code for Java programs and a sample XML document for use with the examples for invoking the Common SQL API stored procedures.

The following files can be found in \Samples\SG247604-COMMONSQLAPI.ZIP.

SPDriver.java
SPWrapper.java
GetConfDriver.java
CompleteMode.xml

B.1.12 Sample QMF queries

This file contains QMF queries and forms that can be used to report on stored procedure information maintained in the DB2 catalog. All queries are denoted by .TXT file extensions. All forms are denoted by .FRM file extensions.

The following files can be found in \Samples\SG247604-QMF.ZIP.

FRPARM70.FRM
FRPARMER.FRM
FRTNONLY.FRM
QRPARM70.TXT
QRPARMER.TXT
QRTNONLY.TXT

B.1.13 Sample DB2 triggers

This file contains DB2 triggers used in the case studies to show interaction between triggers and stored procedures. All files are DDL for the triggers and are denoted by .DDL file extensions.

The following files can be found in \Samples\SG247604-TRIGGER.ZIP:

EMPTRIG1.DDL
EMPTRIG2.DDL
EMPTRIG3.DDL

B.1.14 Sample REXX execs for configuration management

This file contains source code and execution JCL for REXX programs that were used for configuration management purposes. These are not stored procedures. The source code files are denoted by .SRC file extensions. The execution JCL files are denoted by .JCL file extensions. There are no .DDL files as these programs are not stored procedures.

The following files can be found in \Samples\SG247604-REXXEXEC.ZIP.

DDLMOD.SRC
DDLMODJB.JCL

B.1.15 Sample IMS ODBA setup jobs

This file contains execution JCL and sample WLM commands for setting up the IMS environment for our ODBA case study. All files have a .TXT file extension.

The following files can be found in \Samples\SG247604-JCLIMS.ZIP.

DB9AODBA.TXT
IMS01.TXT
IMS02.TXT
IMS03.TXT
IMS04.TXT
IMS05.TXT
IMS06.TXT
IMS07.TXT
IMS08.TXT
IMS09.TXT
IMS10.TXT
IMS11.TXT
IMS12.TXT
IMS13.TXT
WLMDEF.TXT

B.1.16 Sample objects for Data Studio examples

This file contains DDL and load files for the tables used in the Data Studio examples.

The following files can be found in \Samples\SG247604-Data-Studio.ZIP:

CreateTable.sql
CUSTOMER.data
PURCHASEORDER.data

B.1.17 Sample Unified Debugger Session Manager setup jobs

This file contains execution JCL for the tasks to set up the Session Manager for the Unified Debugger. All files have a .JCL file extension.

The following files can be found in \Samples\SG247604-Unified-Debugger.ZIP:

Sessmgr1.jcl
Sessmgr2.jcl
Sessmgr3.jcl

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 2 MB minimum
Operating System: Windows
Processor: Intel® 386 or higher
Memory: 16 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Abbreviations and acronyms

AIB	Application Interface Block
AIX	Advanced Interactive eXecutive from IBM
APAR	authorized program analysis report
ARM	automatic restart manager
ASCII	American National Standard Code for Information Interchange
BLOB	binary large objects
CCSID	coded character set identifier
CCA	client configuration assistant
CCMS	SAP's Computer Center Management System
CFCC	Coupling Facility control code
CTT	created temporary table
CEC	central electronics complex
CD	compact disk
CF	Coupling Facility
CFRM	Coupling Facility resource management
CICS	customer information control system
CLI	call level interface
CLP	command line processor
CPU	central processing unit
CSA	common storage area
DASD	direct access storage device
DB2 PM	DB2 performance monitor
DBAT	database access thread
DBD	database descriptor
DBID	database identifier
DBMS	data base management system
DBRM	database request module
DC	Development Center
DCL	data control language
DD	Distributed Debugger
DDCS	distributed database connection services
DDF	distributed data facility
DDL	data definition language
DLL	dynamic load library manipulation language
DML	data manipulation language
DNS	domain name server
DRDA	distributed relational database architecture
DSC	dynamic statement cache, local or global
DSN	data set name
DT	Debug Tool
DTT	declared temporary tables
EA	extended addressability
EBCDIC	extended binary coded decimal interchange code
ECB	event control block
ECS	enhanced catalog sharing
ECSA	extended common storage area
EDM	environment descriptor management
ERP	enterprise resource planning
ESA	Enterprise Systems Architecture
ESP™	Enterprise Solution Package
ETR	external throughput rate, an elapsed time measure, focuses on system capacity
FTD	functional track directory
FTP	File Transfer Program
GB	gigabyte (1,073,741,824 bytes)
GBP	group buffer pool
GRS	global resource serialization
GUI	graphical user interface
HPJ	high performance Java
IBM	International Business Machines Corporation
ICF	integrated catalog facility
ICMF	internal coupling migration facility
IFCID	instrumentation facility component identifier
IFI	instrumentation facility interface
IMS	information management system
IPLA	IBM Program Licence Agreement
IRLM	internal resource lock manager
ISPF	interactive system productivity facility
ISV	independent software vendor

I/O	input/output
IT	Information Technology
ITR	internal throughput rate, a processor time measure, focuses on processor capacity
ITSO	International Technical Support Organization
IVP	installation verification process
JCL	job control language
JDBC	Java Database Connectivity
JDSD	Job Data Set Display
JFS	journaled file systems
JNDI	Java Naming and Directory Interface
JVM	Java Virtual Machine
KB	kilobyte (1,024 bytes)
LPA	link pack area
LOB	large object
LPL	logical page list
LPAR	logical partition
LRECL	logical record length
LRSN	log record sequence number
LUW	logical unit of work
LVM	logical volume manager
MB	megabyte (1,048,576 bytes)
MFI	main frame interface
NPI	non-partitioning index
ODB	object descriptor in DBD
ODBA	Open Data Base Access
ODBC	Open Data Base Connectivity
OS/390	Operating System/390®
PAV	parallel access volume
PDS	partitioned data set
PIB	parallel index build
PSID	pageset identifier
PSP	preventive service planning
PTF	program temporary fix
PUNC	possibly uncommitted
QMF	Query Management Facility
QA	Quality Assurance
RACF	Resource Access Control Facility
RBA	relative byte address
RECFM	record format
RI	referential integrity
RID	record identifier
RRS	Resource Recovery Services
RRSAF	Resource Recovery Services attach facility
RS	read stability
RR	repeatable read
SC	service class
SDK	software developers kit
SDSF	System Display and Search Facility
SMIT	System Management Interface Tool
SQL	structured query language
SQLJ	Structured Query Language (SQL) that is embedded in the Java programming language
SU	service unit
SPAS	stored procedure address space
SPB	Stored Procedure Builder
SQL	structured query language
UCS	Unicode Conversion Services
UOW	unit of work
WSAD	WebSphere Studio Application Developer
WSADIE	WebSphere Studio Application Developer Integration Edition
WSED	WebSphere Studio Enterprise Developer
WLM	work load manager

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 900. Note that some of the documents referenced here may be available in softcopy only.

- *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083
- *DB2 9 for z/OS Technical Overview*, SG24-7330
- *DB2 9 for z/OS Performance Topics*, SG24-7473
- *LOBs with DB2 for z/OS: Stronger and Faster,* SG24-7270
- *DB2 for z/OS and OS/390: Ready for Java,* SG24-6435
- *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952
- *DB2 for z/OS Application Programming Topics*, SG24-6300-00
- *A Deep Blue View of DB2 Performance: IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS*, SG24-7224
- *Exploring WebSphere Studio Enterprise Developer 5.1.2,* SG24-6483
- *Building the Operational Data Store on DB2 UDB Using IBM Data Replicator, WebSphere MQ Family, and DB2 Warehouse Manager*, SG24-6513
- *Securing DB2 and Implementing MLS on z/OS*, SG24-6480-01
- *Moving Data Across the DB2 Family*, SG24-6905
- *IBM DB2 Performance Expert for z/OS Version 2*, SG24-6867-01
- *Systems Programmer’s Guide to Resource Recovery Services (RRS),* SG24-6980

Other publications

These publications are also relevant as further information sources:

- *DB2 Version 9.1 for z/OS Administration Guide,* SC18-9840-01
- *DB2 Version 9.1 for z/OS Application Programming and SQL Guide,* SC18-9841-01
- *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for JAVA* SC18-9842-01
- *DB2 Version 9.1 for z/OS Codes,* GC18-9843-01
- *DB2 Version 9.1 for z/OS Command Reference,* SC18-9844-01
- *DB2 Version 9.1 for z/OS Data Sharing: Planning and Administration,* SC18-9845-01
- *DB2 Version 9.1 for z/OS Diagnosis Guide and Reference,* LY37-3218-01
- *DB2 Version 9.1 for z/OS Diagnostic Quick Reference,* LY37-3219-00
- *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413

- *Support for Unicode: Using Conversion Services*, SC33-7050
- *DB2 Version 9.1 for z/OS Installation Guide,* GC18-9846-02
- *DB2 Version 9.1 for z/OS Introduction to DB2,* SC18-9847-01
- *DB2 Version 9.1 for z/OS Licensed Program Specifications,* GC18-9848-00
- *DB2 Version 9.1 for z/OS Messages,* GC18-9849-01
- *DB2 Version 9.1 for z/OS ODBC Guide and Reference,* SC18-9850-01
- *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide,* SC18-9851-02
- *DB2 Version 9.1 for z/OS RACF Access Control Module Guide,* SC18-9852-01
- *DB2 Version 9.1 for z/OS Reference for Remote DRDA Requesters and Servers,* SC18-9853-01
- *DB2 Version 9.1 for z/OS SQL Reference,* SC18-9854-02
- *DB2 Version 9.1 for z/OS Reference Summary,* SX26-3854-00
- *DB2 Version 9.1 for z/OS Utility Guide and Reference,* SC18-9855-01
- *DB2 Version 9.1 for z/OS What's New?,* GC18-9856-01
- *DB2 Version 9.1 for z/OS XML Extender Administration and Programming,* SC18-9857-01
- *DB2 Version 9.1 for z/OS XML Guide,* SC18-9858-02
- *Program Directory for IBM DB2 9 for z/OS,* GI10-8737-00
- *Program Directory for IBM DB2 V9.1 for z/OS DB2 Management Clients Package,* GI10-8738-01
- *DB2 QMF Reference Version 9 Release 1,* SC18-9685
- *DB2 Connect Version 9 User's Guide,* SC10-4229-00
- *IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS Version 4.1.0, Reporting User's Guide*, SC18-9983
- *z/Architecture Principles of Operations*, SA22-7832-04
- *z/OS V1R9.0 C/C++ Programming Guide*, SC09-4765-08
- *z/OS V1R9.0 XL C/C++ Run-Time Library Reference,* SA22-7821-09
- *z/OS V1R9.0 Language Environment Programming Guide,* SA22-7569-02
- *z/OS V1R8.0 Security Server RACF Command Language Reference,* SA22-7687-10
- *z/OS V1R9.0 Language Environment Customization*, SA22-7564-09
- *z/OS V1R9.0 Language Environment Programming Guide,* SA22-7561-08
- *z/OS V1R8.0 MVS Setting Up a Sysplex*, SA22-7625-13
- *z/OS V1R9.0 MVS Programming: Resource Recovery*, SA22-7616-07
- *z/OS V1R9.0 MVS Planning: Operations,* SA22-7601-08
- *z/OS V1R8.0 MVS Planning: Workload Management,* SA22-7602-13
- *z/OS V1R7.0 MVS JCL Reference,* SA22-7597-09
- *z/OS V1R9 MVS Initialization and Tuning Guide,* SA22-7591-05
- *z/OS V1R9.0 MVS Support for Unicode: Using Unicode Services*, SA22-7649-08
- *z/OS V1R9.0 Security Server RACF Command Language Reference*, SA22-7687-11
- *z/OS V1R9.0 UNIX System Services Planning*, GA22-7800-12
- *Enterprise COBOL for z/OS Programming Guide Version 3 Release 4*, SC27-1412-05

- *Enterprise COBOL for z/OS Language Reference Version 3 Release 4*, SC27-1408-04
- *IMS Version 9: Open Transaction Manager Access Guide and Reference*, SC18-7829
- *CICS DB2 Guide Version 3 Release 1*, SC34-6457-01
- *CICS 3.1 Application Programming Reference,* SC33-1688-02
- *New IBM Technology Featuring Persistent Reusable Java Virtual Machines*, SC34-6034
- *CICS Transaction Server for z/OS Version 3.2 CICS Application Programming Guide*, SC34-6818
- *z/OS V1R7.0 Resource Management Facility User's Guide,* SC33-7990-10
- *CICS Transaction Server for z/OS Version 2.2 CICS DB2 Guide*, SC34-6014-07
- *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201
- *DB2 Performance Expert for z/OS Version 1 Monitoring Performance from ISPF*, SC27-1652-02
- *z/OS MVS Data Areas V1R7*, GA22-7583
- *Debug Tool for z/OS Debug Tool Utilities and Advanced Functions for z/OS User's Guide Version 8.1,* SC19-1196-01
- *Debug Tool for z/OS Customization Guide*, SC19-1200
- *Debug Tool for z/OS Reference and Messages*, GC19-1198
- *The Rational Developer for System z Host Configuration Guide*, SC23-7658
- *Integrated Cryptographic Service Facility System Programmer's Guide,* SC23-3974
- *CICS Transaction Server for z/OS V3.1 CICS External Interfaces Guide*, SC34-6449-03
- *DB2 Performance Monitor for z/OS Version 7.2, Report Reference*, SC27-1647
- *z/OS MVS Data Areas*, SY28-1164

Online resources

This Web site is also relevant as a further information source:

- Debug Tool Web site at:

  `http://www.ibm.com/software/awdtools/debugtool/`
- The developerWorks Web site at:

  `http://www.ibm.com/developerworks`
- Information on installing the Java SDK at:

  `http://www.ibm.com/servers/eserver/zseries/software/java/`
- SQL Debugger for DB2 UDB V7.2 and DB2 UDB V8.1:

  `http://www.ibm.com/developerworks/db2/library/techarticle/alazzawe/0108alazzawe.html`
- DB2 DEMO workstation GUI tool at:

  `http://www.ibm.com/developerworks/db2/library/demos/db2demo/index.html`
- IBM Data Studio Web site:

  `http://www.ibm.com/software/data/studio`
- DB2 for z/OS library Web page:

  `http://www.ibm.com/software/data/db2/zos/library.html`
- DB2 Tools for z/OS and IMS Tools

http://www.ibm.com/software/data/db2imstools/

- Technote “Converting an external SQL procedure to a native SQL procedure”

  http://www-01.ibm.com/support/docview.wss?rs=64&context=SSEPEK&uid=swg21297948&loc=en_US&cs=utf-8&lang=en

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Redbooks

DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond

(1.5” spine)
1.5”<-> 1.998”
789 <->1051 pages

DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond

Develop and test COBOL, C, REXX, Java and SQL procedures

Set up, control, and tune the operating environment

Learn about IBM Data Studio and other tools

This IBM Redbooks publication helps you design, install, manage, and tune stored procedures with DB2 9 for z/OS. Stored procedures can provide major benefits in the areas of application performance, code re-use, security, and integrity. DB2 has offered an ever improving support for developing and operating stored procedures.

In these days, three years is a generation in the software business; if you have DB2 9 for z/OS, this book replaces the previous DB2 for z/OS Stored Procedures: Through the CALL and Beyond, SG24-7083-00, and it reflects the changes that have happened to DB2 stored procedures and related tools from V8 to V9.

We show how to develop stored procedures in several languages, including Java; we explore the functions available for the z/OS platform deployment; and provide recommendations on setting up and tuning the appropriate stored procedure environment.

We talk about the external and native SQL procedures, the debugging options, the special registers, the deployment, and diagnostics.

A chapter is devoted to the increasing number of DB2-supplied stored procedures. They can be used for almost all of a DBA's tasks.

We also devote a part to tools which can be used for accelerating the development process and go in some detail about the stored procedure support provided by the latest IBM product: Data Studio. For recent information on Data Studio, refer to *Data Studio and DB2 for z/OS Stored Procedures*, REDP-4717.

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

SG24-7604-00 ISBN 0738485934