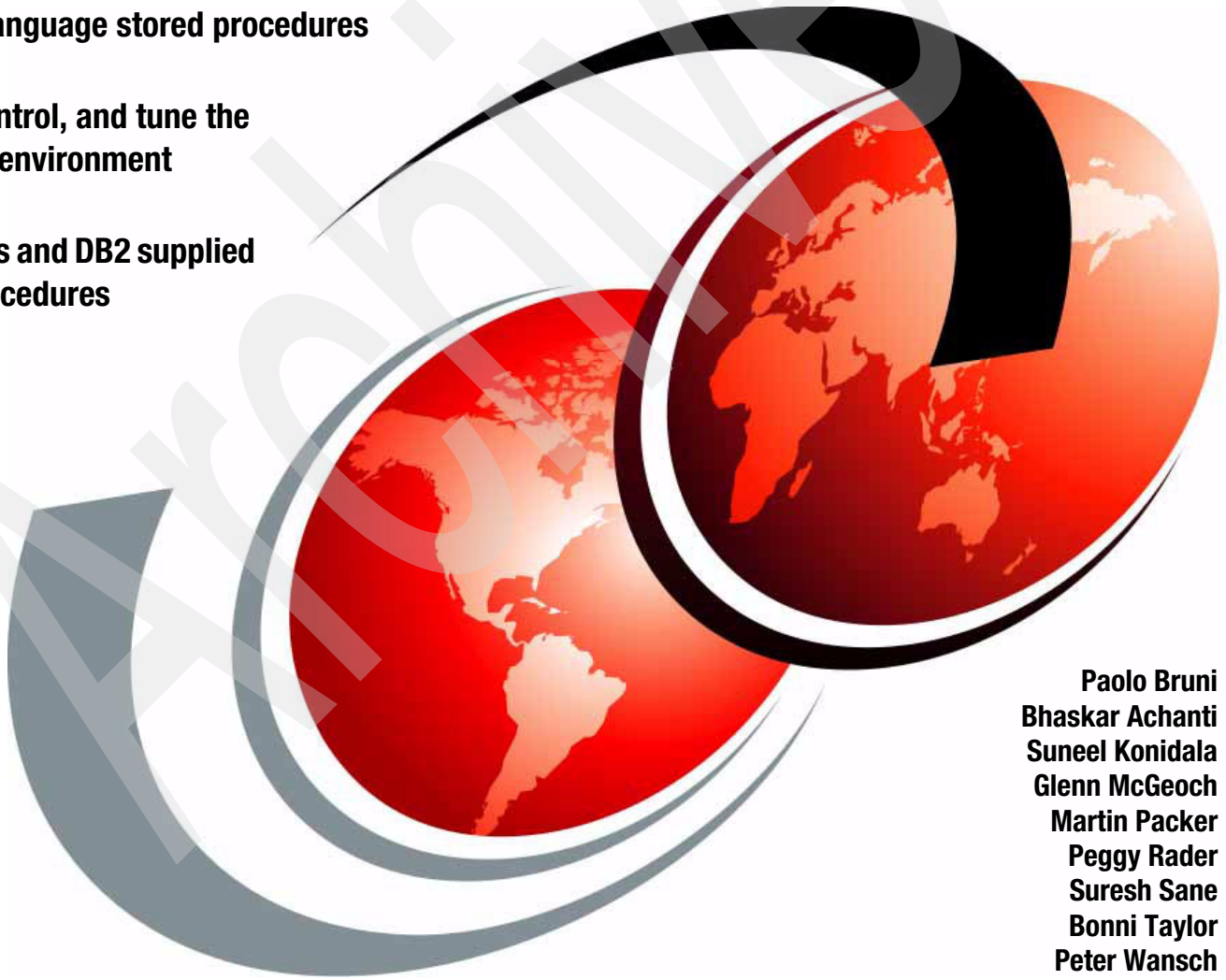
DB2 Information Management Software

IBM

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

- Develop and test COBOL, C, REXX, Java, and SQL language stored procedures
- Set up, control, and tune the operating environment
- Learn tools and DB2 supplied stored procedures



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

Redbooks

ibm.com/redbooks

International Technical Support Organization

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

March 2004

SG24-7083-00

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

First Edition (March 2004)

This edition applies to IBM DB2 UDB for OS/390 and z/OS Version 7 (program number 5675-DB2) and DB2 UDB for z/OS Version 8 (program number 5625-DB2).

Note: This book is based on a pre-GA version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.

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

Trademarks

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

@server®
@server®
Redbooks (logo) ™
developerWorks®
eServer™
iSeries™
z/Architecture™
z/OS®
zSeries®
AD/Cycle®
AIX®
C/370™
CICS®
COBOL/370™
Distributed Relational Database Architecture™
DB2 Connect™
DB2 Universal Database™
DB2®
DFS™
DRDA®
Footprint®
Infoprint®
IBM®
IMS™
Language Environment®
MQSeries®
MVS™
MVS/ESA™
NetView®
OS/2®
OS/390®
OS/400®
QMF™
Redbooks™
RACF®
RMF™
S/390®
System/390®
VisualAge®
VTAM®
WebSphere®

The following terms are trademarks of other companies:

Java, Java Naming and Directory Interface, JDBC, JDK, JVM, J2EE, Solaris, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows NT, 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 changes made in this edition of the book since it was first published. This edition may also include other minor corrections and editorial changes.

Summary of Changes
for SG24-7083-00
for DB2 for z/OS Stored Procedures: Through the CALL and Beyond
as created or updated on February 21, 2006.

March 2004, First Edition

The revisions of this First Edition reflect the additions and the changes described below.

June 2004, First Update

Changed information

Change bars identify the corrected area.

- Page 16, Fig. 2.2, corrected text in the figure from SYSTEM to SYSIBM
- Page 18, Fig. 2.3, corrected bullets for steps 1 and 2
- Page 298, wrong pointer to Fig. 20.3 changed to Fig. 19.3
- Page 386, added clarification on VSAM file definition
- Added CARDIN DD clarification to Appendix D.1.3, “Sample C programs” on page 653
- Changed all references to DB2® manuals from DB2 V7 to V8

September 2004, Second Update

Changed information

Change bars identify the corrected area.

- Edited “dataset” to “data set” througout the redbook.
- Table 3-11 on page 28, corrected errors in the text.
- Updated text under “Usage notes” on page 238, and “Usage notes” on page 239 to reflect considerations on DSNTPSMP code level.
- Example 29-9 on page 514, updated text to reflect DB2 V8.1 libraries.

Added information

Change bars identify the added area.

- Added clarification un using VSAM file in 24.1.1, “Accessing CICS systems through EXCI” on page 387 and 386.
- Added the information the APAR PQ77702 is still open in 24.2.2, “Accessing IMS databases through stored procedure DSNAIMS” on page 399.
- Added 29.2.6, “Development Center and JDBC Driver selection” on page 518.
- Added 29.2.7, “Java SDKs used by DB2 Development Center on OS/390 and z/OS” on page 520.
- Added comments to the GETSQLSP and PUTSQLSP REXX execs in the samples to reflect considerations on DSNTPSMP code level.

November 2004, Third Update

Changed information

Change bars identify the corrected area.

- Edited some text in Figure 16-7 on page 233, Figure 16-8 on page 235, and Figure 16-10 on page 237.

Added information

Change bars identify the added area.

- Added header 16.3.2, “Compile every time” on page 234.

April 2005, Fourth Update

Changed information

Change bars identify the corrected area.

- Clarified recommendation in 13.2.3, “Handling comment lines” on page 160.
- Several minor corrections on descriptions of the REXX execs in 16.4, “Notes on REXX execs” on page 238.
- Corrected Example 17-9 on page 255.
- Corrected Example 18-4 on page 280.
- Corrected Example 29-15 on page 523 and Example 29-16 on page 523 by adding a missing right parenthesis.
- Corrected text from DTWAIMS_FUNCTION to DSNAIMS_FUNCTION in “DSNAIMS_2PC” on page 645.
- Replaced SG247083-REXXEXEC.ZIP file in Additional Material with a more generalized version of DDLMOD.SRC.

Added information

Change bars identify the added area.

- Added clarification that LANGUAGE=C should *not* be specified when creating SQL procedures in 13.2.10, “Re-deploying SQL procedures” on page 168.
- Added a Tip box in 17.5.2, “Preparing stored procedures with only JDBC Methods” on page 256.
- Added comment on environmental variables at the end of 17.2.6, “Setting up the JAVAENV data set for Java stored procedure execution” on page 249.
- Added a Note box in 18.2.4, “Binding the packages for Universal JDBC Driver” on page 281.
- Added information in 24.2.2, “Accessing IMS databases through stored procedure DSNAIMS” on page 399 because of the availability of PTFs for APAR PQ77702.
- Added reference to the new redbook on WSED at the end of 28.3.3, “Creating COBOL stored procedures for debugging” on page 481.
- Added authorization requirement for USS /tmp directory at the end of 29.2.2, “z/OS setup” on page 506.
- Added reference to PK01445 in 29.2.6, “Development Center and JDBC Driver selection” on page 518.
- Added reference to informational APAR II1171 in Appendix A, “Frequently asked questions” on page 581.
- Added Appendix C, “DSNAIMS stored procedure” on page 643.

February 2006, Fifth Update

Changed information

Change bars identify the corrected area.

- Corrected return to COMMIT at “CREATE PROCEDURE” on page 313.
- Changed information at “Step 5: Create required WLM application environments” on page 409 on assigning stored procedure DSNACCMO to a WLM application environment. NUMTCB = 100 is not necessary. Also updated Table 25-6 on page 409.

Added information

Change bars identify the added area.

- Added Example 10-10 on page 100 and clarified the usage of indicator variables for parameters used with parameter style DB2SQL.
- Added clarification on COMMIT at 19.1.2, “The execution life cycle of a stored procedure” on page 298.

Preface

This IBM® Redbook helps you design, install, manage, and tune stored procedures with DB2 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. DB2's enhancements are related to tooling, language support, system environment, and have opened new possibilities for secure, highly portable applications in line with the e-business strategy of today's organizations.

In this project we show how to develop stored procedures in several languages; we explore the new functions available for the z/OS platform deployment; and we provide recommendations on setting up and tuning the appropriate stored procedure environment. The functions we have investigated include setting up the WLM environments, nesting stored procedures, invoking COBOL, C, REXX, Java™ and SQL language programs, accounting, debugging options, special registers, and diagnostics. We have also set up, developed, and debugged Java stored procedures with the new Java Universal Driver in a DB2 for z/OS Version 8 environment. A chapter is devoted to DB2-supplied stored procedures. They can be used for almost all of the DBA's tasks.

We start with the basic information, which is useful for the reader who is just beginning with stored procedures, but we also deal with more detailed and recent functionalities, which will be of interest for the more advanced users.

If you have been one of the many readers of the best-downloading redbook *Getting Started with DB2 Stored Procedures: Give Them a Call through the Network,* SG24-4693-01, and have given them a DB2 *CALL*, now is the time to look at more information on the CALL and to go *Beyond*.

Contents and scope of this redbook

Many roles within the IT shop are involved in defining and using stored procedures.

The possible roles in handling stored procedures, depending on the structure of your organization, are:

- Application programmer (in the various languages) for the stored procedure modules
- Tools specialists
- DBA or operations for the stored procedure definition to DB2
- MVS™ system programmer for WLM Environment, RRSAF, LE set up, system tuning parameters
- Network support for connectivity

Except for the unusual case of an individual experimenting with stored procedures in a sandbox system, cooperation of several IT areas is highly recommended for a successful implementation. In this chapter we briefly describe the contents and the structure of this redbook. Two reasons for this: anticipate the contents and their purpose, and try to help each role in locating the sections relevant by responsibility.

This redbook is structured in eight parts as follows:

- Introduction

- Operating environment
- Developing stored procedure
- Java stored procedures
- Performance
- Extending the functions
- Cool tools for an easier life
- Appendixes

In this section we briefly introduce the topics contained in each part.

Introduction

In Part 1, “Introduction” on page 1 we introduce the stored procedures, their importance, the contents of the book, and the case study that we used throughout the project.

Importance of stored procedures

Since their introduction in DB2 for MVS/ESA™ Version 4, the roles of stored procedures in an enterprise are continually growing. In this chapter we discuss what stored procedures are, why they are important, and how pervasive they are becoming.

Stored procedures overview

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

Stored procedures are DB2 objects as well as programs. They must be defined to the DB2 catalog and allowed to execute under MVS.

This chapter summarizes definitions and components.

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.

Note: All of the files required to reconstruct the sample applications can be downloaded from the IBM Redbooks™ Web site: ibm.com/redbooks. Refer to Appendix D, “Additional material” on page 651 for more details.

Operating environments

In Part 2, “Operating environment” on page 31, we describe the operating environment requirements for stored procedures written for the DB2 for z/OS environment.

Workload Manager

Support of DB2-established stored procedure address spaces (SPAS) is diminishing, and will be eliminated for new stored procedures in DB2 for z/OS V8. For this reason, we use the Workload Manager (WLM) SPAS in the discussions and examples throughout this redbook. One of the major benefits of WLM SPAS is its ability to manage multiple address spaces as opposed to the single SPAS of DB2-established address spaces. Details of WLM are included in Chapter 4, “Setting up and managing Workload Manager” on page 33. Chapter 8, “Operational issues” on page 67 details operational issues such as refreshing the SPAS, prevention and termination of hanging or looping procedures, and handling application

failures. For more information on WLM SPAS management, refer to 20.2, “Managing server address spaces” on page 324, and Chapter 20, “Server address space management” on page 319.

Language Environment

Before the introduction of Language Environment® (LE), it was important to know what programming languages were collectively used in an application. STEPLIB DD statements for a run step had to include the run-time libraries for each of the languages present. Therefore, if you were missing a runlib, then the portion of the application that referenced the features of the missing language subroutines would, to say the least, develop any one of many potential problems. The analysis time to arrive at the solution of proper runlib concatenation can be expensive.

With LE, you can use one run-time environment for your applications, regardless of the application programming languages or system resource needs. It is good performance practice to have the LE library CEE.SCEERUN in LNKLST or in LLA in order to benefit from the lookaside MVS buffering capabilities. When a stored procedure is defined to DB2, you can specify specific LE run-time options to be utilized by your stored procedure. See Chapter 5, “Language Environment setup” on page 41 for more information.

Resource Recovery Services Attachment Facility

When your application program determines that a unit of work is complete, you issue a request to commit or rollback the unit of work. The application requires all resources to be processed in the same manner. Either the entire unit of work is committed or none of the work is committed. The resources referenced in the stored procedure are also included in this unit of work. Obviously, *coordination* of this activity is necessary.

For z/OS Resource Recovery Services (RRS) enabled subsystems (including the WLM-established address spaces), RRS coordinates two-phase commit processing with all of the resource managers for the recoverable resources referenced by an application.

Stored procedures that execute in a WLM address space must use the Resource Recovery Services Attachment Facility (RRSAF) to ensure proper synchronization of all protected resources. DSNRLI is the language interface module for RRSAF applications and must be included in the load modules of stored procedures.

With RRSAF, you can coordinate DB2 updates with updates made by all other resource managers that also use z/OS RRS. Refer to Chapter 6, “RRSAF” on page 47 for a more detailed description of how this attachment is used by DB2 for stored procedures.

Security and authorization

Stored procedures add a few additional authorization tasks to the mix of DB2 security and authorization requirements. Privileges are required to:

- Access WLM resources
- Execute CREATE/ALTER PROCEDURE SQL statement
- Bind the stored procedure
- Execute the stored procedure

These privileges and other additional security considerations are discussed in Chapter 7, “Security and authorization” on page 55.

Developing stored procedures

Part 3, “Developing stored procedure” on page 73 deals with the steps necessary for defining, coding, testing, and maintaining stored procedures. We provide examples in COBOL, C, REXX, and the SQL language.

Defining stored procedures

Stored procedures are DB2 objects that must be defined to the DB2 catalog with the CREATE PROCEDURE statement. This statement names the procedure; defines the parameters to be shared with the client; specifies the run-time environment information, the load module name, the language of the source code, and more. You can find the details in Chapter 9, “Defining stored procedures” on page 75.

Coding stored procedures

Stored procedures can be written with any of the DB2 supported languages. In all these languages, the stored procedure program assumes that it is connected to DB2 already (because it is, at CALL time). So, no CAF or RRS attach calls, nor REXX attachment calls are coded in the program. In JDBC you have to create a connection (usually to jdbc:default:connection) to be able to create a statement; in SQLJ it is common to create a connection context object. The actual program preparation steps are somewhat language and interface dependent. With some exceptions (REXX, Java, and SQL Procedures language), stored procedures are prepared in the same manner as any other DB2 embedded SQL program.

In this redbook, we have chapters detailing the coding considerations by language.

See Chapter 10, “COBOL programming” on page 93, Chapter 11, “C programming” on page 127, Chapter 12, “REXX programming” on page 151, and Chapter 13, “SQL Procedures language” on page 157.

Because Java offers many unique considerations, preparation, setup, etc., we have a separate part, Part 4, “Java stored procedures” on page 243 for your perusal.

There are ongoing concerns and discussions about replacing COBOL subprograms with stored procedures. In 10.3, “COBOL subprogram interfaces” on page 111, we offer insights into subprogramming and stored procedures, nested stored procedures, and recycling existing code.

Testing and debugging stored procedures

Debugging stored procedures presents new challenges for the developer. There are a number of new tools and techniques available for debugging a stored procedure. In Chapter 14, “Debugging” on page 173, we address SQL errors, debugging options for z/OS, and use the IBM Debug Tool and a DB2 for z/OS Version 8 function called *Get Diagnostics*, which enables applications to retrieve intricate diagnostic information about SQL statements that have executed. Depending on the tools available on your development environment’s platform, you may also want to read Chapter 28, “Tools for debugging DB2 stored procedures” on page 463.

Remote stored procedure calls

When nesting requires a call to a remote stored procedure, there are concerns for when and where to bind the procedure. Chapter 15, “Remote stored procedure calls” on page 213 provides you with helpful information.

Code level management

All IT organizations maintain multiple DB2 environments and multiple application releases within each environment. Versioning and promoting applications certainly presents a large number of challenges. Read Chapter 16, “Code level management” on page 223 for an understanding of configuration management and tips.

Java stored procedures

Part 4, “Java stored procedures” on page 243 is dedicated to Java stored procedures. First we discuss the stored procedures setup, and provide examples of coding and debugging, then we introduce the new Java Universal Driver, and show how to use it and migrate to it.

Building Java stored procedures

Java and DB2 for z/OS work together in order to allow you to run mission critical applications written in Java. We show how to set up a Java environment, prepare, define, code, and debug stored procedures with both SQLJ and JDBC.

Using the new Universal Driver

Several enhancements have been introduced by IBM to Java functions, such as 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, and the deprecation of compiled Java. We show the JCC setup for DB2 stored procedures, the SQLJ preparation process using the new JCC, and advise on Migrating stored procedures to use the new JCC driver.

Performance

Performance, performance, performance. Issues, issues, issues. Organizing performance issues into “themes” is logical. Each one has a beginning, a middle, and an end component. The components differ based upon the needs of the “theme”. This is how Part 5, “Performance” on page 293 is built.

General performance considerations

This section offers an introduction to performance characteristics, tools, and capacity planning of DB2 stored procedures. We begin with another perspective of the basic concepts of stored procedures from the performance aspect. Concepts can be very different depending on your focus. Chapter 19, “General performance considerations” on page 295 offers this and recommendations for performance oriented definitions of stored procedures.

Server address space management

WLM manages the life cycle of server address spaces. Are your stored procedures not being scheduled soon enough? Are your applications missing the performance mark? How do you know? In Chapter 20, “Server address space management” on page 319, we have a detailed description of how WLM SPAS work, and how they can be monitored and controlled.

I/O performance management

If system level performance tools suggest the server address spaces are performing a significant amount of non-DB2 I/O, causing inadequate performance in your workload, then a discussion of how to manage stored procedure I/O performance and contention issues is prescribed for relief. Chapter 21, “I/O performance management” on page 331 highlights the nature of I/O and ENQs that can delay stored procedures.

Extending the functions

In Part 6, “Extending the functions” on page 335, 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.

Enhancements to stored procedure with DB2 V8

As DB2 for z/OS Version 8 approaches, you might want to get a head start on some of the enhancements for stored procedures. Chapter 22, “Enhancements to stored procedures with DB2 V8” on page 337 provides a general overview of major enhancements. Some of the topics include IBM DB2 Universal Driver for SQLJ and JDBC features, DDF communication database enhancements, and specific stored procedure and UDF enhancements.

Developing multi-threading stored procedures in C

In this chapter we explore the possibility of multi-threading within stored procedures. If the function executed within the stored procedure is complex and can be split and assigned to multiple concurrent threads, then, as for any other case of concurrent actions, multi-threading can largely reduce execution time and improve performance.

Accessing CICS and IMS

There are a number of interfaces available to accommodate a need to reference IMS™ databases or CICS® transactions from a stored procedure. Chapter 24, “Accessing CICS and IMS” on page 385 explains how to utilize the stored procedure DSNACICS to execute an existing CICS program, 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. We also discuss how to code SQL calls in CICS and IMS applications and give an overview of how to prepare those applications.

DB2-supplied stored procedures

DB2 provides several stored procedures that you can call in your application programs to perform database and system administration functions. Chapter 25, “DB2-supplied stored procedures” on page 403 provides an overview of all the stored procedures supplied by DB2, how to install and activate them, and how to use them with a comprehensive sample program written in Java, which shows how to process results and handle errors correctly.

Using LOBs

Chapter 26, “Using LOBs” on page 439 offers you an opportunity to delve into creating DB2 tables and populating them with LOB data, building a Java stored procedure to access an employee photo resume table, building a Java servlet to stream the output on to a Web browser, and returning BLOBs and CLOBs.

Using triggers and UDFs

While triggers and user defined functions are not new to DB2, Chapter 27, “Using triggers and UDFs” on page 451 provides an excellent explanation of what they are and how to effectively use stored procedures with them. Parameter passing to stored procedures called from a trigger and error handling details are incorporated. A comparison of stored procedures and UDFs is clearly presented with discussions of stored procedures invoking UDFs and UDFs invoking stored procedures.

“Cool” tools for an easier life

We designed Part 7, “Cool tools for an easier life” on page 461 to introduce you to a set of tools that will enhance your development and debugging efforts in a very “*cool*” way.

Options for debugging DB2 stored procedure

What we labeled as *traditional* debugging options is presented in Chapter 14, “Debugging” on page 173. But in Chapter 28, “Tools for debugging DB2 stored procedures” on page 463, we present debugging options for stored procedures utilizing some tools that can be launched from non z/OS platforms to debug z/OS stored procedures. This chapter gives the implementation requirements and details to use:

- The Development Center Integrated SQL Debugger for SQL stored procedures
- Debugging High Level Language stored procedures with the IBM Distributed Debugger
- Debugging options for DB2 Java stored procedures on z/OS

Also, see Chapter 30, “Using WSAD to debug Java stored procedures converted to Java applications” on page 553 for debugging Java and SQLJ stored procedures with WebSphere® Studio Application Developer (WSAD).

The Development Center

The DB2 Development Center included in DB2 V8.1 UDB Application Development Client component is the LUW (Linux®, UNIX®, and Microsoft® Windows®) follow-on product to the DB2 Stored Procedure Builder in the DB2 V7.2 UDB Application Development Client component. You can use the Development Center to create SQL and Java stored procedures for DB2 for z/OS. To learn about these features and some advanced topics such as generating multiple SQL statements, multiple result sets, and including code fragments, read Chapter 29, “The DB2 Development Center” on page 499.

Using WSAD to debug Java stored procedures

Since there is no specific tool currently available for debugging Java stored procedures, in Chapter 30, “Using WSAD to debug Java stored procedures converted to Java applications” on page 553 we show a possible solution by the use of WSAD to debug Java stored procedures converted to Java programs.

Appendixes

Part 8, “Appendixes” on page 579 contains the following:

Appendix A

You might actually find Appendix A, “Frequently asked questions” on page 581 to be an appropriate starting point for reading this redbook. The frequently asked questions, with their answers, might prompt you with ideas of where to start your research in this book. In this appendix we have divided the questions into four categories: application development, performance, administration, and miscellaneous facts. Where appropriate, the answers will refer you to details contained in this redbook or other IBM reference materials.

Appendix B

Some of the DB2-provided stored procedures source code is documented in Appendix B, “Samples for using DB2-supplied stored procedures” on page 595. Here, we provide you with the source for six of the many useful stored procedures that can be utilized for management of data sets, submitting utility JCL, issuing USS or DB2 commands, and automating RUNSTATS.

Appendix C

Sample stored procedures used in this redbook can be located on the Web. Appendix D, “Additional material” on page 651 provides you with the necessary information for system requirements to download the samples, and how to access them after the download.

Related publications

This redbook presents numerous topics and information compiled by a highly motivated and talented team of DB2 professionals covering a wide range of disciplines. It was a difficult challenge to keep this book from becoming a six-volume encyclopedia. Throughout this document are references to related IBM publications. A list of all of the publications can be found in Appendix , “Related publications” on page 657.

The team that wrote this redbook

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

Paolo Bruni is a certified Consultant IT Architect working as a Data Management Project Leader at the International Technical Support Organization, San Jose Center since 1998. In this capacity he has authored several redbooks on 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 mainly related to database systems.

Bhaskar Achanti heads the DB2 system and Data Administration Group at National Bank of Kuwait (NBK), one of the largest Banks in the Middle East, Kuwait. He holds a bachelors degree in Mechanical Engineering from the Indian Institute of Technology, Kharagpur, India. Bhaskar has over 11 years of experience in areas of DB2 Application and Database Design, performance tuning, disaster recovery, installation, and customization of DB2 subsystems. He is well versed with distributed and Web based technologies, and their interaction with DB2 for z/OS. Prior to joining NBK, Bhaskar was headed the Corporate Database Group at TATA STEEL, India.

Suneel Konidala is a DB2 Systems Programmer with Capital One Services Inc., Richmond, VA, USA. He is IBM Certified Solutions Expert in DB2 for z/OS database administration, and DB2 for UNIX, Microsoft Windows and OS/2® database administration. He has previously worked with DB2 as an application programmer and database administrator. Suneel has over 8 years of experience in the areas of DB2 application programming and design, database design, installation and tuning of data sharing environments, disaster recovery, and migration of applications from non-RDBMS like IMS and IDMS to DB2. He holds a master’s degree in Engineering from Indian Institute of Technology, Mumbai, India.

Glenn McGeoch is a Senior DB2 Consultant for IBM’s DB2 for z/OS Services organization in the United States, working out of San Francisco, CA. He has 26 years of experience in the software industry, with 18 years of experience working with DB2 for z/OS and OS/390®. 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 7 years with IBM assisting DB2 customers. His areas of expertise include application design and performance, DB2 data sharing, and DB2 migration planning. He has given presentations on data sharing at regional DB2 User Groups and he has presented to customers on DB2 stored procedures, migration planning, and application programming topics.

Martin Packer is an IBM mainframe and DB2 performance consultant in the United Kingdom. He has 18 years of experience in large systems performance and 10 years of experience in DB2 performance. He holds a bachelor’s degree in Mathematics and Physics, and a master’s degree in Information Technology, both from University College, London. His areas of expertise include MVS, batch, and DB2 tuning. He also spends a good deal of time developing analysis tools to support IBM’s worldwide performance consulting practice. He

has written extensively on Data In Memory, batch, MVS performance, z/OS 64-bit, and DB2 performance.

Peggy Rader is an IBM Senior Software Engineer in the DB2 Application Development Tooling Test and Build at IBM's Silicon Valley Lab in San Jose, California. She has more than 25 years of experience in large systems including IMS and DB2. She holds a degree in mathematics from the University of Washington. Her areas of expertise include stored procedures and many workstation client tools for application development for DB2 on all platforms. She presents and demonstrates the integration of IBM's Windows-based development tools for z/OS and Microsoft Windows environments, including the DB2 Development Center, and WebSphere Studio Application Developer for users internationally.

Suresh Sane is a Database Architect with DST Systems in Kansas City, Missouri, USA. He co-authored the redbook *Squeezing the Most Out of Dynamic SQL* in 2002. He is actively involved with the North American conferences of IDUG with numerous presentations and an article in the IDUG Solutions Journal. He has worked with DB2 since Version 1. He holds a bachelor's degree in Electrical Engineering from Indian Institute of Technology, Mumbai, India, and an MBA from Indian Institute of Management, Kolkutta, India.

Bonni Taylor is a Senior Consulting Instructor with TechEd Associates in Middletown, New Jersey. Currently, she specializes in a variety of DB2 for OS/390 and z/OS topics, including teaching stored procedures to Enterprise COBOL programmers around the world. She has been a COBOL and Assembler programmer for more than 30 years, and has utilized DB2 since Version 1.2.

Peter Wansch is a Software Engineer in DB2 Content Manager at IBM's Silicon Valley Lab in San Jose, California. He has 7 years of experience in DB2 client and Web application development as a consultant, developer, architect, and project manager. He holds a master's degree in computer engineering from the University of Technology, Vienna, Austria. He is a Sun™ certified Java developer, certified DB2 administrator on Linux, UNIX, Windows and OS/390, and a certified AIX® administrator. He has written extensively on Java network programming and media streaming. During the last 3 years Peter worked at the Toronto Lab where he was responsible for the development of the new DB2-provided stored procedures that are used by Control Center and SAP Management Console CCMS.

A photo of the team is in Figure 1.

Figure 1 Left to right: Bonni, Glenn, Paolo, Suneel, Bhaskar, Peter, Martin, Peggy, and Suresh (photo courtesy of Bart Steegmans)

Special thanks to Peggy Abelite Zagelow for her technical guidance throughout this project.

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

Rich Conway
Maritza M. Dubec
Emma Jacobs
Bob Haimowitz
Bart Steegmans
IBM International Technical Support Organization, USA

Peggy Abelite Zagelow
Maria Sueli Almeida
Meg Bernal
Bill Bireley
Ben Budiman
Marion Farber
Christopher Farrar
Laura Grady
Wendy Koontz
Vikram Manchala
Phyllis Marlino
Roger Miller
Todd Munk
Frank Neyen
Mary Petras
Suzanne Phillips
Tom Ross

Jim Ruddy
Akira Shibamiya
Sheila Sholars
Hugh Smith
Rich Vivenza
Xavier Yuen
IBM Silicon Valley Lab, USA

Nick Carbone
Mark Picard
Rick Tallman
IBM Poughkeepsie Lab, USA

Ernie Mancill
IBM DB2 Tools for z/OS and OS/390, USA

Ken Blackman
Steve Zemblowski
IBM Americas Advanced Technical Support, USA

Janice Haywood
IBM, Mount Laurel, USA

Naomi McClung
IBM Learning Services, Birmingham, USA

Rob Weaver
IBM, Gaithersburg, USA

Ted Blank
IBM Washington Systems Center, USA

Peter Cram
IBM Hartford, USA

Terry Berman
Mike Todd
DST Systems, Inc., USA

Randy Horman
Juliana Hsu
Alex Stevens
Yves Tolod
Debbie Yu
IBM Toronto Lab, Canada

Andreas Mueller
Johannes Schuetzner
IBM eServer™ Software Development, Boeblingen, Germany

Become a published author

Join us for a two- to seven-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll 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 Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

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

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. QXXE Building 80-E2
  650 Harry Road
  San Jose, California 95120-6099

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 21 defines the environment and the case study which was 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 are continually growing. 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 are stored procedures
- Benefits of stored procedures
- Use of stored procedures
- Multi-tiered applications and stored procedures

1.1 What are stored procedures

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.

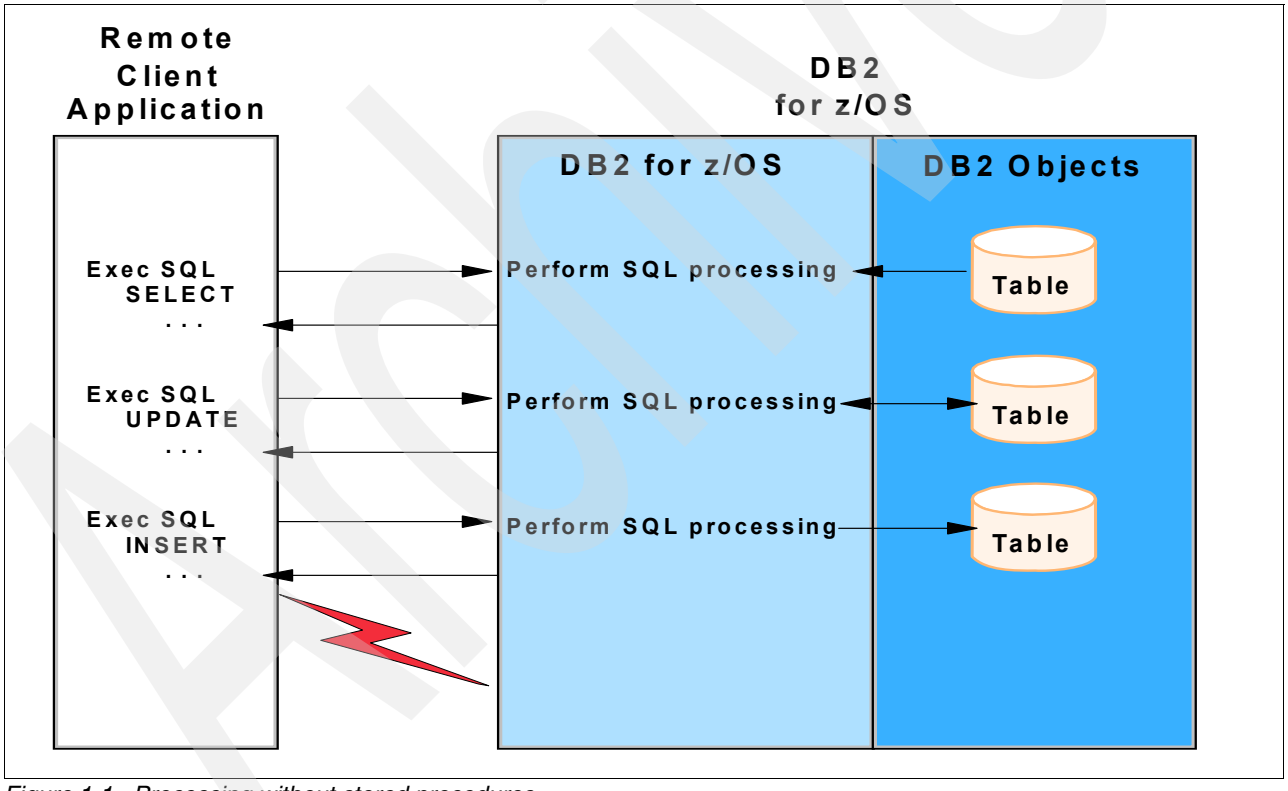


Figure 1-1 Processing without stored procedures

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 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.

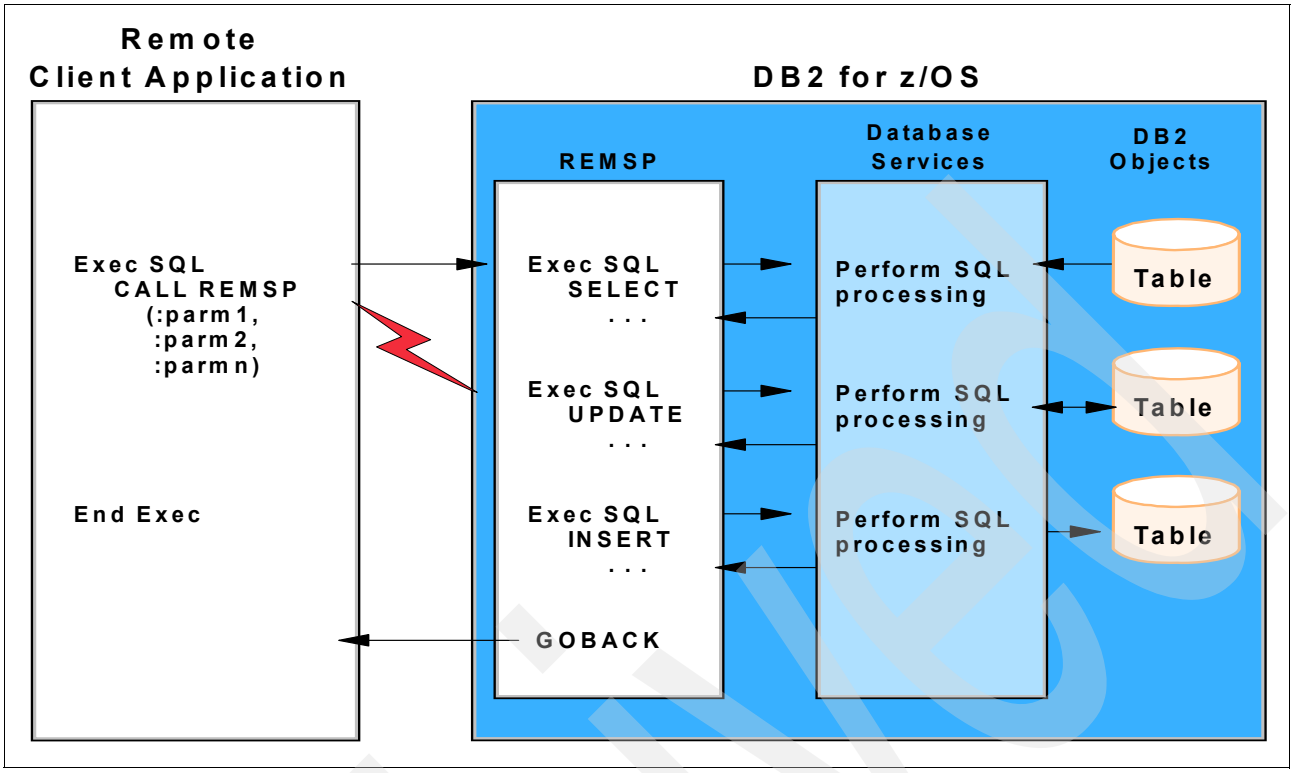


Figure 1-2 Processing with stored procedures

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 distributed 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 will process 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:
  - 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 accesses to multiple data and platforms.

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 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:

- Execute two or more SQL statements

  Remote SQL statements can create many network send and receive operations, which results in increased processor costs.
- Contain manipulative and or logical processing that must be consistent across multiple processes, and the goal is to provide coding that is reusable and therefore requiring minimal maintenance efforts.
- Access 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 design architects a solution 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 as 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 and Java. Support for invoking a stored procedure and processing its results 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 categories of stored procedures, depending on the actual coding of the procedure being available as an external load module in an MVS library, or the procedure being entirely written in SQL:

- External procedures
- SQL procedures

For the SQL (or internal) procedure, SQL is the only language used, and the program logic is part of the definition. For external stored procedure, definition and program logic are two separate components.

2.1.1 External procedures

An external 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 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 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 procedure" on page 73 for programming details, and Part 2, "Operating environment" on page 31 for setting up the stored procedures 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 75, 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:

- Single copy can be shared by multiple tasks in the 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, linkedit 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 SQL 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 stored procedure, an SQL procedure consists of a stored procedure definition and the code for the stored procedure program. Most of the CREATE PROCEDURE options are the same. The definitions differ in the way that they specify the location of the code. An external stored procedure definition specifies the name of a load module, while an SQL procedure definition *contains* the source code for the stored procedure. We can specify name of load modules even for SQL Procedures through the *external name* option for both V7 and V8.

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

Example 2-3 CREATE PROCEDURE sample

```
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 SQL procedures, see Chapter 13, “SQL Procedures language” on page 157.

2.2 Example of a stored procedure flow

In Figure 2-1, we show the statements 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 amount of network transmissions that is 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 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

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 redbook, we examine the components, services, and considerations that must be integrated to insure success.

Figure 2-1 represents the flow of our sample stored procedure:

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, 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 see 16.2, “Versioning of stored procedures” on page 226.

4. Stored procedures are executed in address spaces that run fenced away from the DB2 code. For DB2 Version 7, a single address space is available if we use the DB2-established address space, and multiple WLM (work load manager) address spaces are available if we use a WLM environment for the stored procedure. DB2 Version 7 is the last version that will support the creation of DB2-managed stored procedures. In DB2 for z/OS Version 8, all newly created stored procedures must use the WLM-established stored procedures address space. Those that existed prior to Version 8 will continue to run, but all new stored procedures must run in a WLM-managed stored procedures address space.

   The required release of z/OS 1.3 makes WLM goal mode mandatory. For DB2-established or WLM-established address spaces, 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.

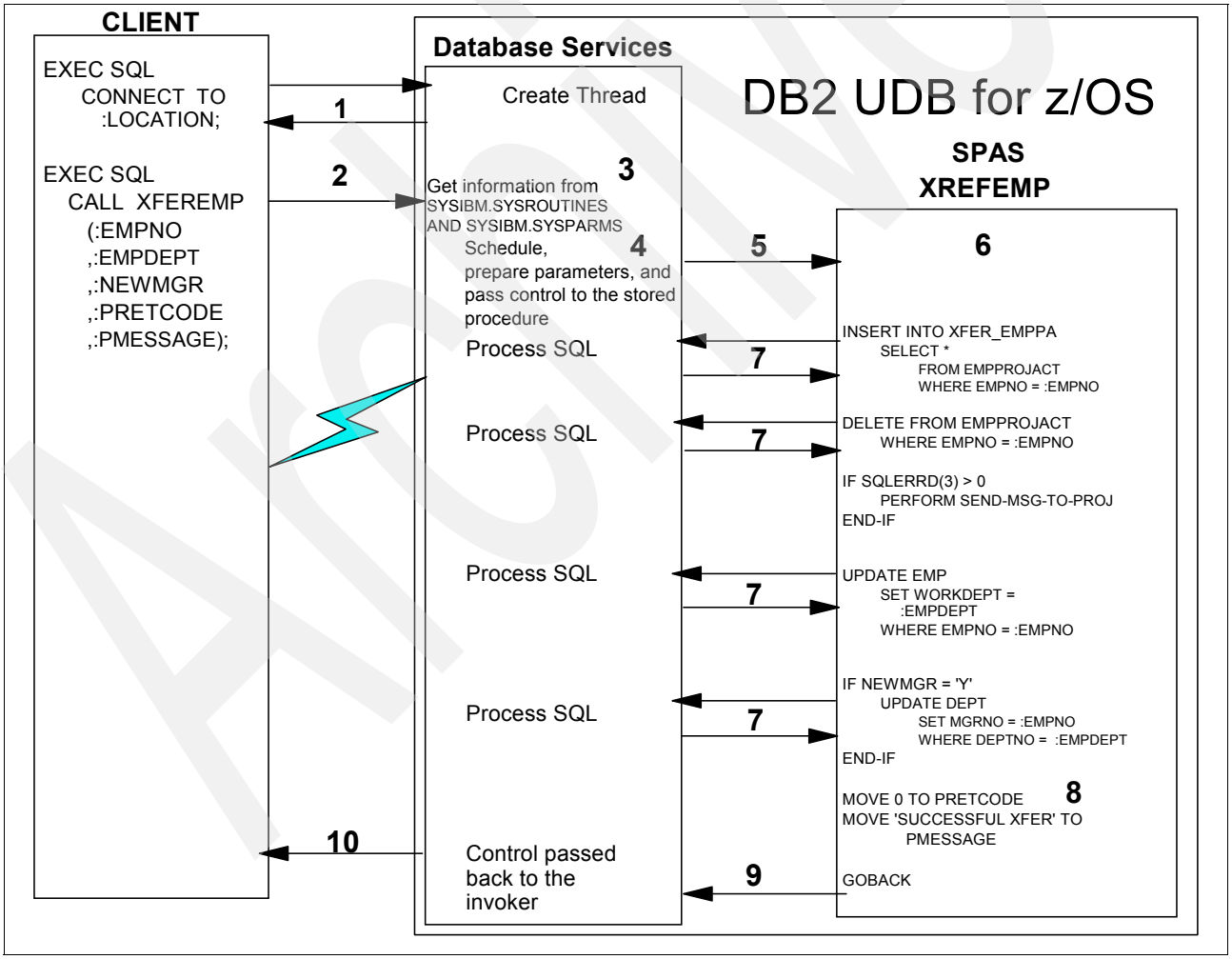


Figure 2-1 Stored procedure that transfers employees - Statement’s flow

5. When the stored procedures 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 procedures address space uses the LE/370 product libraries to load and execute the stored procedure. Through SYSIBM.SYSROUTINES, you can pass run-time 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.
   - 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 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. 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 value in the CREATE PROCEDURE statement. See 9.1, “CREATE or ALTER PROCEDURE parameters” on page 76 for details of the CREATE statement. 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, 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 run-time environment, language, number of parameters, parameter style, whether or not result sets can be returned, if DB2 should execute a commit when returning to the caller, etc.

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

Example 2-4 Query to retrieve stored procedure run-time 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 = 'EMPDTLSC' AND SCHEMA = 'DEVL7083'
```

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

Example 2-5 QMF output

```
PROCEDURE NAME: DEVL7083.PAOLOR6.EMPDTLSC
************************************************************
COLLECTION ID: DEVL7083  LANGUAGE: COBOL  EXT. NAME: EMPDTLSC

WLM ENVIRONMENT: DB2GDEC1  RESIDENT: N  MAIN/SUB M

RUN OPTIONS: TRAP(OFF),RPTOPTS(OFF),TEST(OFF)

SQL DATA ACCESS: M   RESULT SETS RETURNED: 0

COMMIT ON RETURN: N

NUMBER OF PARMS: 10 PARAMETER STYLE: G   DBINFO: N
```

For details on producing the QMF report, refer to *QMF Reference,* SC27-0715.

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, if the parameter row is associated with a locator or table.

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

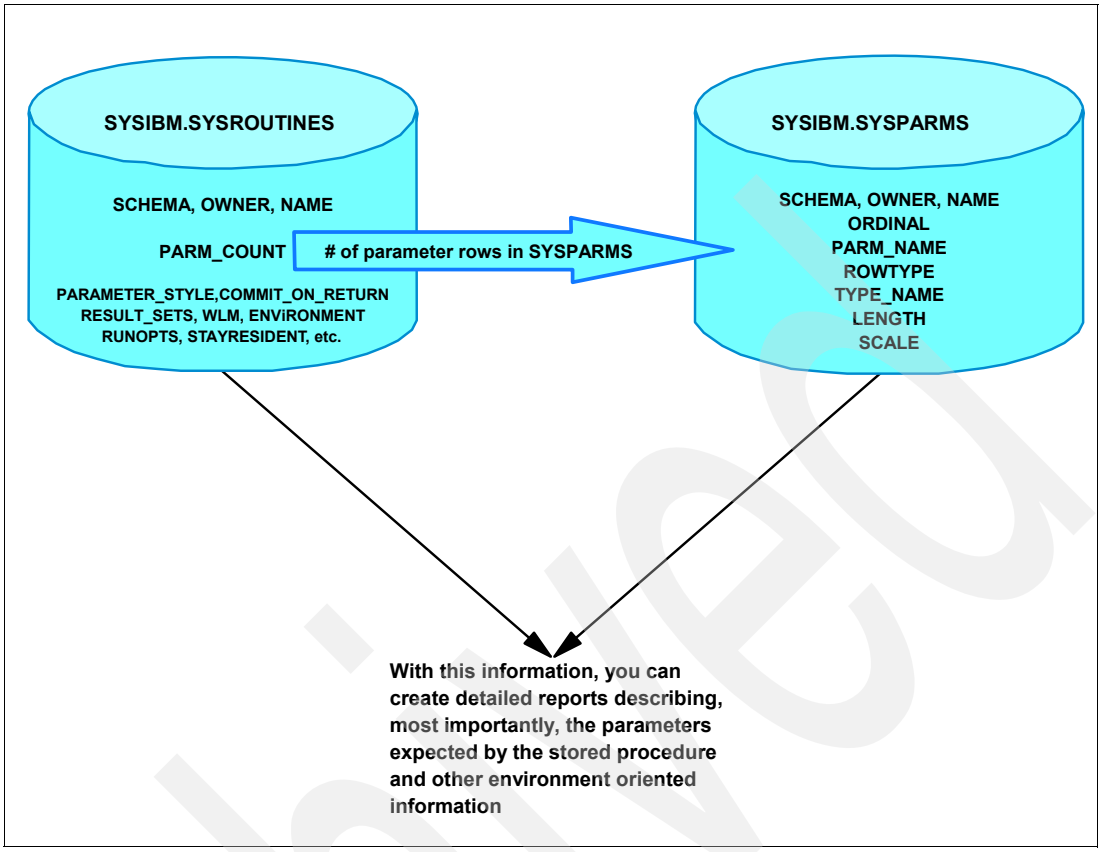


Figure 2-2 Relationship between SYSIBM.SYSROUTINES and SYSIBM.SYSPARMS

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 = 'EMPDTLSC'
     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 = 'EMPDTLSC'
     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.PAOLOR6.EMPDTLSC  10 PARMS/G                          N   M   N
                       00001      PEMPNO          CHAR(6)       P   N   N
                       00002      PFIRSTNME       VARCHAR(12)   O   N   N
                       00003      PMIDINIT        CHAR(1)       O   N   N
                       00004      PLASTNAME       VARCHAR(15)   O   N   N
                       00005      PWORKDEPT       CHAR(3)       O   N   N
                       00006      PHIREDATE       DATE(4)       O   N   N
                       00007      PSALARY         DECIMAL(9,2)  O   N   N
                       00008      PSQLCODE        INTEGER(4)    O   N   N
                       00009      PSQLSTATE       CHAR(5)       O   N   N
                       00010      PSQLERRMC       VARCHAR(250)  O   N   N
```

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 2-3 illustrates the relationship between application code, DB2 address spaces, and WLM application environments, etc.

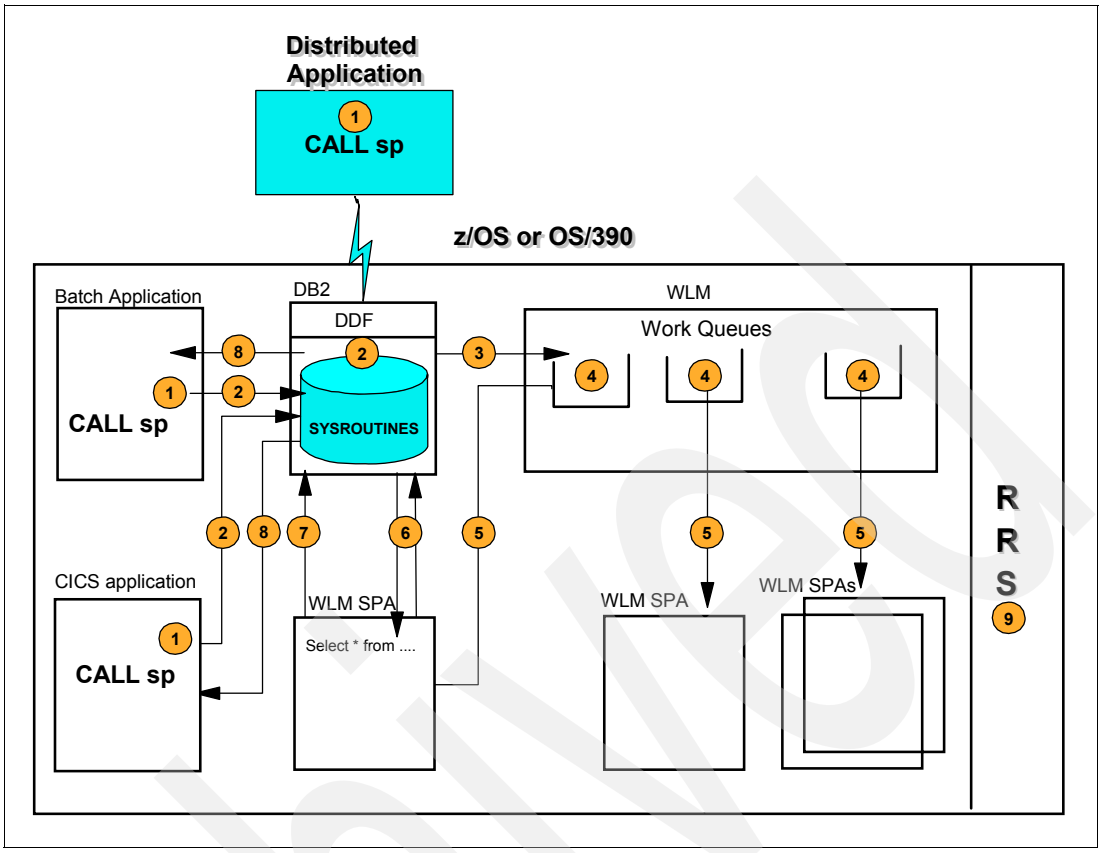


Figure 2-3 The system management of stored procedures

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

1. The application, most likely from a client workstation, issues 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 parameters definitions, retrieves the collection ID (COLLID) and WLM application environment (WLM_ENVIRONMENT) names associated with the stored procedure.
3. DB2 uses the following sequence to determine the collection ID:

   a. CURRENT PACKAGE PATH special register in the program (DB2 V8)
   b. CURRENT PACKAGESET in the program
   c. COLLID collection name in CREATE PROCEDURE
   d. CURRENT PACKAGESET in the calling application (DB2 V8)
   e. COLLID in the calling application
   f. PKLIST in the calling application
4. DB2 sends a request to WLM to schedule the stored procedure in an application environment.

5. 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 within a queue are processed in an FIFO manner.
6. Once your request has its turn, WLM checks for the availability of WLM SPAS.
   - If no WLM SPAS exists, it 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 service class, it waits for availability of 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 will be 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 will be started that executes the stored procedure.
   - If WLM AE is in a stopped or quiesced state, it sends a return code back to the calling program.
7. Once the stored procedure is scheduled, it executes the SQL inside it (if SQL are present).
8. When the stored procedure reaches return statement, it passes control back to DB2 with or without results depending on the logic.
9. DB2 passes control back to the calling program with or without results depending on the logic.
10. 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, again all the above steps will happen.

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 of 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 D, “Additional material” on page 651 for more details.

This chapter contains the following:

- Overview
- Sample application components
- Naming conventions

3.1 Overview

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, and SQL Procedures language.

3.2 The environment

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

- DB2 for OS/390 and z/OS Version 7 for all examples except those in the Version 8 chapter
- z/OS Version 1.4
- CICS Transaction Server Version 2.2
- IMS Version 7
- DB2 Performance Expert for z/OS Version 1
- DB2 Performance Monitor Version 7.2
- QMF Version 7.1
- DB2 UDB Enterprise Server Edition V8.1.4
- DB2 Connect™ Personal Edition V8.1.4
- IBM Debug Tool for z/OS and OS/390 Version 3.1
- Java Developer Kit Version 1.3.1
- Java Developer Kit Version 1.4.1 (for DB2-supplied stored procedures samples)

The hardware configuration for our case studies is as follows:

- z990 (2084-A08) processor in an LPAR with 2 GB central storage and two logical processors

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 write multi-threaded stored procedures in C; how to write stored procedures that interact with CICS and IMS; how to write applications to call DB2-supplied stored procedures; and how to call stored procedures from triggers and user defined functions.

Some of the Java and SQL language stored procedures were imported into the DB2 Development Center in our Version 8 system to show you how to create and debug stored procedures using the GUI capabilities of the Development Center. See 29.4.5, “Creating an SQL stored procedure on z/OS” on page 534, and 29.4.6, “Creating a Java stored procedure on z/OS” on page 535 for details on creating Java and SQL stored procedures within the Development Center. See 28.2, “Debugging SQL SPs on z/OS, Windows, UNIX, and Linux” on page 466 for details on debugging sample SQL stored procedure EMPDTLSS.

Our sample stored procedures were run on DB2 for OS/390 and z/OS Version 7, 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

Table 3-2 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

Object Type	Name	Description
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 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
	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
Calling program	CALDTLSJ	Java application program that calls the EMPDTLSJ stored procedure. CalDtlsJ.java
	EmpClobSpServlet	Java servlet that calls a stored procedure EMPCLOBJ and writes out the output to a Web Page. EmpClobSpServlet.java
	EmphotoSpServlet	Java servlet calls a 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
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 SQL language programming examples.

Table 3-6 Objects for 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 examples of developing multi-threaded stored procedure in the C language.

Table 3-7 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	DB2CallRUNSTATP	Java application that calls stored procedure RUNSTATP. DB2CallRUNSTATP.java

Table 3-8 lists the objects that were used for the examples of applications that call DB2-supplied stored procedures.

Table 3-8 Objects for DB2-supplied stored procedure examples

Object Type	Name	Description
Calling program	DB2Command	Java application program that calls the DB2-supplied stored procedure DSNACCMD, which executes DB2 commands
	DB2DatasetUtilities	Java application program that calls the DB2-supplied stored procedures DSNACCDS, DSNACCDR, DSNACCDE, DSNACCDL and DSNACCDD, which manage datasets
	DB2JCLUtilities	Java application program that calls the DB2-supplied stored procedure DSNACCJS, which submits a JCL job
	DB2Runstats	Java application program that calls the DB2-supplied stored procedures DSNACCOR and DSNACCMO, which can be used to automate the RUNSTATS process
	DB2SystemInformation	Java application program that calls the DB2-supplied stored procedures DSNACCSS, DSNACCSI, DSNWZP and DSNUTILS, which display system information
	DB2USSCommand	Java application program that calls the DB2-supplied stored procedure DSNACCUC, which issues UNIX System Services commands
	DB2USSUserInformation	Java application program that shows how to use the DB2 provided system user-defined function DSNAICUG to query the USS user database
	DB2WLMRefresh	Java application program that calls the DB2-supplied stored procedure WLM_REFRESH, which can be used to refresh a WLM application environment

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 Windows 2000. In many cases we did not provide a calling application for all the stored procedures in each language; instead we used the DB2 Development Center 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 3-9 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 14 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 3-9 QMF objects

Object Name	Description
QRPARMER	QMF query that retrieves information about a stored procedure and associated parameters as well as external name and run-time options.

Object Name	Description
QRPARM70	QMF query that retrieves information about a stored procedure and associated parameters
QRTNONLY	QMF query that retrieves stored procedure run-time 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 3-10 lists the triggers that are used in our examples to show their interaction with stored procedures and user defined functions.

Table 3-10 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 3-11 lists three REXX execs that we used for configuration management purposes. See 16.4, “Notes on REXX execs” on page 238 for more details.

Table 3-11 REXX execs for configuration management

REXX Name	Description
GETSQLSP	REXX exec that extracts source of an SQL stored procedure from DB2 catalog table (SYSROUTINES_SRC) table and stores in a data set.
PUTSQLSP	REXX exec that stores an SQL stored procedure into DB2 catalog table (SYSROUTINES_SRC) table from a data set.
DDLMOD	REXX exec that modifies the DDL based on the values specified in the configuration file and generates SYSIN cards which can be used for WLM refresh, drop stored procedure and SET CURRENT SQLID.

Table 3-12 lists the jobs that we used to set up the IMS environment for our ODBA example. See 24.2.1, “Accessing IMS databases through ODBA interface” on page 392 for a detailed description of the ODBA setup process.

Table 3-12 Jobs for IMS ODBA setup

Job Name	Description
IMS01	Add the DBD, PSB and IMS Tran macros to IMS stage 1 gen
IMS02	Define the DBD source and run the DBDGEN
IMS03	Define the PSB source and run the PSBGEN
IMS04	Define the ACBGEN source and run the ACBGEN
IMS05	Define the VSAM data set and run IDCAMS
IMS06	Define the source and run the dynamic allocation job for the database
IMS07	Define and run the DBRC registration for the DEPT database

Job Name	Description
IMS08	Load the database with DFSDDLT0
IMS09	Define and assemble the DFSPRP macro
IMS10A	Perform the IMS Stage 2 gen
IMS10B	Perform the IMS online change
IMS11	Define the execution WLM environment
IMS12	Define the associated WLM proc for the WLM environment

3.4 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 33 refers to WLM application environments that are used for the case studies. The WLM application environment naming standard is also described in 3.4.3, “WLM application environment names” on page 29. Many of the naming conventions are based on the last four digits of the redbook publication number.

3.4.1 Table names

We created copies of two of the sample tables provided with DB2 for OS/390 and z/OS Version 7:

- **EMP** is the sample Employee table
- **DEPT** is the sample Department table

3.4.2 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

3.4.3 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 33 to understand the criteria we used for defining WLM environments. The naming convention for the WLM application environments is summarized in Figure 3-1.

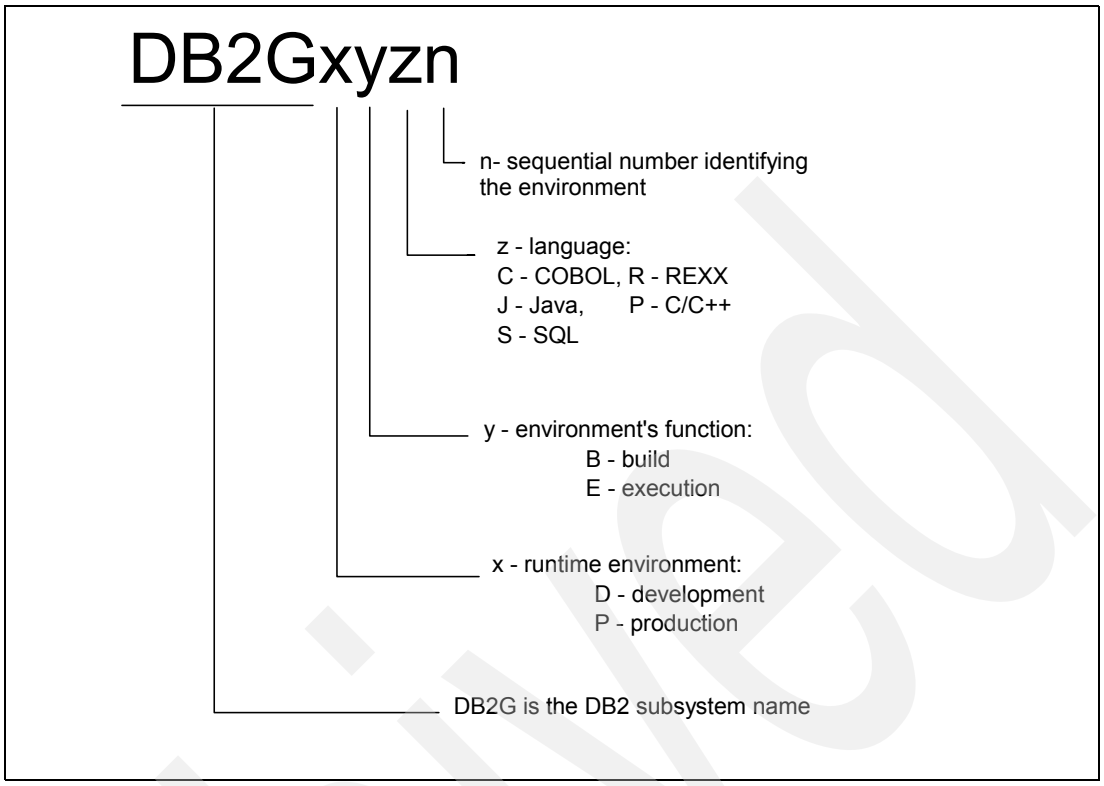


Figure 3-1 WLM environment names

When we first designed this naming convention, we thought there was value in the y environment function setting. Later on, we decided this setting was not needed but did not change all the definitions that were already created.

Part 2

Operating environment

In this part we describe the set up of the operating environment which 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:

- How to diagnose SQLCODE -471/-00E79002 (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
- 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 OS/390 and 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 OS/390 and 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 will determine 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 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 319.

The setting of the WLM set up is dependent upon the functions that are being executed:

- Miscellaneous DB2-supplied stored procedures

  One WLM Application Environment is recommended for miscellaneous DB2-supplied stored procedures used by Development Center, as described in Table 4-1.

- DB2 Development Center REXX related stored procedures

  One WLM Application Environment is recommended to process the two DB2-supplied REXX stored procedures used by Development Center during the build process. The DSNTPSMP REXX stored procedure is used to create SQL stored procedures, and the DSNTBIND REXX stored procedure is used to do the bind for SQLJ support. These two REXX stored procedures can share the same WLM environment as shown in Table 4-1.

- SQL, COBOL, C/C++ and PL/1 related stored procedures

  One or more WLM Application Environments are recommended for executing SQL, COBOL, C/C++, and PL/1 user stored procedures. The same WLM Application Environment can be used when the same STEPLIB requirements apply to all languages. When different STEPLIB datasets are required, then additional WLM Application Environments need to be created to support the processing of the different STEPLIB datasets.

- Debugging COBOL, C/C++ and PL/1 related stored procedures

  When Debug Tool is used to debug COBOL, C/C++, or PL/1 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 for debugging than when they are not executing in debug mode.

- Java related stored procedures

  A separate WLM Application Environment is recommended for executing the users’ Java stored procedures. Java stored procedures *must* execute in a WLM environment; SQL stored procedures can execute in the DB2 stored procedure address space (DB2 SPAS).

How many WLM Application Environments?

The max NUMTCB setting depends on available resources. The only required setting is 1 for REXX stored procedures; Java stored procedures should be no more than 8, since a Java Virtual Machine (JVM™) is loaded for each NUMTCB value. Table 4-1 makes a starting recommendation that should be modified as needed.

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

Stored procedure name	NUMTCB	Customization job hlq.SDSNSAMP	Comments
DB2DEBUG.SYS* SQLJ.INSTALL_JAR SQLJ.REPLACE_JAR SQLJ.REMOVE_JAR SYSIBM.SQL* SYSPROC.WLM_REFRESH SYSPROC.DSNTJSPP	40-60	DSNTIJSD DSNTIJSG DSNTIJSG DSNTIJSG DSNTIJSG DSNTIJSG DSNTIJSG	DB2-supplied stored procedures. All Steplib datasets must be authorized for WLM_REFRESH
SYSPROC.DSNTPSMP SYSPROC.DSNTBIND	1	DSNTIJSG DSNTIJSG	Both DSNTPSMP and DSNTBIND are REXX stored procedures and must run in an address space with NUMTCB=1
User SQL stored procedures	10-40	none / created manually or by using DB2 Development Center or Stored Procedure Builder	Must have one unauthorized data set. COBOL, C/C++, PL/1 stored procedures can share if the JCL is comparable
User Java stored procedures	5-8	none / created manually or by using DB2 Development Center or Stored Procedure Builder	Must have one unauthorized data set. One JVM is loaded for each NUMTCB specified. The presence of the //JAVAENV in the JCL causes the JVM to be loaded

Figure 4-1 illustrates using one WLM Application Environment for running DSNTPSMP to create SQL stored procedures using DB2 Development Center, and a second WLM Application Environment for executing the user SQL stored procedure, which may have comparable JCL for COBOL, C/C++, and PL/1 stored procedures.

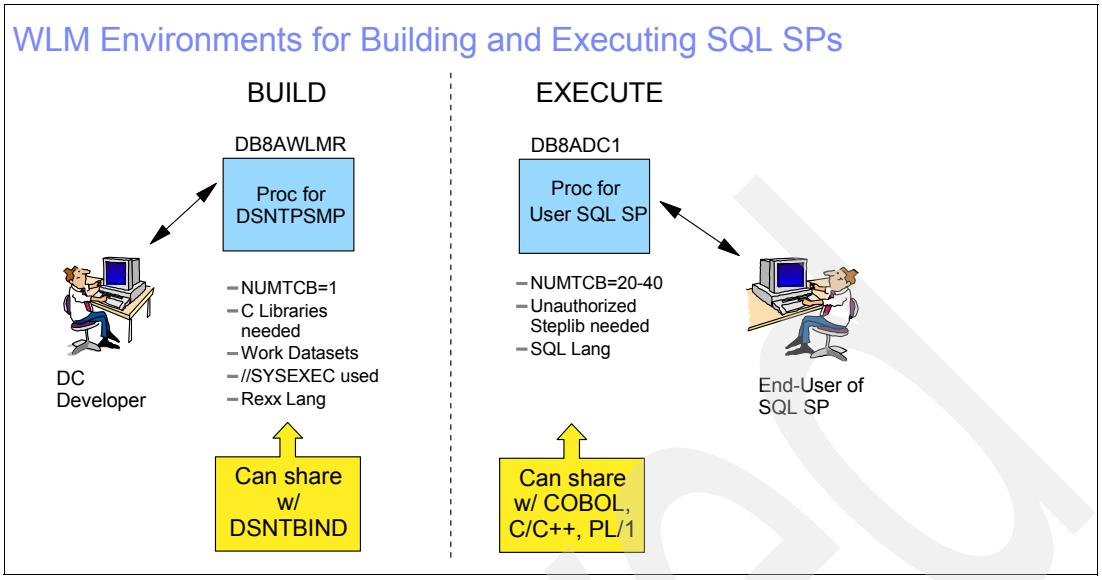


Figure 4-1 WLM definitions for build and for execute SQL, COBOL, C/C++, PL/1 SPs

Figure 4-2 illustrates using one WLM Application Environment for running DSNTJSPP, one WLM Application Environment for executing DSNTBIND using Development Center, and a third WLM Application Environment for executing the user Java stored procedure.

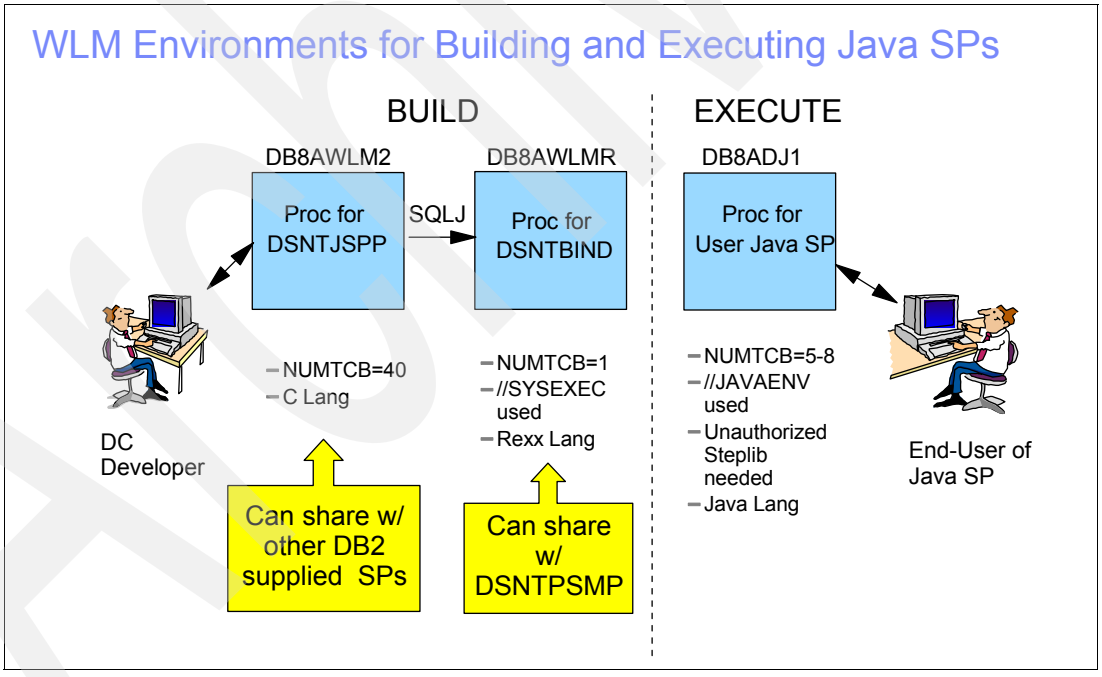


Figure 4-2 Three WLM definitions, two for building and one for executing Java SP

NUMTCB recommendations for different languages

The NUMTCB value that you choose for each application environment will vary by language, as we have discussed in this chapter. Table 4-2 provides general recommendations for the WLM proc NUMTCB setting for the different language stored procedures. These recommendations are based on available resources and should be tuned accordingly.

Table 4-2 NUMTCB recommendations for user stored procedures

Language	NUMTCB	Comments
REXX	1	REXX stored procedures must run in a WLM proc with NUMTCB = 1. If they execute in a proc with NUMTCB>1, unpredictable results, such as an 0C4 will occur.
SQL, COBOL, C/C++, PL/1	10-40	The compiled languages listed here, can share WLM procs when they have similar processing and JCL requirements.
Java	5-8	A Java Virtual Machine (JVM) is loaded for each NUMTCB when the WLM proc includes a JAVAENV DD. The JAVAENV DD is required for executing Java stored procedures.

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 Development Center and WebSphere Studio Application Developer.

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, as the JCL in this procedure is unique to DB8A. 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, it is recommended to 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 . : DB8AWLM
 Description  . . . . . . . . . . General DB2 SPs
 Subsystem Type . . . . . . . . . DB2
 Procedure Name . . . . . . . . . DB8AWLM
 Start Parameters . . . . . . . . DB2SSN=&IWMSSNM,APPLENV=DB8AWLM

 Limit on starting server address spaces for a subsystem instance:
 1   1.  No limit
     2.  Single address space per system
     3.  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 datasets, so we run the DB2 system WLM_REFRESH.stored procedure here as well, which requires all APF authorized STEPLIB datasets. 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 definition need re-installing z/OS service policy at LPAR or sysplex level. If the LE run-time 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.
//*
//*************************************************************
//DB2GWLM  PROC RGN=0K,APPLENV=DB2GWLM,DB2SSN=DB2G,NUMTCB=40
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV',REGION=0M
//STEPLIB  DD  DISP=SHR,DSN=DB2G7.SDSNEXIT
//         DD  DISP=SHR,DSN=DB2G7.SDSNLOAD
//*        DD  DISP=SHR,DSN=CEE.SCEERUN
```

Example 4-3 is a sample procedure for executing DSNTPSMP and DSNTBIND stored procedures which 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

```
/DB2GWLMR PROC RGN=OK,APPLENV=DB2GWLMR,DB2SSN=DB2G,NUMTCB=1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=SG247083.DEVL.LOAD
//         DD  DISP=SHR,DSN=DB2G7.SDSNEXIT
//         DD  DISP=SHR,DSN=DB2G7.SDSNLOAD
//*        DD  DISP=SHR,DSN=CEE.SCEERUN
//SYSTSPRT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSABEND DD  DUMMY
//SYSEXEC  DD  DISP=SHR,DSN=DB2G7.SDSNCLST <== Location of DSNTPSMP and DSNTBIND
//DSNTRACE DD  SYSOUT=*
//CFGTPSMP DD DISP=SHR,DSN=DB2GU.CFGTPSMP    <== ConfigurationFile
//**** Data sets required by the SQL Procedure Processor
//SQLDBRM  DD  DISP=SHR,DSN=SG247083.DEVL.DBRM <== DBRM Library
//SQLCSRC  DD  DISP=SHR,DSN=SG247083.SRCLIB.DATA      <== Generated C Source
//SQLLMOD  DD  DISP=SHR,DSN=SG247083.DEVL.LOAD      <== Application Loadlib
//SQLLIBC  DD  DISP=SHR,DSN=CEE.SCEEH.H            <== C header files
//         DD  DISP=SHR,DSN=CEE.SCEEH.SYS.H
//SQLLIBL  DD  DISP=SHR,DSN=CEE.SCEELKED                 <== Linkedit includes
//         DD  DISP=SHR,DSN=DB2G7.SDSNLOAD
//SYSMSGS  DD  DISP=SHR,DSN=CEE.SCEEMSGP(EDCPMSGE)  <== Prelinker msg file
//**** Workfiles required by the SQL Procedure Processor
//SQLSRC   DD  UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=FB,LRECL=80)
//SQLPRINT DD  UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=VB,LRECL=137)
```

```
//SQLTERM  DD  UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=VB,LRECL=137)
//SQLOUT   DD  UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=VB,LRECL=137)
//SQLCPRT DD UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=VB,LRECL=137)
//SQLUT1  DD  UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=FB,LRECL=80)
//SQLUT2  DD UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=FB,LRECL=80)
//SQLCIN   DD  UNIT=SYSDA,SPACE=(32000,(20,20))
//SQLLIN DD UNIT=SYSDA,SPACE=(8000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//SQLDUMMY DD
UNIT=SYSDA,SPACE=(16000,(20,20)),DSORG=PS,DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160
//SYSMOD   DD  UNIT=SYSDA,SPACE=(16000,(20,20)),DCB=(RECFM=FB,LRECL=80)      <= PRELINKER
```

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

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

```
//DB2GDC1 PROC RGN=OK,APPLENV=DB2GDC1,DB2SSN=DB2G,NUMTCB=10
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=SG247083.DEVL.LOAD
//         DD  DISP=SHR,DSN=DB2G7.SDSNEXIT
//         DD  DISP=SHR,DSN=DB2G7.SDSNLOAD
//*        DD  DISP=SHR,DSN=CEE.SCEERUN
//SYSTSPRT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSABEND DD DUMMY
//SQLDUMMY DD  DUMMY
```

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

```
//DB2GDJ1 PROC  DB2SSN=DB2G,NUMTCB=1,APPLENV=DB2GDJ1
//DB2GDJ1 EXEC  PGM=DSNX9WLM,TIME=1440,
//           PARM='&DB2SSN,&NUMTCB,&APPLENV',REGION=0M
//STEPLIB  DD    DISP=SHR,DSN=SG247083.DEVL.LOAD
//         DD    DISP=SHR,DSN=DB2G7.SDSNLOD2
//         DD    DISP=SHR,DSN=DB2G7.SDSNEXIT
//         DD    DISP=SHR,DSN=DB2G7.SDSNLOAD
//*        DD    DISP=SHR,DSN=CEE.SCEERUN
//JAVAENV  DD    DISP=SHR,DSN=SG247083.JAVAENV
//JSPDEBUG DD    SYSOUT=*
//CEEDUMP  DD    SYSOUT=*
//SYSPRINT DD    SYSOUT=*
//SYSOUT   DD    SYSOUT=*
```

5

Language Environment setup

In this chapter we provide an overview of Language Environment for z/OS Version 1, Release 4, including the run-time options that impact stored procedure development.

This chapter contains the following:

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

5.1 Language Environment concepts

Language Environment (LE) is a component of the OS/390 and z/OS operating systems. LE establishes a common run-time environment for different programming languages. It combines essential run-time services, such as routines for run-time 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 run-time environment for your applications, regardless of the application's programming language or system resource needs.

DB2 uses LE to provide a run-time 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 33. Since multiple languages can share the same LE run-time library, you do not have to specify the language-specific libraries in the JCL procedure of each stored procedure address space.

A LE run-time library is required for WLM-managed stored procedure address spaces, and it must be the only run-time library available. You must not reference any other language run-time libraries within the system link list or within the joblib or steplib for the stored procedure started task. If other language run-time 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 run-time 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 run-time 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 run-time options, including those options needed to use tools to debug your stored procedures.

5.2 Language Environment run-time options

Each high level language (COBOL, PL/I, C) has a number of run-time options used to control the execution environment for applications written in each high level language. The default values for each of the LE run-time options is documented in *z/OS V1R4.0 Language Environment Customization*, SA22-7564-04. Note that you may already be overriding the z/OS default values for your LE run-time 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 run-time 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 run-time 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 run-time options are those that are most frequently overridden for DB2 stored procedure.

5.2.1 MSGFILE

The MSGFILE run-time option specifies the *ddname* of the file that contains run-time 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 datasets is to maintain one MSGFILE data set and specify the ENQ sub-option within your MSGFILE run-time option on your stored procedure DDL. Specifying the ENQ sub-option resolves the JES spool serialization error without requiring you to maintain multiple MSGFILE datasets. See "Chapter 12" of *z/OS V1R4.0 Language Environment Customization*, SA22-7564-04 for details on when the ENQ option should be used.

5.2.2 RPTOPTS

The RPTOPTS run-time option generates a report of the run-time options in effect while the application was running. The report is directed to the *ddname* specified in the MSGFILE run-time 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 run-time 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 DB2 Development Center, you need to specify the LE run-time 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 463 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 463 for more details.

5.2.4 Options to limit storage required by LE at execution time

There are a number of LE run-time options that you can specify to minimize storage usage below the 16 MB line. They are documented in “Chapter 25, Using stored procedures for cliet/server processing” of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415. 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 run-time 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 Run-time options shown in SYSIBM.SYSROUTINES

Note that stored procedures EMPDTLSC, EMPDTLSJ, and EMPRSETJ will use the installation default values for the LE run-time 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 31” of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

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 is the last version of DB2 that will support DB2-established stored procedure address spaces. Starting with DB2 Version 8, all new stored procedures must be defined in a WLM-established address space; therefore, all stored procedures will 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). Steps to implement RRS were previously documented in the redbook *Getting Started with DB2 Stored Procedures: Give Them a Call through the Network*, SG24-4693-01. This section reproduces the steps required for implementing RRS that were documented in SG24-4693-01, updating the examples to show the jobs that were used for the RRS environment used for our test cases.

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

As with the implementation of DB2-established stored procedure address spaces, 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. For details on placing the log streams on disk, see the redbook *Getting Started with DB2 Stored Procedures: Give Them a Call through the Network*, SG24-4693-01. To place the log streams in the Coupling Facility, you must:
   - Add definitions for the structure in the CFRM policy.
   - Define the log streams.
   - Activate the new definitions.
2. Set up the RRS procedure in SYS1.PROCLIB.
3. 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 V1R4.0 MVS Setting Up a Sysplex*, SA22-7625-06, and “Understanding RRS Logging Requirements” in *z/OS V1R4.0 MVS Programming: Resource Recovery*, SA22-7616-02.

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

- Delayed unit-of-recovery log state stream:

  ATR.*gname*.DELAYED.UR

- Resource manager data log stream:

  ATR.*gname*.RM.DATA

- Restart log stream:

  ATR.*gname*.RESTART

- Archive log stream (This log is recommended but optional.)

  ATR.*gname*.ARCHIVE

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 will be 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,SUB=MSTR
```

You can stop RRS with the following operator command:

```
SETRRS CANCEL
```

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.
IEF450I RRS RRS - ABEND=S5C4 U0000 REASON=FFFF2222 064
IEF404I RRS - ENDED - TIME=hh.mm.ss
```

```
IEF196I IEF472I RRS RRS - COMPLETION CODE - SYSTEM=5C4 USER=0000
IEF196I REASON=FFFF2222
IEF196I IEF373I STEP/RRS /START 1997303.1507
IEF196I IEF374I STEP/RRS /STOP 1997303.1905 CPU 0MIN 03.02SEC
```

Notice that RRS abends with S5C4 code. No action is necessary for reason code X'FFFF2222'.

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.

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 DB2 Development Center, see “Development Center authorization set up” on page 507.

This chapter contains the following:

- Workload Manager security requirements
- Privileges required to create stored procedures
- Privileges required to execute stored procedures
- Additional stored procedure security considerations

See the Security section of the *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413 for more information on these topics.

7.1 Workload Manager security requirements

DB2 stored procedures that run on z/OS or OS/390 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 will no longer support the creation of *new* stored procedures in the DB2-established address space, we focus our attention in this chapter 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 to 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 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.DB2G.WLMENV)
  ```

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

  ```
  RDEFINE SERVER (DB2.DB2G.DB2DEC1)
  ```

- Activate the resource class:

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

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

  ```
  PERMIT DB2.DB2G.DB2DEC1 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 control application developers from creating stored procedures in production application environments. The RACF command we used to protect WLM application environment DB2GDEC1 on subsystem DB2G is:

```
RDEFINE DSNR (DB2G.WLMENV.DB2GDEC1) 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 DB2GDEC1:

```
PERMIT DB2G.WLMENV.DB2GDEC1 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 an OS/390 command line, or by executing the DB2-supplied WLM_REFRESH stored procedure. We issued the following command to refresh application environment DB2GDEC1, which contains the majority of the COBOL stored procedures in our test cases:

```
/V WLM,APPLENV=DB2GDEC1,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 DB2GDEC1 on subsystem DB2G is shown in Example 7-1.

Example 7-1 Permit access to WLM_REFRESH resource profile

```
RDEFINE DSNR (DB2G.WLM_REFRESH.DB2GDEC1)
PE DB2G.WLM_REFRESH.DB2GDEC1 +
   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;
```

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 10, “Verifying with the sample applications” in *DB2 UDB for z/OS Version 8 Installation Guide,* GC18-7418 for details on job DSNTEJ6W. See Appendix B.2, “Refresh a WLM environment with DB2WLMRefresh” on page 605 in this redbook for more details on WLM_REFRESH.

7.2 Privileges required to create stored procedures

Stored procedures are typically created by a DBA or an application programmer, depending on how roles and responsibilities are defined for an 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 current authid in effect 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.

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
```

7.2.2 BINDADD privilege for stored procedures that contain SQL

If the stored procedure being created contains SQL statements then 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 secondary authid, which represents a group of users who create stored procedures.

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 qualified procedure name, as shown here:

    CALL *procedure-name*

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

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

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 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

In our example in 7.2, “Privileges required to create stored procedures” on page 57, we granted user ID PAOLORW the privileges required to create stored procedure EMPDTLSC in 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 did not have EXECUTE authority on the stored procedure. Figure 7-1 shows the error message that was returned to the Windows client.

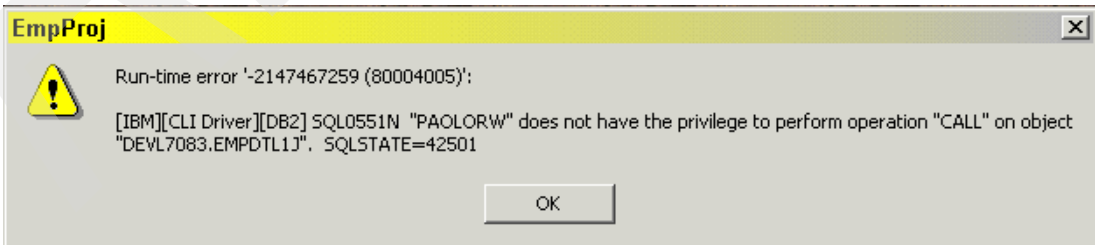


Figure 7-1 Sample error message on Windows client when EXECUTE privilege does not exist

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.

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.

For more details please refer to *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

7.3.2 Privileges to execute a stored procedure called statically

For static SQL programs that use the syntax CALL procedure, the owner 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 they 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 had been granted no access to any DB2 packages or plans. 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 DB2G 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
```

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

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

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

7.3.3 Autorization 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:

- The owner (the definer) of the stored procedure
- The owner of the plan that contains the statement that invokes the package if the application is local, the application is distributed and DB2 for z/OS is both the requester and the server, or the application uses Recoverable Resources Management Services attachment facility (RRSAF) and has no plan.
- 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

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 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

7.4 Additional stored procedure security considerations

In this section we discuss some additional considerations with regards 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

7.4.1 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.

7.4.2 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.

7.4.3 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
```

7.4.4 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. You must make sure that you set caps off prior to issuing the GRANT statement.

See 17.5, "Preparing Java stored procedures" on page 255 for more details on preparing jar files and using the DB2-supplied INSTALL_JAR stored procedure.

7.4.5 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 run-time environment, determines what values apply at run time 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 run-time environment affect dynamic SQL statement behavior when the statement is in a package that is invoked from a stored procedure (or user-defined function).

Table 7-1 How is run-time behavior determined?

DYNAMICRULES value	**Stored procedure or user-defined function environment**	**Authorization ID**
BIND	Bind behavior	Plan or package owner
RUN	Run behavior	Current SQLID
DEFINEBIND	Define behavior	Owner of user-defined function or stored procedure
DEFINERUN	Define behavior	Owner of user-defined function or stored procedure
INVOKEBIND	Invoke behavior	Authorization ID of invoker
INVOKERUN	Invoke behavior	Authorization ID of invoker

The DYNAMICRULES option along with the run time environment of a package (whether the package is run stand-alone or under the control of a stored procedure or user-define 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 run-time behavior means

Dynamic SQL attribute	Bind behavior	Run behavior	Define behavior	Invoke behavior
Authorization ID	Plan or package owner	Current SQLID	Owner of user-defined function or stored procedure	Authorization ID of invoker
Default qualifier for unqualified objects	Bind OWNER or QUALIFIER value	Current SQLID	Owner of user-defined function or stored procedure	Authorization ID of invoker
CURRENT SQLID	Not applicable	Applies	Not applicable	Not applicable
Source for application programming options	Determined by DSNHDECP parameter DYNRULS	Install panel DSNTIPF	Determined by DSNHDECP parameter DYNRULS	Determined by DSNHDECP parameter DYNRULS
Can execute GRANT, REVOKE, 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.

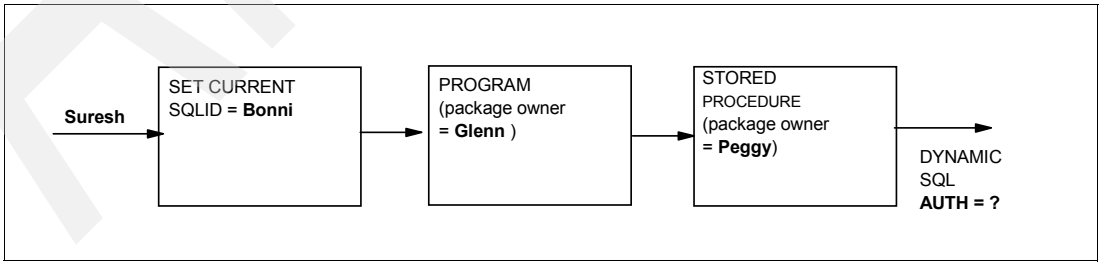


Figure 7-2 Security implications of dynamic SQL in a stored procedure

Consider a user, Suresh, using a current SQLID of Bonni, who executes a package bound by Glenn. This package calls a stored procedure bound by Peggy. Whose authority is checked at run time? Depending on the option chosen, the authorization ID used to determine whether or not the execution of dynamic SQL within this stored procedure is permitted could be:

- Suresh (invoker)
- 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.

7.4.6 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 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

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
- 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 is 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 store procedure address space. In this case, you must restart the stored procedures address space.

The method you use to perform these tasks is dependent on whether or not you are using WLM and whether it is operating in goal mode or compatibility mode. We will discuss the actions you must take for each of the three cases below. See *z/OS MVS Planning: Workload Management* for more information about the command VARY WLM.

If the stored procedure abends a specified number of times (see 8.4, “Handling application failures” on page 71 for details), DB2 places it in a STOPABND 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
```

8.1.1 WLM-established address spaces with WLM in goal mode

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 also 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:

- 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 B.2, “Refresh a WLM environment with DB2WLMRefresh” on page 605 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 re-start of environment DB2GDEC1:

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

8.1.2 Handling error conditions in the application environment

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, 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 message appears even in response to the VARY commands. The key word to look in the message is `IWM032I Internal` stop to distinguish between the VARY command and WLM stopping the AE.

8.1.3 WLM-established address spaces with WLM in compatibility mode

Use this z/OS command to stop the stored procedures address space:

```
/CANCEL DB2GDEC1
```

Follow this with:

```
/START DB2GDEC1
```

8.1.4 DB2-established address spaces

Use the DB2 commands STOP PROCEDURE and START PROCEDURE to perform the refresh as shown in the example below:

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

This results in:

```
DSNX947I  -DB2G DSNX9SP2 STOP PROCEDURE SUCCESSFUL FOR
DEVL7083.EMPDTLSC
DSN9022I  -DB2G DSNX9COM '-STOP PROC' NORMAL COMPLETION
```

Follow this with:

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

This results in:

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

8.2 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 *n* (where *n* is an integer from 1 up to 2 Giga) 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), this 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.

8.3 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.

   Tip: Apply PTFs UK01174 (DB2 V7) and UK01175 (DB2 V8) for APAR PQ99524. It reduces the chances of abends or subsystem termination when cancelling a thread.

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.
4. Cancel the WLM application environment where the problem stored procedure is executing. It should be the only one left when the refresh was issued.

8.4 Handling application failures

Attention: This option is available in DB2 V8 only.

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 zparm STORMXAB on installation panel DSNTIPX. V8 introduces a new parameter that allows you to control this behavior at the 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 on installation panel DSNTIPX. This is the default and 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 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.

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 procedure

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 a stored procedures in COBOL, C, Java, SQL language, and REXX. For creating and coding stored procedures in Java refer to Part 4, “Java stored procedures” on page 243.

This part contains the following chapters:

9

Defining stored procedures

In order for a 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 some of the 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 27, “Using triggers and UDFs” on page 451 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 UDB for z/OS Version 8 SQL Reference,* SC18-7426 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 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. For each parameter, we discuss meaning, choices, and recommendations.

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

9.1.1 The install panel

Figure 9-1 shows the DB2 *routine parameter* install panel DSNTIPX for DB2 V8. The entries on this panel are used to generate the sample JCL used start the stored procedures address space where to run stored procedures or user-defined functions. The values shown are the default values. Notice that, like most DB2 system parameters, options 4 through 8 are online zparms, and can be modified after the installation without stopping DB2.

```
                          ROUTINE PARAMETERS
===>

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

* 1 WLM PROC NAME    ===>  ssnWLM    WLM-established stored procedure JCL PROC
* 2 DB2 PROC NAME    ===>            DB2-established stored procedure JCL PROC
  3 NUMBER OF TCBS   ===>     8      Number of concurrent TCBs (1-100)
  4 MAX ABEND COUNT  ===>     0      Allowable ABENDs for a procedure (0-255)
  5 TIMEOUT VALUE    ===>   180      Seconds to wait before SQL CALL or
                                     function invocation fails (5-1800,NOLIMIT)
  6 WLM ENVIRONMENT  ===>                    Default WLM environment name
  7 MAX OPEN CURSORS ===>   500      Maximum open cursors per thread
  8 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. The default procedure will be named by appending the string WLM to the DB2 subsystem name.
- 2 **DB2 PROC NAME**: It specifies a name for the JCL procedure that is used to start the DB2-established address space. In DB2 V8, support for DB2-established address spaces is deprecated. If you are installing DB2, this field is blank and cannot be updated. If you are migrating from DB2 Version 7, you can change this field if required. Existing stored procedures can still run in a DB2-established stored procedure address space, but you should move them to WLM environments as soon as possible. If you CREATE or ALTER

an existing stored procedure, it cannot run in a DB2-established stored procedure address space.

- 3 **NUMBER OF TCBS**: It specifies how many SQL CALL statements or an invocation of a user-defined function can be processed *concurrently* in one address space. This value is limited by the USS MAXPROCUSER (maximum number of processes for the user.) value. With V8, the value specified in NUMTCB is sent to WLM as a maximum task limit. See also 22.3.2, “Exploit WLM server task thread management” on page 354.
- 4 **MAX ABEND COUNT**: 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. (DSNZPARM parameter STORMXAB). 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 UDF equally. However, with DB2 V8, you can specify a value for each stored procedure or UDF as shown in 22.3.1, “Maximum failures” on page 352.
- 5 **TIMEOUT VALUE**: 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.
- 6 **WLM ENVIRONMENT**: It specifies the name of the WLM_ENVIRONMENT to use for stored procedure when a value is not given for the WLM_ENVIRONMENT option on the CREATE FUNCTION or CREATE PROCEDURE statements.
- 7 **MAX OPEN CURSORS**: 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 only applicable to DB2 V8.
- 8 **MAX STORED PROCS**: 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 only applicable to DB2 V8.

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 have introduced 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 the *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

Figure 9-2 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.

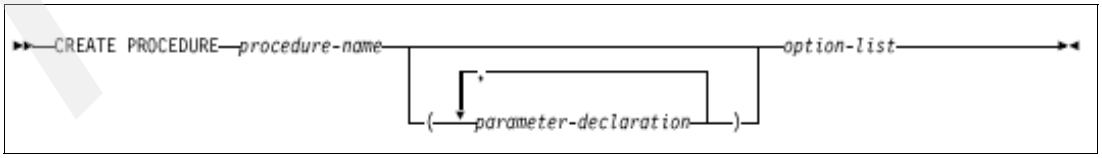


Figure 9-2 The CREATE PROCEDURE statement structure

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

CREATE PROCEDURE (EXTERNAL) option list

Figure 9-3 shows the CREATE PROCEDURE option list for an external stored procedure. This list is for DB2 V8 and includes the new option STOP AFTER SYSTEM DEFAULT FAILURES.

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

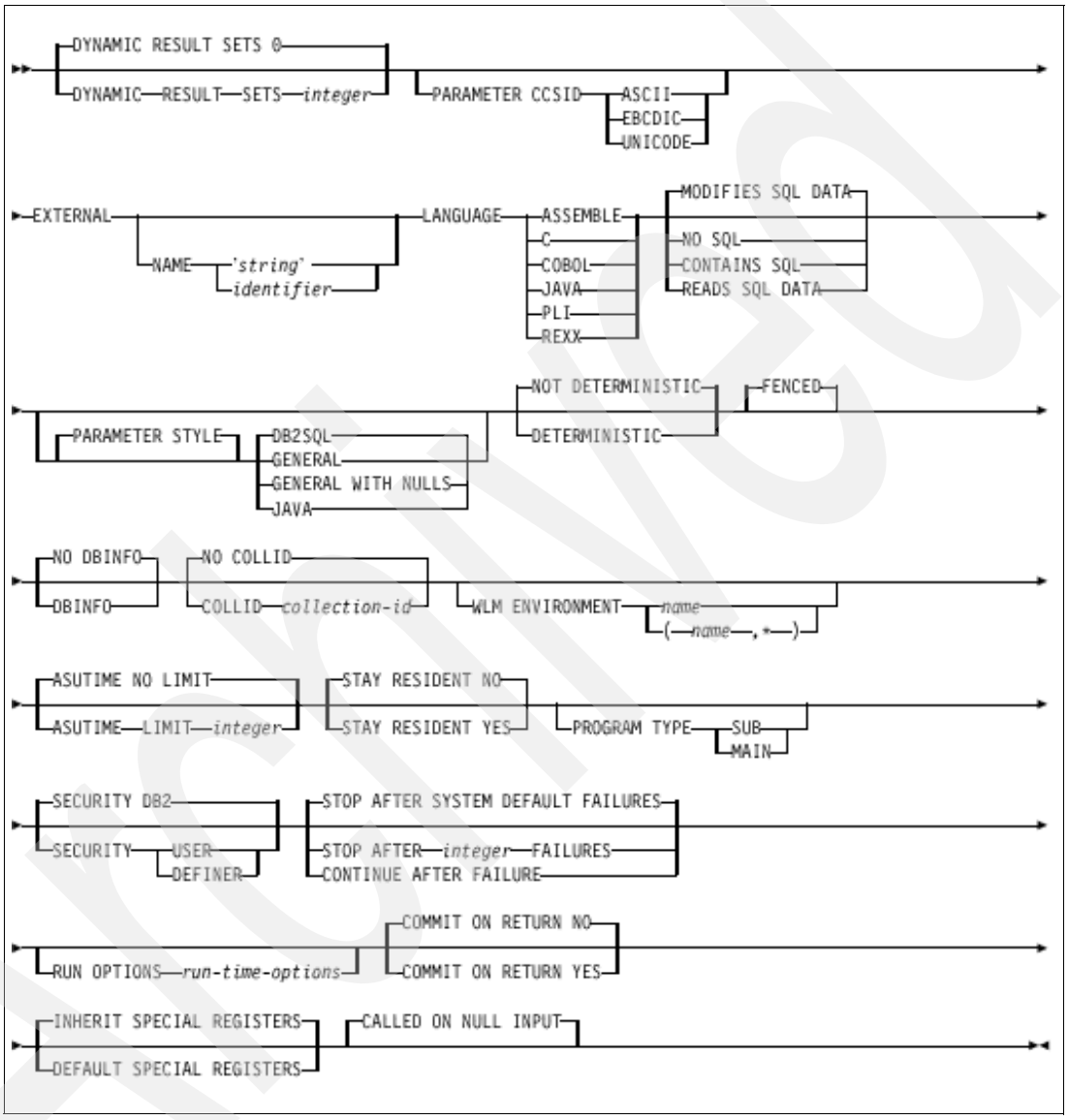


Figure 9-3 The option list for CREATE and ALTER PROCEDURE EXTERNAL

Note: With DB2 V7, COMPJAVA was listed among the supported languages, but it is no longer supported with V8. Users need to migrate to JAVA.

With DB2 V8, PARAMETER and STYLE keywords are required on the PARAMETER STYLE clause (they were optional with V7). Also, SQL has been added as a synonym for DB2SQL in the PARAMETER STYLE clause.

CREATE PROCEDURE (SQL) option list

Figure 9-4 shows the CREATE PROCEDURE option list for an SQL stored procedure. This list is also for DB2 V8, and includes the new option STOP AFTER SYSTEM DEFAULT FAILURES.

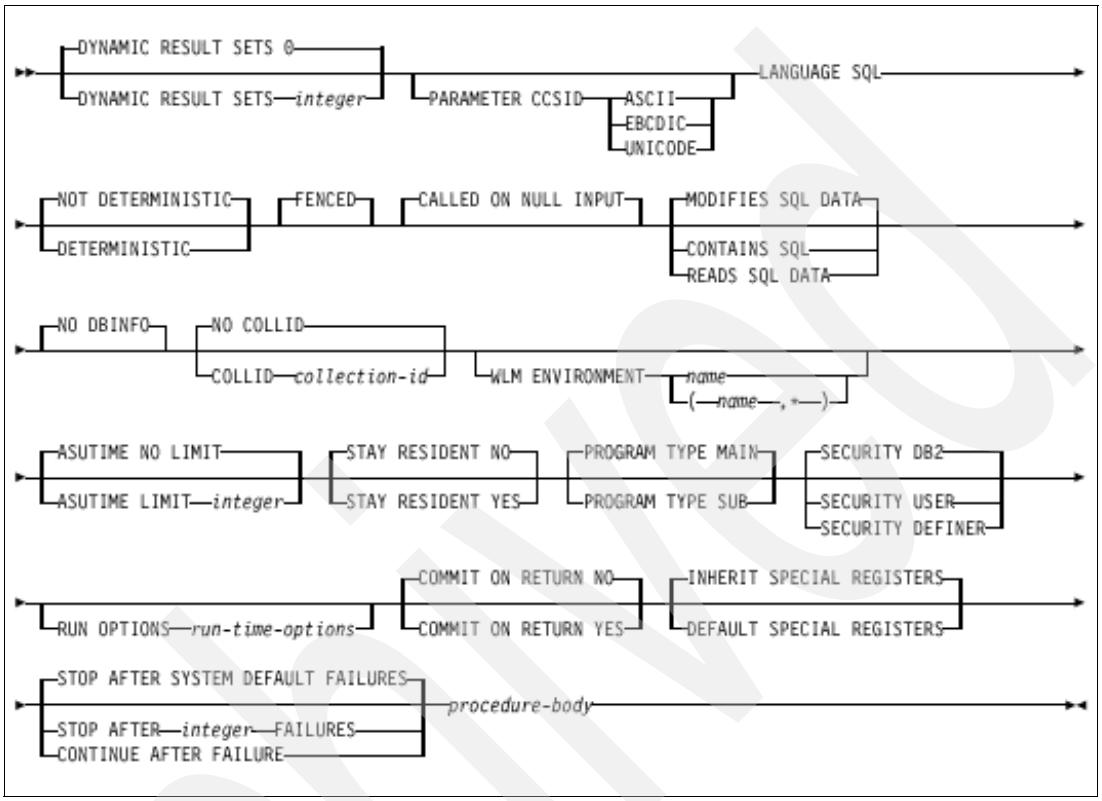


Figure 9-4 The option list for CREATE and ALTER PROCEDURE SQL

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.

There are no known implications of 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 point in 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. In this case, DBINFO, PROGRAM TYPE MAIN, and RUN OPTIONS are not permissible parameters.

When LANGUAGE REXX is specified, PARAMETER STYLE of DB2SQL 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 A” of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426 for details on which statements are allowed depending on the value of this parameter.

There are no known performance implications of 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:

DB2SQL 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 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 DB2SQL for REXX stored procedures.

Figure 9-5 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.

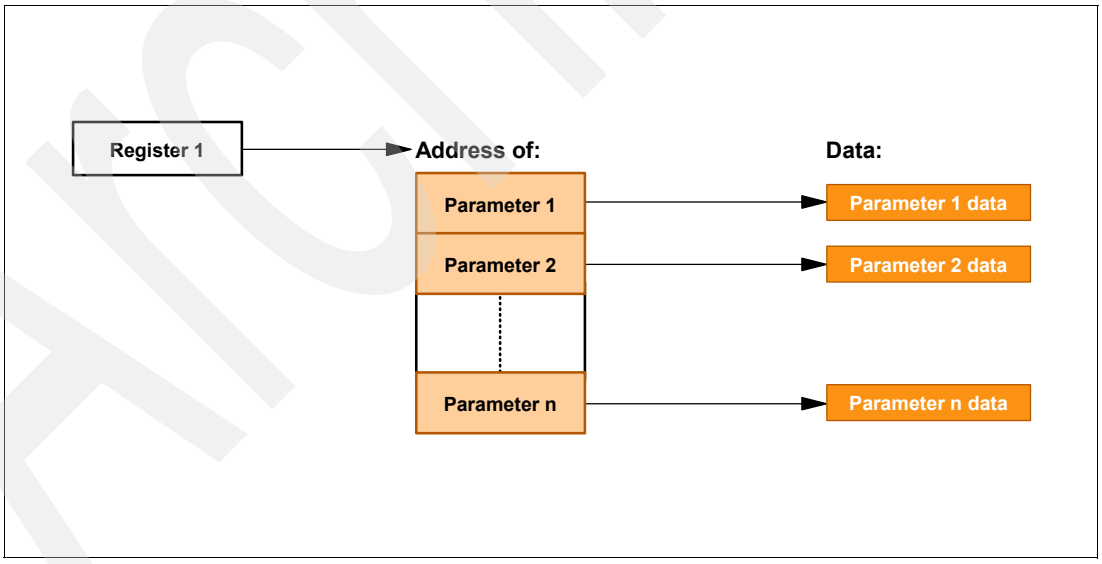


Figure 9-5 Parameter convention GENERAL for a stored procedure

Figure 9-6 shows the structure of the parameter list when PARAMETER STYLE GENERAL WITH NULLS is used.

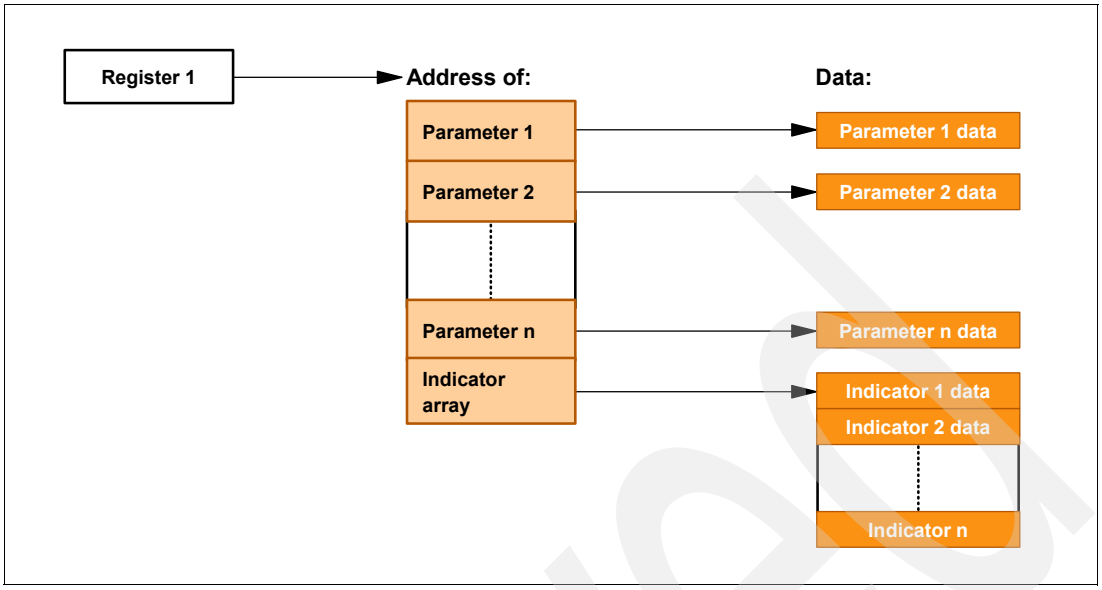


Figure 9-6 Parameter convention GENERAL WITH NULLS for a stored procedure

Figure 9-7 shows the structure of the parameter list when PARAMETER STYLE DB2SQL is used.

Important: Remember that when using parameter style DB2SQL, an array of indicator variables is not supported, you must specify an elementary item for each indicator variable.

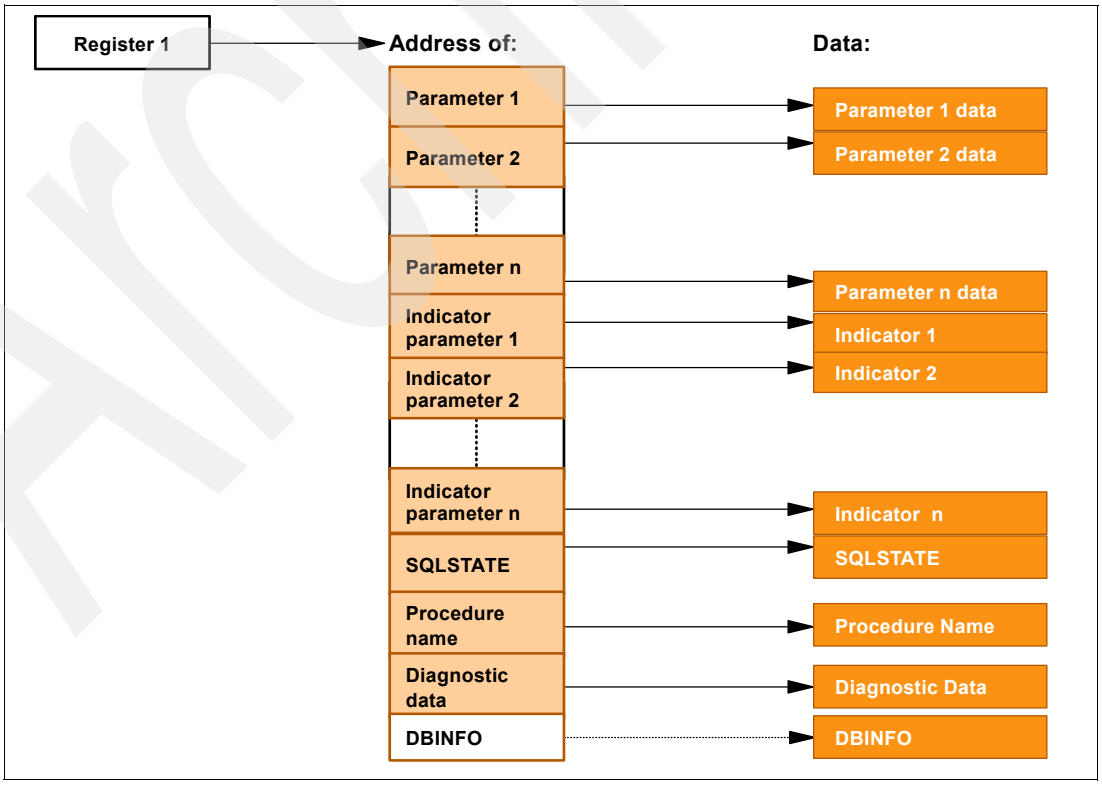


Figure 9-7 Parameter convention DB2SQL for a stored procedure

Figure 9-8 shows the structure of the parameter list when PARAMETER STYLE JAVA is used. The list of ResultSet parameters is optional.

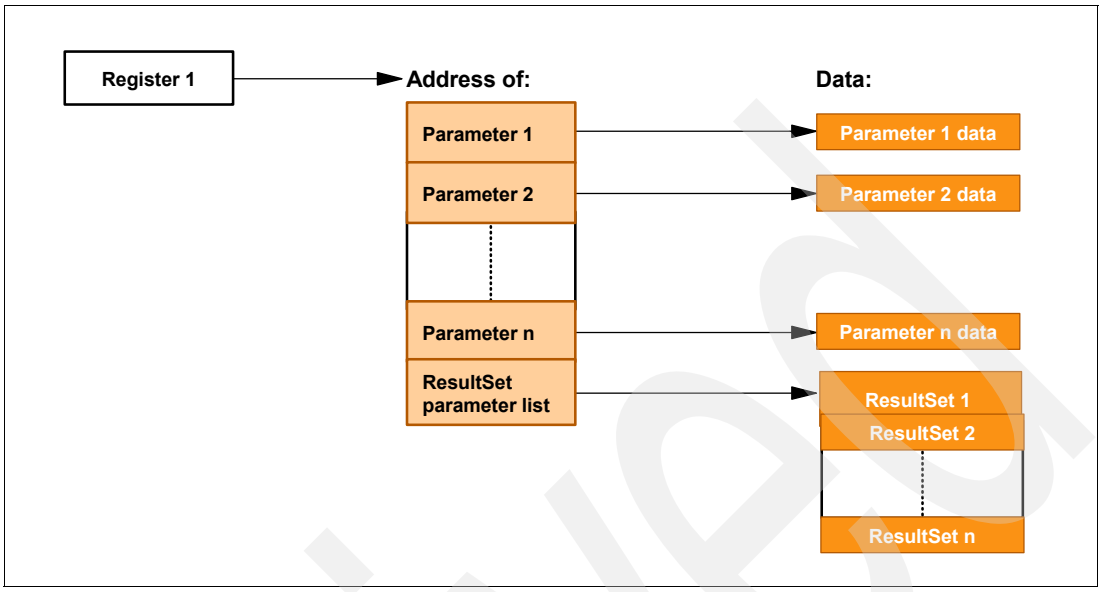


Figure 9-8 Parameter convention GENERAL WITH NULLS for a stored procedure

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`. `NOT DETERMINISTIC` 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 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 run-time 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 DB2SQL` is specified. `NO DBINFO` is the default.

9.1.9 Collection ID 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 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 DB2 uses the list of collection IDs specified in the PKLIST in the specified order. This process is specially 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 till 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 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 needs 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.

9.1.10 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 MVS Initialization and Tuning Guide,* SA22-7591-01 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.

9.1.11 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 stored procedures.

9.1.12 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 below:

- 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.

9.1.13 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.

9.1.14 Max number of failures (new in DB2 V8)

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 zparm STORMXAB on installation panel DSNTIPX. V8 introduces a new parameter that allows you to control this behavior at the 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 on installation panel DSNTIPX. This is the default and 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 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.

9.1.15 Run-time options

You can specify the Language Environment run-time options to be used for the stored procedure as a character sting 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 run-time options, and Language Environment uses its installation defaults. See 5.2, “Language Environment run-time options” on page 42, and Chapter 25 of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

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.

9.1.16 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, but `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 procedure defined with `COMMIT ON RETURN YES`.

9.1.17 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.

Section “Using special registers in a stored procedure” in Chapter 25 of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415 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.

9.1.18 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 80.

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.

9.1.19 WLM environment

WLM ENVIRONMENT name can be specified in two ways

- WLM ENVIRONMENT name

or

- WLM ENVIRONMENT (name, *)

Attention: APARs OA04555 for WLM and PQ80631 (PTF UQ90158 for DB2 V7 and UQ90159 for DB2 V8) provide changes which make the second specification a valid alternative.

9.1.20 Naming your stored procedure

With DB2 V7, the name of the stored procedure can be up to 18 characters. A stored procedure, external or SQL, is associated with a load module on z/OS server. Also if the stored procedure is a DB2 program, it contains a package also. As there is one to one correspondence between stored procedure name, load module and DB2 package, it's recommended to use same name for all of them. DB2 package name and load module cannot be greater than eight characters. Hence restrict the stored procedure name also to 8 characters, if your organization naming standards permit. Having the same name helps in identifying the package and load module for a stored procedure.

As of DB2 V7, for SQL procedures, if you did not specify the `External Name` option in the CREATE PROCEDURE statement, DB2 takes the first eight characters of the stored procedure name for external name. However, if you use DC or SPB or DSNTPSMP in batch mode, this issue is covered since intelligence is built into the DSNTPSMP stored procedure, so as to not allow a duplicate external name for the fully qualified stored procedure (schema.stored procedure).

For example, the following two stored procedures will have the same external name (DETAILBY) in the DB2 catalog table:

```
CREATE PROCEDURE SCH.DETAILBY_SSN
(IN SSN char(9),
OUT NAME char(25)
)
Language SQL
(processing logic)


CREATE PROCEDURE SCH.DETAILBY_ACNO
(IN ACNO char(9),
OUT NAME char(25)
)
Language SQL
(processing logic)
```

However, for external stored procedure, the EXTERNAL clause is not optional. Development Center or Stored Procedure Builder generate external names in the format of `SQLxxxxx` where `xxxxx` is a number.

Note: With DB2 V8, the name of a DB2 stored procedure can be 128 bytes in V8 New Function Mode. The procedure name is no longer truncated to 8 bytes to obtain the EXTERNAL name if there is no EXTERNAL NAME clause. If the EXTERNAL NAME is specified or defaulted (to the full procedure name), it is checked for a valid MVS load module name, and sqlcode -449 is issued. The example above with DETAILBY is no longer valid for V8.

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.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(7,2)
,OUT PSQLCODE      INTEGER
,OUT PSQLSTATE     CHAR(5)
,OUT PSQLERRMC     VARCHAR(250)
)
DYNAMIC RESULT SETS 0
EXTERNAL NAME EMPDTLSC
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB2GDEC1
```

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 WLMENV1
     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)
)
DYNAMIC RESULT SETS 5
EXTERNAL NAME EMPRSETR
LANGUAGE REXX
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
WLM ENVIRONMENT DB2GDER1
STAY RESIDENT NO
COLLID DSNREXCS
PROGRAM TYPE MAIN
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF)'
```

Java stored procedure

See Example 9-6.

Example 9-6 Parameters for Java stored procedures CREATE

```
CREATE PROCEDURE DEVL7083.EMPRST1J ( IN  WORKDEPT   CHARACTER(3),
                                     OUT OUTPUTMESSAGE VARCHAR(250))
    EXTERNAL NAME 'EmpRst1J.GetEmpResult'
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    COLLID DEVL7083
    PROGRAM TYPE SUB
    DYNAMIC RESULT SETS 1
    WLM ENVIRONMENT DB2GWEJ1
```

SQL language stored procedure

See Example 9-7.

Example 9-7 Parameters for SQL language 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 DB2GDES1
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 SQL procedures as external C load modules you must always specify LANGUAGE SQL in the CREATE PROCEDURE, never re-deploy externally the module as a LANGUAGE C stored procedure. Unpredictable errors will happen.

9.3 Summary of recommendations

Based on the discussion above, our recommendations for the values for the parameters are summarized in Table 9-1.

Table 9-1 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 PROECDURE if SQL is added later. There are no known performance implications.
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 DB2SQL
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
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)

Parameter	Recommended value
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 YES for distributed applications (make sure the application is aware of the impact) and COMMIT ON RETURN NO for non-distributed applications
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 D, “Additional material” on page 651.
Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 22 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 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 redbook, we simply point to them here for convenience. They are:

1. WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 33 for details.
2. LE environment must be set up. See Chapter 5, “Language Environment setup” on page 41 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 75 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 55 for details.
6. Develop the calling application, if needed.
7. See Chapter 14, “Debugging” on page 173 for details on testing and debugging.

10.2 Developing COBOL stored procedures

In this section we discuss the development of stored procedure in COBOL. 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 DB2SQL
- Handling DBINFO parameter
- Handling result sets in the calling program

10.2.1 Passing parameters

Example 9-3 on page 89 shows our first sample COBOL stored procedure. More examples are available as described in Appendix D, “Additional material” on page 651.

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 76. 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 98. 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 it 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

No special processing is needed for *preparing* a stored procedure except for the following requirements:

- You must compile it with the NODYNAM option.
- You must linkedit the language interface module DSNRLI for the RRSAF.
- You must specify the parameter AMODE(31) when you linkedit it.

Note on DYNAM versus NODYNAM compiler option

The *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415 clearly states that you must use the NODYNAM option when coding COBOL programs that issue SQL statements under CICS or stored procedures. This is no longer strictly true, and we expect that the manual will be amended accordingly.

With the COBOL compile DYNAM option, COBOL will dynamically load modules that are external references, including the DB2 language interface code (DSNELI, DSNHLI, DSNRLI, etc.). By default, the DB2 precompiler will generate code that uses the external entry point name DSNHLI to invoke the DB2 language interface module. This is not the correct module for either CICS or stored procedures, hence the recommendation to use NODYNAM.

However, an enhancement to the precompiler allows the use of a name other than DSNHLI in the precompiler generated code. This is true for the ATTACH(RRSAF) keyword for the precompiler. With this enhancement, there is no issue in dynamically loading the RRS Attach DB2 language interface module.

DYNAM is supported for stored procedures, you just need to make sure that the correct DB2 language interface module is loaded dynamically by either:

- Using the ATTACH(RRSAF) precompiler option

  or

- Copying the DSNRLI module into a load library concatenated in front of the DB2 libraries using the member name DSNHLI

Some CICS installations use this approach to get DYNAM to work with CICS. We provide a complete example with stored procedures in “Solution 2: Dynamic invocation of language interface module” on page 122.

If the stored procedure contains SQL statements, you must process them like they are in any other SQL application program through the DB2 pre-compiler or the SQL statement co-processor, and you must bind the resulting DBRM into a package. It does not require a plan since it runs under the thread for the calling application. No special processing is needed for *binding* a stored procedure except for the following restrictions:

- 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.
- 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 55 for details.
- The collection ID associated with the stored procedure package must be based on the following rules:
  - If you specify NO COLLID when creating the stored procedure, the package must use the same collection ID as the calling program.
  - If you specify COLLID *collection_id* when creating the stored procedure, the stored procedure must use this *collection_id*.

Also, see 9.1.9, “Collection ID stored procedure runs in” on page 84 for details on the definition of the collection ID.

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, and this is discussed in 10.2.8, “Handling result sets in the calling program” on page 109.

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, and this is discussed in 10.2.8, “Handling result sets in the calling program” on page 109.

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 a 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 these additional parameters as 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 these indicator variables in the CALL as shown in Example 10-9. The indicators are matched with parameters strictly on 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, you must 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 DB2SQL

When you specify PARAMETER STYLE DB2SQL 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 procedure that contain the following information:

- SQLSTATE defined as CHAR(5)
- Qualified name of the stored procedure defined as VARCHAR(27)

  Notice that it is extended to VARCHAR(517) with DB2 V8. 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(18). It is the same as the unqualified name.

  Notice that it is extended to VARCHAR(128) with DB2 V8.
- SQL diagnostic string defined as VARCHAR(70)

Important: When you use PARAMETER STYLE DB2SQL, you must be aware of three important code requirements:

- The CREATE PROCEDURE ddl must ***not*** specify these additional parameters:

  ```
  ,OUT DSQLSTATE     CHAR(5)
  ,OUT DSPNAME       VARCHAR(27)
  ,OUT DSPECNAME     VARCHAR(18)
  ,OUT DDIAGMSG      VARCHAR(70)
  ```

- You must define the additional variables in the linkage section of the stored procedure.
- You must 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 DB2SQL

```
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.

Example 10-11 Parameter list of calling application using PARAMETER STYLE DB2SQL

```
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 these indicator variables as level 01 (highlighted in the example) as shown in Example 10-12.

Example 10-12 Parameter list in the linkage section using PARAMETER STYLE DB2SQL

```
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(27).
01  DSPECNAME.
    49  DSPECNAME-LEN     PIC S9(4) COMP.
    49  DSPECNAME-TEXT    PIC X(18).
01  DDIAGMSG.
    49  DDIAGMSG-LEN      PIC S9(4) COMP.
    49  DDIAGMSG-TEXT     PIC X(70).
```

The procedure division for the stored procedure must receive these additional parameters as shown in Example 10-13.

Example 10-13 Procedure division using the parameters using PARAMETER STYLE DB2SQL

```
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 DB2SQL, as shown in Example 10-14.

Example 10-14 SQL CALL COBOL example using PARAMETER STYLE DB2SQL

```
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 DB2SQL, 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 below 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
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

Stored procedure sets this SQLSTATE	Caller receives this SQLCODE	Caller receives this SQLSTATE	Sample SQLCA output
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
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

Stored procedure sets this SQLSTATE	Caller receives this SQLCODE	Caller receives this SQLSTATE	Sample SQLCA output
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 the SQLSTATE to value that can be misinterpreted by the calling application since it may not be aware of the fact 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 note 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 with DB2 V8: SQLCODE-463 is replaced for Java by SQLCODE -4302, SQLSTATE 38000.

10.2.7 Handling 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 DB2SQL 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 25" of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

- Location name

  A 128-byte character field containing the location to which the invoker is currently connected. Example:

  ```
  SP-DBINFO-LOCATION      =   DB2G
  ```

- 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
  ```

- 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        = DSN07010
  ```

- Operating system

  A 4-byte integer fields 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

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 DB2 Development Center):

  ```
  APPLICATION-ID=G9012726.PB05.008686193306
  or when called by alocal application:
  APPLICATION-ID=USIBMSC.SCPDB2G.BA7AF5C1BC78~/
  ```

The calling program does not change.

The stored procedure definition must include the parameters shown below:

```
PARAMETER STYLE DB2SQL
...
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  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(27).
01  D18.
    49  D18-LEN           PIC S9(4) COMP.
    49  D18-TEXT          PIC X(18).
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 Chapter 9, “Defining stored procedures” on page 75), 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 9.1.16, “Use of commit before returning” on page 87 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.

   ```
   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.

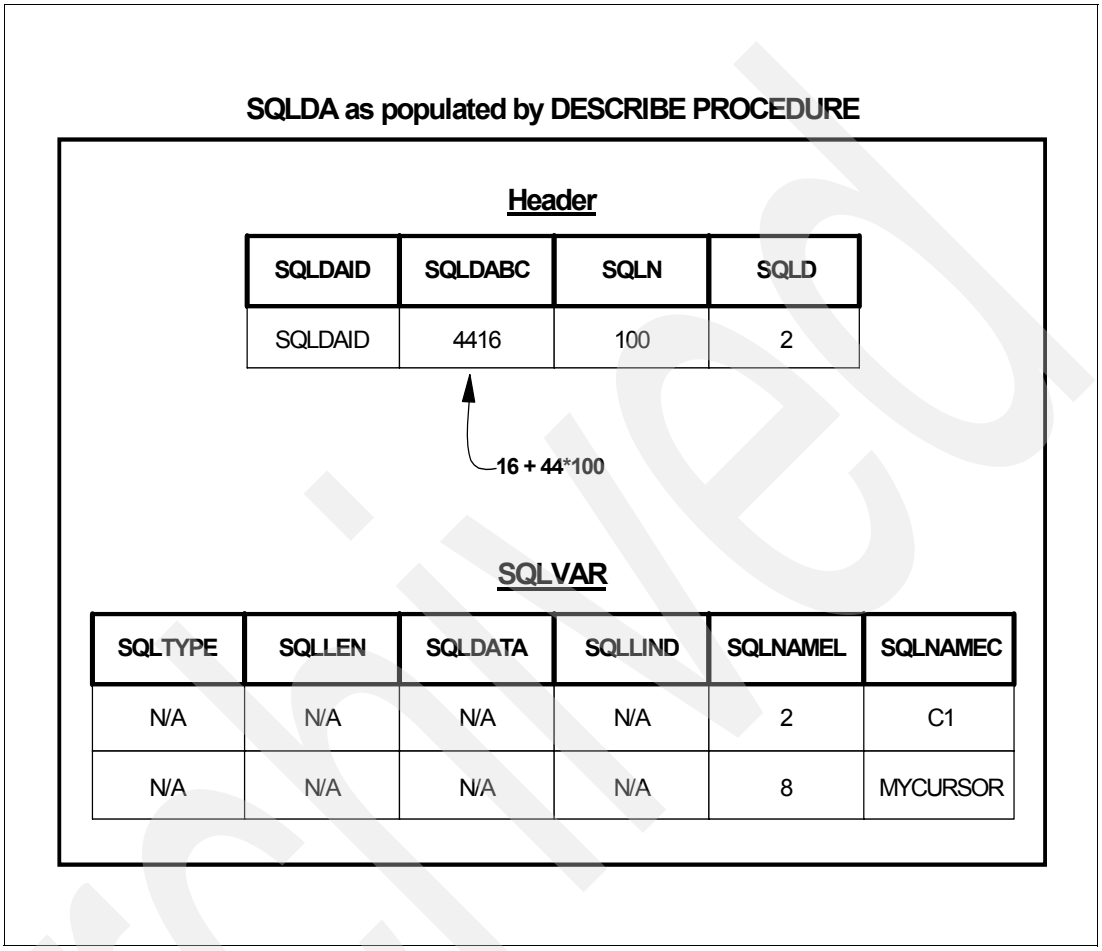


Figure 10-1 SQLDA as populated by the DESCRIBE PROCEDURE statement

6. SQLD contains the number of result sets returned by the stored procedure.
7. SQLNAME 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 as follows:

```
EXEC SQL ASSOCIATE LOCATORS (:LOC-EMPRSETC)
     WITH PROCEDURE EMPRSETC
END-EXEC.
```

9. Allocate a cursor for each result set to be processed as follows:

```
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.

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 a 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 contents of this section on the description of nested stored procedures 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.

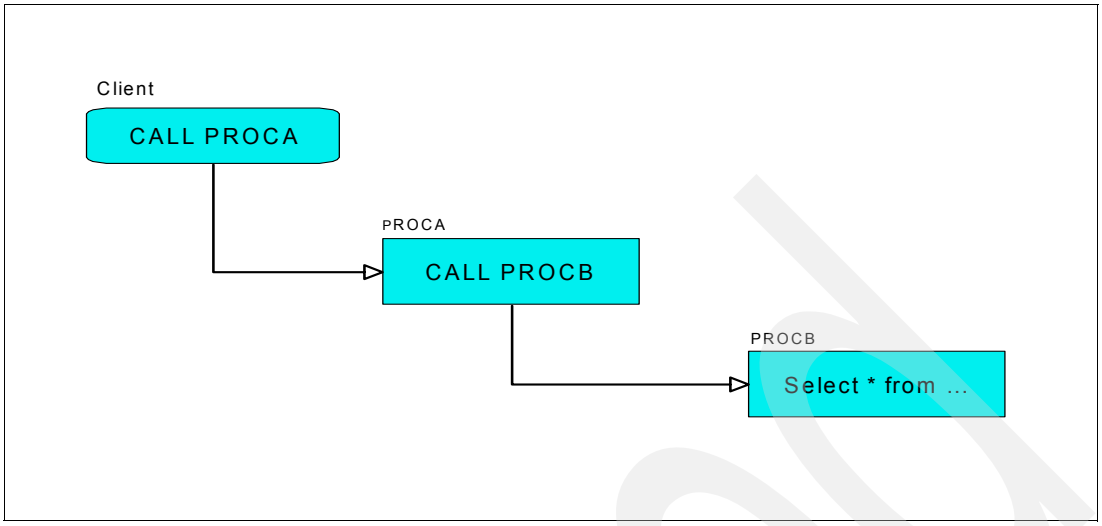


Figure 10-2 Nested stored procedures

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 DB2-established address space or 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 TCB to each WLM application environments.

Currently, there are no hard and fast rules for setting the NUMTCB parameter. In this redbook we provide some guidelines based on the experience of different customers. See 20.1.3, “NUMTCB” on page 322 for details.

In an informal test, we prepared a non-DB2 stored procedure and non-DB2 subprogram. When 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 normal practical life, the stored procedures contain some SQL statements, and when you compare the cost of total transaction with the cost of just CALL statement, the cost of CALL might be negligible.

For more information on instrumentation, refer to 20.2, “Managing server address spaces” on page 324.

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 or OS/390 are built with the COBOL language, the following sections provides 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 or OS/390), 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 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*, COBOL CALL statement can be used as an alternative. The COBOL CALL statement can be used to call other programs. The program containing CALL statement is the calling program where as the program identified in the CALL statement is subprogram. Called programs can contain CALL statements. For complete description and usage of COBOL subprograms, refer to the following COBOL manuals:

- *Enterprise COBOL for z/OS and OS/390 Programming Guide Version 3 Release 2*, SC27-1412-01

- *Enterprise COBOL for z/OS and OS/390 Language Reference Version 3 Release 2*, SC27-1408-01

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 also may 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 either to be used as a stored procedure or subprogram will be same. For stored procedures, additionally, DDL has to be created to define stored procedure to the DB2 subsystem. The difference consists in the way you invoke them; see Table 10-2. Stored procedures (irrespective of the type) can be invoked by SQL CALL statement and COBOL subprograms can be invoked by 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 definition.	Same or less compared to stored procedures. Additional parameters needed to compensate for DB2SQL and DB2INFO information.
Ability to call from anywhere	Possible. Generally should not be invoked by batch programs instead of local subroutines causing 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 procedure as well as on 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 requires 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 above table, the program preparation part of COBOL subprograms will be more or less the same as COBOL stored procedures. They exhibit different behavior during run-time 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 SQL call with COBOL call, which are described in 10.3.3, “Hybrid approach for optimization” on page 116.

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 where as with a static call resolves the subprogram name during linkedit 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 linkedited to the main program.	None of the subprograms need to be linkedited with main program.
Load module size	Big relatively as it contains all load modules of the subprograms.	No difference
Performance	Performs better as all load modules are loaded into memory in one pass	Slight performance overhead as at each call to subprogram requires the subprogram to be located and loaded into memory
Flexibility	Not flexible, as whenever a subprogram is compiled, the main program also need to be linkedited[a].	Flexible, as compilation of subprogram does not require main program to be linkedited.

a. If there is logic change between main program and subprogram, then both need to be compiled and linkedited.

As shown above, it is recommended to use dynamic calls as it allows flexibility and provides 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 elongated 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 stored procedure and other as subprogram. Our intention is to maintain one single program and invoke them 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 outer most program as a stored procedure, afterwards, the inner levels will be subprogram calls.

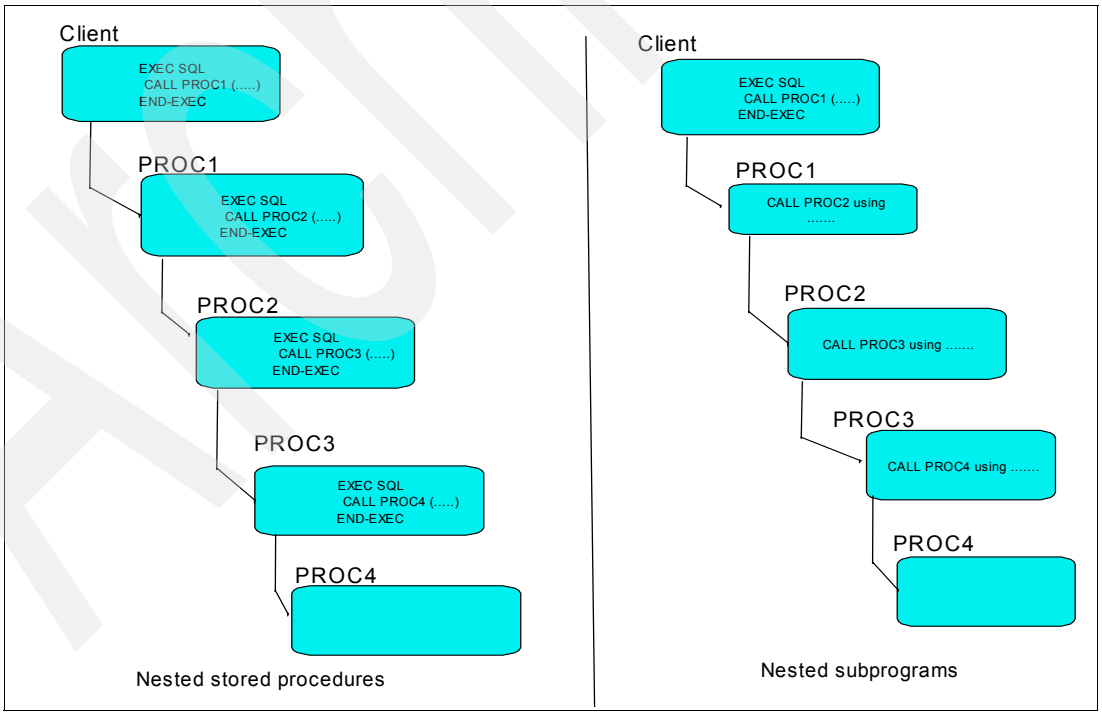


Figure 10-3 Nested stored procedure versus nested subprograms

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 subprogram. 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 subprogram through PROC2, DEF does not have to invoke PROC3 as 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. Pre-compile and compile the programs.
3. Link-edit with DSNRLI.
4. Bind the DBRMs to produce a DB2 packages.
5. Refresh WLM SPAS.
6. Define PROC1, PROC2, PROC3, and PROC4 as stored procedures.

Invocation

From application ABC, invoke PROC1 as shown below:

```
EXEC SQL
CALL PROC1 (..............)
END-EXEC.
```

From application DEF, invoke PROC3 as shown below:

```
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 on 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	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 Caller (with COBOL CALL): ► Declare a cursor similar to the one in subprogram and fetch from it

Note: As you see from Table 10-4, handling result sets with subprograms may be costlier as it involves the opening of two cursors on the same table for the same purpose, once in 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 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

Based on our experience, we recommend usage of *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 they are referenced within the program.
- DTTs are treated as dynamic SQLs, and hence the program involves in “incremental bind” every time the table is referred.

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 like it is CICS or distributed components. Traditionally, 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 the to requirement of different language interface modules. The COBOL-DB2 sub-program 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.

- Maintain multiple load modules for the same program, each linkedited differently using one of the above language interface modules.
- Maintain one load module and dynamically load the language interface module during run-time.

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 the step by step preparation of the program preparation and setting up of the run-time environment with respect to both solutions mentioned above.

Solution 1: Multiple load modules

Here we linkedit the program into multiple load modules, with 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, REU, NODYNAM, LIB`
3. Have three linkedit steps each with different interface modules as shown in Example 10-17. Ensure that the following linkedit options are specified:

```
RENT, REUS, AMODE(31), RMODE(ANY)
```

Example 10-17 Sample JCL to compile and linkedit

```
//*********************************************************************
//*        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 under go above process.

Run-time 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 run-time 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 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 run-time 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
       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
       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
       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
```

```
      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. Pre-compile 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, REU, DYNAM, LIB
```

3. Link-edit to create the load module. Only one linkedit is sufficient:

```
RENT, REUS, AMODE(31), RMODE(ANY)
```

Compile and linkedit the JCL as shown in Example 10-20.

Example 10-20 Sample JCL to compile and linkedit

```
//*********************************************************************
//*        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 packages.
5. Refresh WLM SPAS.
6. Define PROC1, PROC2, PROC3, and PROC4 as stored procedures.

Run-time 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 run-time 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 subprogram.

Special considerations

As 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 the *CICS Transaction Server for z/OS Version 2.3 CICS Application Programming Guide*, SC34-6231, 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 enhances 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 overhead of scheduling is more, 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 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 retrieves employee information for a specific employee number, and the second one 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 D, “Additional material” on page 651

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 run-time 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 run-time 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 redbook; we simply point to them here for convenience. They are:

1. WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 33 for details.
2. LE environment must be set up. See Chapter 5, “Language Environment setup” on page 41 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 75 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 55 for details.
6. Develop the calling application, if needed.
7. See Chapter 14, “Debugging” on page 173 for details on testing and debugging.

11.2 Passing parameters

Example 9-4 on page 89 shows our first sample C stored procedure. More examples are available as described in Appendix D, “Additional material” on page 651.

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 76. 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 139. In this section we use the parameter style GENERAL that does not permit NULL values.

Both parameter style DB2SQL 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 Parameters definition 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];
```

You have the choice of writing a C stored procedure either as a main program or as a subprogram. Unless your C stored procedure is very short and performance critical you should write it as a main program to benefit from the following tasks that Language Environment performs for you when running your stored procedure:

- Initialization and cleanup processing
- Allocating and freeing storage
- Closing all open files before exiting

All the stored procedures in this chapter are written as main programs. The stored procedure parameters are passed through *argv* and *argc*.

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 C/C++ Run-Time Library Reference,* SA22-7821. 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 11-3 shows includes and compiler defines.

Example 11-3 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 11-4 shows constants defines.

Example 11-4 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 11-5 shows the message defines.

Example 11-5 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 11-6 shows structures, enums, and types defined.

Example 11-6 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 run-time 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 11-7 shows the global variable declarations.

Example 11-7 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 11-8 shows the definitions for these functions.

Example 11-8 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 11-9 shows the SQLCA include and the DB2 host variable declarations.

Example 11-9 SQLCA include and DB2 host variables 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 run-time library function to do that, so we have to provide one as shown in Example 11-10.

Example 11-10 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 11-11.

Example 11-11 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 11-12 shows how to do this.

Example 11-12 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 11-13.

Example 11-13 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 11-14.

Example 11-14 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 11-15 shows the JCL used to prepare, compile, and linkedit 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 the *z/OS V1R4.0 Language Environment Programming Guide,* SA22-7561-04, for a detailed description of re-entrancy. Although CEESTART is the default main entry point for C applications, it is good practice to explicitly specify it in your linkedit SYSIN. You also must linkedit 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 11-15 JCL to compile EMPDTL1P

```
//EMPDTL1P 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 EMPDTL1P
//*          STORED PROCEDURE
//*-------------------------------------------------------
//STEP01   EXEC PROC=DSNHCPP,MEM=EMPDTL1P,
//         PARM.PC=('HOST(C)',CCSID(1047)),
//         PARM.COMP='/OPTFILE(DD:COPT)'
//PC.DBRMLIB   DD DSN=SG247083.DEVL.DBRM(EMPDTL1P),
//             DISP=SHR
//PC.SYSLIB    DD DSN=SG247083.PROD.SOURCE,
//             DISP=SHR
//PC.SYSIN     DD DSN=SG247083.PROD.SOURCE(EMPDTL1P),
//             DISP=SHR
//COMP.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)
   NAME EMPDTL1P(R)
/*
//*-------------------------------------------------------
//* 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(DB2G)
   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 like 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:

- 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 collection ID associated with the stored procedure package must be based on the following rules:
  - If you specify NO COLLID when creating the stored procedure, the package must use the same collection ID as the calling program.
  - If you specify COLLID *collection_id* when creating the stored procedure, the stored procedure must use this *collection_id*.

Also, see 9.1.9, “Collection ID stored procedure runs in” on page 84 for details on the definition of the collection ID.

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 may be a good choice.

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 11-16. Notice that the BEGIN/END DECLARE section for the calling application is different from the called application.

Example 11-16 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 if 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 11-17 list the new structures, notice the differences from Example 11-6 on page 131 because of the `isxxxNull` definitions.

Example 11-17 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 11-18 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 11-18 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 11-19.

Example 11-19 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 11-20 shows the new query_info definition.

Example 11-20 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 if any output parameters contain an unexpected NULL value as shown in Example 11-21, which also shows the modified DECLARE section, which is different from Example 11-16 on page 138.

Example 11-21 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 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.

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 75), 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 9.1.16, “Use of commit before returning” on page 87 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 as follows:

```
EXEC SQL ASSOCIATE LOCATOR (:rs_loc)
                        WITH PROCEDURE EMPRSETP;
```

6. Allocate a cursor for each result set to be processed as follows:

```
EXEC SQL ALLOCATE EMPRSETP_CSR CURSOR
             FOR RESULT SET :rs_loc;
```

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.

11.8 Handling result sets using Global Temporary Tables

If you need to pass results back that are not the result of a 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.

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 11-22 shows how to define a created temporary table to pass back a list of employees for a specific department.

Example 11-22 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 11-23 shows the function query_dept that queries all the employees from a department and inserts the rows into a created temporary table.

Example 11-23 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 11-24. 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 that requires your application to be structured as in the example.

Example 11-24 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 11-25.

Example 11-25 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 11-26. 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 11-26 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 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 Appendix D.1.3, “Sample C programs” on page 653.

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. Up to DB2 V7, you had to order a separate, no extra charge feature for the REXX interface. In DB2 V8, you do not have to order this feature; the REXX interface is included in every DB2 license.

We will 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 the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-74156 for details.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions can be found in Appendix D, “Additional material” on page 651.

Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 22 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 and we will simply list them here for convenience. They are:

1. WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 33 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 41 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 75 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 153 for details.

6. Grant the necessary privileges to the authorization ID of the use that executes the stored procedure. See Chapter 7, “Security and authorization” on page 55 for details.

7. Develop the calling application if needed.

Also, see Chapter 14, “Debugging” on page 173 for details on testing and debugging.

12.2 Passing parameters

In Example 9-5 on page 90 is our first sample REXX stored procedure. More examples are available as described in Appendix D, “Additional material” on page 651.

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 DB2GDER1
STAY RESIDENT NO
COLLID DSNREXX
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 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 DB2SQL 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 pre-compile 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

Package 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 COLLID value specified in the CREATE PROCEDURE statement. If NO COLLID is specified then DSNREXX package should in the collection ID of the caller.

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 EMPDTLSC( :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:

```
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 155.

If you want to add a statement about debugging a REXX stored procedure, you can either:

- Add the following REXX statement in my stored procedures

  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 75), 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 109.

Example 12-3 provides sample REXX code to process a result set.

Example 12-3 REXX code for result set processing

```
ADDRESS DSNREXX "EXECSQL ASSOCIATE LOCATOR (:LOC1) WITH PROCEDURE :PROC"
IF SQLCODE ¬= 0 THEN CALL SQLCA
SQLSTMT = "ALLOCATE C150 CURSOR FOR RESULT SET ?"
ADDRESS DSNREXX "EXECSQL ALLOCATE C150 CURSOR FOR RESULT SET :LOC1"
IF SQLCODE ¬= 0 THEN CALL SQLCA

DO UNTIL(SQLCODE ¬= 0)
  ADDRESS DSNREXX "EXECSQL FETCH C150 INTO :SEQNO, :TEXT"
  IF SQLCODE = 0 THEN
    DO
      SAY TEXT
    END
END
IF SQLCODE < 0 THEN CALL SQLCA

ADDRESS DSNREXX "EXECSQL CLOSE C150"
IF SQLCODE ¬= 0 THEN CALL SQLCA

RETURN
SQLCA:
TRACE O
```

```
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

SQL Procedures language

In this chapter we focus on the development of stored procedures using the SQL Procedures language. We will 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 “SQL Procedure Statements” in the *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426 for details.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions can be found in Appendix D, “Additional material” on page 651.

Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 22 to decide what components are applicable to your environment.

This chapter contains the following:

- Verify the environment
- Defining an SQL procedure
- Handling error conditions

13.1 Verify 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. WLM environment must be set up. See Chapter 4, “Setting up and managing Workload Manager” on page 33 for details.
2. LE environment must be set up. See Chapter 5, “Language Environment setup” on page 41 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 75 for details.
4. Develop the stored procedure. See 13.2.1, “Preparing and binding an SQL procedure” on page 159 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 55 for details.
6. Develop the calling application if needed.
7. Also see Chapter 14, “Debugging” on page 173 for details on testing and debugging.

13.1.1 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 75. 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;
```

While the parameters associated with an external stored procedure can be modified without modifying the program logic, ALTER PROCEDURE is different for an SQL procedure: it can

be used to change the options, not the parameters; if you want to modify the program logic, you need to drop and re-create the SQL procedure.

The other important difference is in 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 13.3, “Handling error conditions” on page 168 for error handling in SQL procedures as well as 14.1.5, “Unhandled SQL errors to CALL statements” on page 185 for error handling in external stored procedures.

13.2 Defining an SQL procedure

In this section we show the steps and the parameters used for defining an SQL language stored procedure.

13.2.1 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 13.1.1, “What is different about an SQL procedure?” on page 158. The relevant source code must therefore be extracted from the CREATE PROCEDURE statement.

When you use the client-based IBM DB2 Development Center to define the SQL procedure to DB2, the preparing and binding is automatically done for you by the build utility. If you do not use the Development Center, 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 pre-compiler or the SQL statement co-processor.
- Bind the resulting DBRM into a package.

13.2.2 Handling terminators defaults

You need to change statement terminators in DSNTIAD, DSNTEP2, and SPUFI when processing a CREATE statement for an SQL procedure, because the statement will probably have embedded semicolons in the procedure body, and DSNTIAD, DSNTEP2, SPUFI default to semicolon for their statement terminator symbol. If you are using the DSNHSQL proc to prepare your SQL procedure, and if you are using DSNTEP2 or DSNTIAD, or SPUFI to register the procedure in the SYSIBM.SYSROUTINES catalog table, you will need to make the following changes to change the SQL terminator character from the default (**;**) to some other character (such as %).

This is because the CREATE PROCEDURE statement must include the procedure body, which contains embedded semicolons at the end of each statement, and these programs will incorrectly interpret these embedded semicolons as statement terminators unless the changes are made:

- For DSNTEP2: Add PARMS('/SQLTERM(%)')
- For DSNTIAD: Add PARM('SQLTERM(%)')
- For SPUFI: Choose **Change defaults** in the SPUFI input panel, and change the SQL terminator to some special character other than **;**, such as %.

13.2.3 Handling comment lines

When you do not use the Development Center, some language elements that are valid in one SQL statement interface, may be invalid in an other. For instance the comment lines in SPUFI identified by two dash characters (--) are invalid for the SQL precompiler. Similarly, 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 13-1 should be eliminated in order to be able to submit the statement in SPUFI.

Example 13-1 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)
...
)
...
```

Note: Remember to use the allowed comment notation for your development environment.

13.2.4 Statements in an SQL procedures: Statements

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 13-2.

Example 13-2 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 13-3.

Example 13-3 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 13-4.

Example 13-4 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 13-5.

Example 13-5 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 13-6.

Example 13-6 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 13-7.

Example 13-7 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 13-8.

Example 13-8 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 13-9.

Example 13-9 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 13-10.

Example 13-10 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 13-11 shows an example.

Example 13-11 Compound statement

```
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
;
SELECT SQLCODE, SQLSTATE INTO PSQLCODE, PSQLSTATE FROM SYSIBM.SYSDUMMY1;
SET PSQLERRMC  = 'ADIOS';
END
```

Note: SQL statements in SQL procedures

A subset of the SQL statements that are described in Chapter 5 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426 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 Appendix A of the same manual.

The *additional statements available in DB2 V8* are discussed here.

GET DIAGNOSTICS

The GET DIAGNOSTICS statement obtains information about the execution status of the previous SQL statement that was executed. See Example 13-12.

Example 13-12 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 (LOOP, REPEAT, WHILE). See Example 13-13.

Example 13-13 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 13-14.

Example 13-14 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 13-15.

Example 13-15 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 13-16.

Example 13-16 RETURN statement

```
DECLARE ANY_ERRORS CHAR(1);
...
IF  ANY_ERRORS = 'Y' THEN RETURN 16;
                     ELSE RETURN 0;
END IF;
```

13.2.5 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, etc.

Note the following restrictions on SQL variables:

- SQL variable names can be up to 64 bytes in length and can include alphanumeric characters and the underscore character. In DB2 V8, SQL variable-names can be 128 bytes in new function mode.
- Although lower case characters are allowed in variable names, DB2 converts all variable names to uppercase, and two SQL variables such as myvar and MYVAR that are identical except for the case, are not allowed.
- 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.
- You cannot use an SQL reserved word as an SQL variable name, even when it is delimited.
- 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.
- You can use an SQL procedure parameter with a LOB data type, but you cannot declare an SQL variable with a LOB data type.

Variable names are implicitly or explicitly qualified and we suggest the following guidelines:

- When you use an SQL procedure parameter in the procedure body, qualify it with the procedure name as shown in Example 13-17.

Example 13-17 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 13-18.

Example 13-18 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 13-19.

Example 13-19 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;
```

13.2.6 Passing parameters

Example 9-7 on page 90 shows our first sample SQL language stored procedure. More examples are available as described in Appendix D, “Additional material” on page 651.

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 13-20.

Example 13-20 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.

13.2.7 Actions for the calling application

The calling application must initialize all passed parameters and call the stored procedure. The call is shown in Example 13-21.

Example 13-21 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, and this is discussed in 13.2.9, “Handling result sets” on page 167.

13.2.8 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 and this is discussed in 13.2.9, “Handling result sets” on page 167.

13.2.9 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 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. We will discuss each of these alternatives in detail below.

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 75), 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. 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 109.

13.2.10 Re-deploying SQL procedures

DB2 builds SQL procedures as external C load modules, but this resulting load module 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 re-deployment of the SQL procedure. In DB2 Version 7, this causes unpredictable results. In Version 8 the invocation generally fails.

You can re-deploy 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.

13.3 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 statements called handlers to trap the error or warning conditions. We discuss this in 13.3.1, “Using handlers in an SQL procedure” on page 168.
- You can return a condition code to the calling application. We discuss this in 13.3.2, “Using the RETURN statement for the SQL procedure status” on page 169.
- You can use SIGNAL or RESIGNAL to raise a specific SQLSTATE and a text message. We discuss this in 13.3.3, “Using SIGNAL and RESIGNAL to raise a condition” on page 170.
- 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 13.3.4, “Forcing errors in an SQL procedure when called by a trigger” on page 170.

13.3.1 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. 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.

Example of a CONTINUE handler

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';
```

Example of an EXIT handler

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 (available in DB2 V8) provides information about the execution status of a statement. See *DB2 Universal Database for z/OS SQL Reference*, SES1-2306-02 for a list of all variables that are returned by the GET DIAGNOSTICS statement. Example 13-22 shows how the MESSAGE_TEXT can be used.

Example 13-22 MESSAGE_TEXT statement

```
DECLARE EXIT     HANDLER FOR SQLEXCEPTION
GET DIAGNOSTICS CONDITION
    1 PSQLERRMC = MESSAGE_TEXT
    , PSQLCODE  = DB2_RETURNED_SQLCODE
    , PSQLSTATE = RETURNED_SQLSTATE;
```

13.3.2 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

If you do not include a RETURN statement in an SQL procedure, by default, DB2 sets the return status to 0 for an SQLCODE that is 0 or positive and sets it to -1 for a negative SQLCODE.

13.3.3 Using SIGNAL and RESIGNAL to raise a condition

You can use the SIGNAL statement anywhere in a 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 13-23 shows.

Example 13-23 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 13-24.

Example 13-24 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

13.3.4 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 statement are accepted at CREATE PROCEDURE, but at run time they violate the restriction that COMMIT and ROLLBACK statements are not allowed in procedures invoked from a trigger and generate the error as shown in Figure 13-25.

Example 13-25 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 specially 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 27, “Using triggers and UDFs” on page 451 for details on triggers invoking a stored procedure.

14

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 DB2SQL parameter style 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 (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

14.1 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 UDB for z/OS Version 8 Messages and Codes,* GC18-7422 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 LUW across multiple environments.
- **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.

14.1.1 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 14-1 represents the type of errors that can be uncovered.

Table 14-1 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 were 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.

14.1.2 Connectivity SQL errors

The errors listed in Table 14-2 reflect various types of connection problems from Logical Unit of Work issues to security violations for remote processing.

Table 14-2 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.

SQLCODE	REASON	RESPONSE
-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 rollback 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. ► See *DB2 UDB for z/OS Version 8 Messages and Codes*, GC18-7422 for reason code explanations.	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.

14.1.3 CALL statement error SQLCODEs

Table 14-3 contains information relating to the errors that can occur on the CALL statement.

Table 14-3 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 14.2, “Debugging options” on page 190

SQLCODE	REASON	RESPONSE
-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 DB2SQL 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 Part 4 of *DB2 UDB for z/OS Version 8 Messages and Codes*, GC18-7422 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 `WLM DISPLAY,APPLENV=wlmenv` command to verify the status of the application environment. Then the MVS `WLM VARY APPLENV=wlmenv,RESUME` command 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.

SQLCODE	REASON	RESPONSE
-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.
-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.

SQLCODE	REASON	RESPONSE
-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.

14.1.4 Invoking program, non-CALL SQL errors

You have executed the CALL statement, the stored procedure returns to your invoking problem with an SQLCODE of zero. What can go wrong? Table 14-4 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 14-4 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 a 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	THE CURSOR NAME cursor-name IS NOT DEFINED Cursor cursor-name was referenced in an SQL statement, and one of the following is true: ► Cursor 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 was not allocated (using the ALLOCATE CURSOR statement) in the application program before it was referenced. ► Cursor was referenced in a positioned UPDATE or DELETE statement, which is not a supported operation for an allocated cursor. ► Cursor was allocated, but was closed before this cursor reference. ► Cursor was allocated, but a ROLLBACK operation occurred before this cursor reference. ► Cursor was allocated, but its associated stored procedure was called again since the cursor was allocated, new result sets were returned, and cursor was deallocated.	Check the application program 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 before SQL statements that reference the cursor. If the cursor-name was <UNKNOWN>, then the cursor was not successfully declared or allocated. This 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: ► 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.

14.1.5 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 DB2SQL

One technique to simplify the CALL statement parameters to be shared for describing errors encountered is the use of PARAMETER STYLE DB2SQL. 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 14-5 describes the additional parameters.

Table 14-5 DB2SQL additional parameters

DB2SQL 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(27) value.
SPECIFIC PROCNAME	The specific name of the stored procedure. The specific name is a VARCHAR(18) 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 14-5, *must* be referenced in the linkage definition area of the stored procedure source program. The DB2SQL additional parameters are *not* defined or referenced in the invoking program or in the CREATE PROCEDURE. 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.

```
COBOL:
LINKAGE SECTION.
01 PARM1 ....
01 PARM2 ,,,,
01 PARM3 ...
01 IV-PARM1 ...
01 IV-PARM2 ...
01 IV-PARM3 ,,,
01 DB2SQL-SQLSTATE                              PIC    X(05).
01 DB2SQL-QUAL-PROCNAME.
     49 DB2SQL-QUAL-PROCNAME-LEN       PIC    S9(04)   COMP.
     49 DB2SQL-QUAL-PROCNAME-TEXT     PIC    X(27).
01 DB2SQL-SPEC-PROCNAME.
     49 DB2SQL-SPEC-PROCNAME-LEN       PIC    S9(04)   COMP.
     49 DB2SQL-SPEC-PROCNAME-TEXT     PIC    X(18).
01 DB2SQL-DIAGNOSTICS.
     49 DB2SQL-DIAGNOSTICS-LEN             PIC    S9(04)   COMP.
     49 DB2SQL-DIAGNOSTICS-TEXT           PIC    X(70).

PROCEDURE DIVISION USING PARM1, PARM2, PARM3,
                                IV-PARM1, IV-PARM2, IV-PARM3,
                 DB2SQL-STATE, DB2SQL-QUAL-PROCNAME,
                 DB2SQL-SPEC-PROCNAME, DB2SQL-DIAGNOSTICS.
```

For examples in other languages refer to *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

If the stored procedure program sets the DB2SQL-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 DB2SQL" on page 100.

14.1.6 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 cannt 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 UDB for z/OS Version 8 Messages and Codes,* GC18-7422, 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.

Application conditions:

– The CURRENT PACKAGESET special register was not set correctly by the application.
– The application was not connected to the proper location.

- 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.

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 UDB for z/OS Version 8 Messages and Codes,* GC18-7422, all the operations under reason 02 and 03 might fix the problem.

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';
```

14.2 Debugging options

In this chapter, we will 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/1 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 463.

14.3 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.

14.3.1 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 14-1 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 14-1 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 14-1 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 14-1 Console messages for abend that resulted in SQLCODE -430

14.3.2 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 run-time option to specify a ddname for the LE debugging messages. Example 14-2 highlights the MSGFILE and WLM name that will be used throughout this discussion. For details on the RUN options, see 5.2, “Language Environment run-time options” on page 42.

Example 14-2 Sample CREATE with LE run-time 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 and OS/390 Programming Guide Version 3 Release 2*, SC27-1412.
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 14-3 accesses the Job Data Set panel by coding ? in the NP field.

Example 14-3 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 14-4 shows the Job Data Set Display (JDSD) accessed, and the selection of the SYSDBOUT data set referenced in the run-time options of Example 14-2.

Example 14-4 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 14-5, the stored procedure PRGTYPE1 terminated with a S0C7 at statement 331.

Example 14-5 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 14-6 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 14-6 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 192, locate the compile SYSPRINT for the stored procedure. As shown in Example 14-7, you now have the failing statement in the stored procedure source code.

Example 14-7 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
.
.
```

14.3.3 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 14-8 highlights the Linkage and the PROCEDURE DIVISION USING clause. This also confirms that PCALL-CTR is a parameter.

Example 14-8 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 14-6 on page 193 was indicative of an uninitialized value.

PCALL-CTR was defined as an output parameter according to the CREATE statement in Example 14-2 on page 191.

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.

14.3.4 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.

14.4 Compiler and LE options for debugging

We describe the COBOL Compiler and LE options available for debugging.

14.4.1 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 run-time debugging:

```
LIST, MAP, OFFSET, TEST, VBREF, XREF
```

Refer to *Enterprise COBOL for z/OS and OS/390 Programming Guide Version 3 Release 2*, SC27-1412 for details.

14.4.2 Language Environment run-time options

There are several run-time options that affect debugging in Language Environment (LE). The TEST run-time 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 run-time 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.

14.5 IBM Debug Tool

Debug Tool combines the richness of the z/OS and OS/390 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.

14.5.1 IBM Debug Tool overview

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 or OS/390 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 14-6 and Table 14-7 map out the combinations of compiler and subsystems that are supported.

An updated list of supported compilers and environments is available on the Debug Tool Web site at:

http://www.ibm.com/software/awdtools/debugtool

Table 14-6 Debug Tool interface type by compiler or Assembler

COMPILER or Assembler	Batch mode	Full screen mode	Remote mode
VS COBOL II V1R3 and V1R4 (with limitations)	Y	Y	
AD/Cycle® COBOL/370™ V1R1 V1R1	Y	Y	
COBOL for MVS and VM	Y	Y	Y
COBOL for OS/390 and VM	Y	Y	Y
Enterprise COBOL for z/OS and OS/390	Y	Y	Y
OS PL/I 2.1, 2.2, and 2.3 (with limitations)	Y	Y	
PL/I for MVS and VM	Y	Y	
Enterprise PL/I	Y	Y	Y
AD/Cycle C/370™ V1R2	Y	Y	
C/C++ for MVS/ESA Version 3 Release 2	Y	Y	
OS/390 C/C++ feature version 1.3 and earlier	Y	Y	
OS/390 C/C++ feature version 2.4 and later	Y	Y	Y
z/OS C/C++ feature	Y	Y	Y
IBM High Level Assembler (HLASM)	Y	Y	Y

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 three windows:

- A Source window in which to view your program source or listing
- A Log window, which records commands and other interactions between Debug Tool and your program

- A Monitor window in which to monitor changes in your program

You can debug all languages supported by Debug Tool in full-screen mode.

You can debug non-TSO programs in full-screen mode by using the full-screen mode through a VTAM terminal facility. For example, you can debug a COBOL batch job running in MVS/JES, a DB2 stored procedure, an IMS transaction running on a IMS MPP region, or an application running in UNIX System Services. Contact your system administrator to determine if the full-screen mode through a VTAM terminal facility is active on your system.

Our example Debug Tool session,14.5.2, “IBM Debug Tool on z/OS, VTAM MFI example” on page 198, uses the VTAM terminal facility also known as Main Frame interface (MFI) VTAM.

Table 14-7 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 provided by some compiler products and WebSphere Studio Enterprise Developer, provides users with the ability to debug host programs, including batch, through a graphical user interface (GUI) on the workstation.

The remote debuggers VisualAge®, Remote Debugger, and IBM Distributed Debugger are available through products such as:

- C/C++ Productivity Tools for OS/390
- VisualAge COBOL for Windows 3.0
- VisualAge for Java, Enterprise Edition for OS/390
- VisualAge PL/I for Windows

The compiled language debugger is the remote debugger available through WebSphere Studio Enterprise Developer.

Remote debug mode also provides important additional functions such as the ability to interactively debug batch processes. 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 equipped with a remote debugger. You can debug the following applications:

- VisualAge PL/I for OS/390 applications
- Enterprise PL/I for z/OS and OS/390 applications
- Applications running in UNIX System Services Shell
- Enterprise COBOL for z/OS and OS/390
- COBOL for MVS and VM applications
- COBOL for OS/390 and VM applications

Table 14-8 shows the operating system and communication protocol for each remote debugger.

Table 14-8 Remote debugger by operating system and communication protocol

Operating system and communication protocol	VisualAge COBOL for Windows	z/OS or OS/390 C/C++	VisualAge for Java, Enterprise Edition	WebSphere Studio Enterprise Developer
OS/2 4.0 and TCP/IP			Y	
Windows NT 4.0 and TCP/IP	Y	Y	Y	Y
Windows 2000 and TCP/IP	Y		Y	Y
Windows XP and TCP/IP	Y			Y
Windows 95 and TCP/IP			Y	

14.5.2 IBM Debug Tool on z/OS, VTAM MFI 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 run-time loadlib. Therefore, the default debugging tool on z/OS is the IBM Debug Tool (DT). We used Debug Tool V3.1.

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, DT gives you the capability of testing programs in batch, using a non programmable terminal (VTAM MFI) 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 DT, we will discuss the benefits, followed by a description of the DT setup for the MFI VTAM mode, which is available on z/OS. Next, we will 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 and OS/390 User's Guide, SC18-7171* for details of using the Debug Tool in this and other modes.

Benefits of VTAM MFI

Main Frame interface (MFI) is the preferred user interface that allows you to control the debugging session in interactive full screen mode. This interface has VTAM, TSO, and CICS modes. Using the VTAM MFI interface gives you great flexibility since your program becomes a VTAM application without tying up your TSO/ISPF session.

While you are interactively debugging your procedure, you are free to use one session for other activities including:

- Submitting your JCL to launch the application, which calls the stored procedure you are debugging
- Validating the DDL that defined the procedure
- Submitting other jobs and maintaining your regular productive activities

If you have the task of debugging procedures from other environments, you are not locked into a single TSO task for your debugging activities.

Setup for the VTAM MFI mode

In this example, we demonstrate how to utilize the Debug Tool in VTAM MFI mode on a 3270 emulator. In this example we used IBM Personal Communications (PCOMM). You can use your emulator of choice:

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 14-2 displays the customize communication panel of PCOMM to be modified to define the host.
   - On the panel where you define your host connection, put in your host domain name, ours was `WTSC63.ITSO.IBM.COM`, and the port number that your systems programmer sets up for you in your TCP/IP, SYSTCPD data set. Our port was 1023 and was specified in TCP.SC63.TCPPARMS(TCPPROF). This port has to be an unused port, and defining it here now reserves it for use by Telnet.

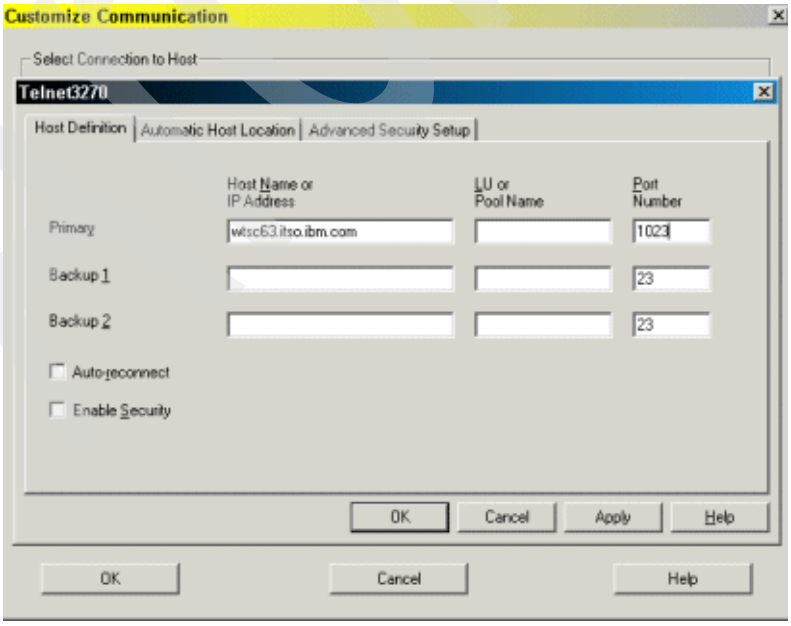


Figure 14-2 Defining the z/OS connection

– Save this Second session as a Workstation Profile, and give it a meaningful name such as MFIVTAM Debug Tool as shown in Figure 14-3.

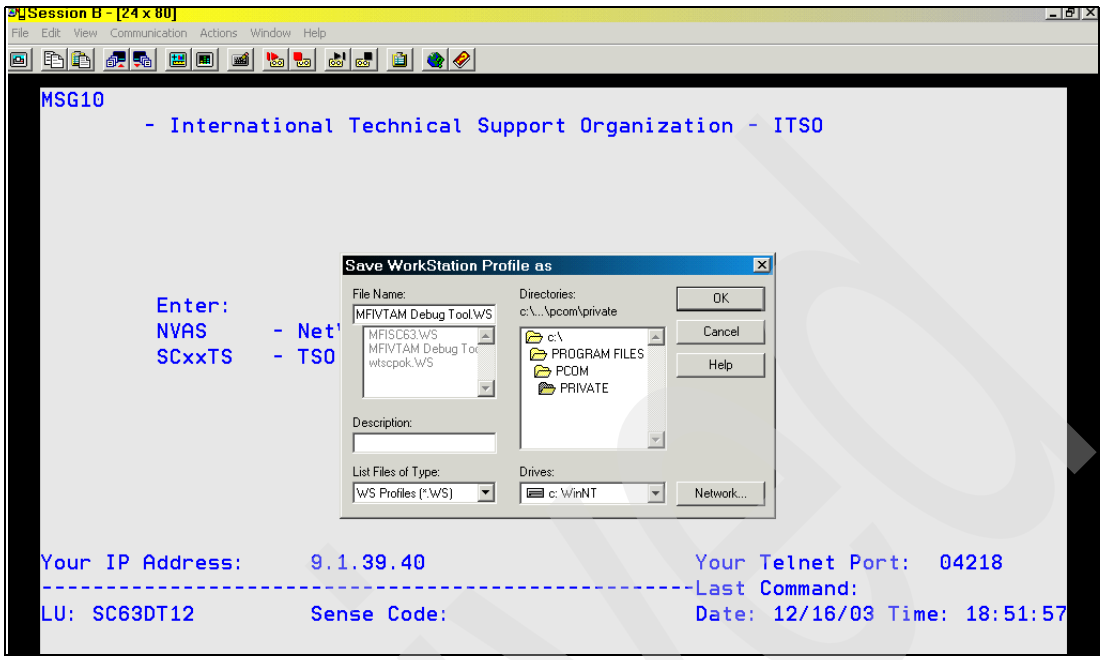


Figure 14-3 Saving second session profile

– Start the session. Locate the LU name in the lower left hand corner. See Figure 14-4 for a highlighted view of the LU name.

– For this session, our setting was LU: SC63DT12

```
MSG10
       - International Technical Support Organization - ITSO



        Enter:
        NVAS     - NetView Access Services
        SCxxTS   - TSO on SCxx (fill in the "xx")




Your IP Address:     9.1.39.40                  Your Telnet Port:  04218
-------------------------------------------------Last Command:
LU: SC63DT12         Sense Code:                Date: 12/16/03 Time: 18:51:57
```

Figure 14-4 The LU name

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(,,,MFI%SC63DT12:*)'

2. Using your first session, re-compile 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.

Demonstration of Debug Tool with VTAM MFI

Figure 14-5 shows the initial screen of our sample debugging program PRGTYPE1, 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

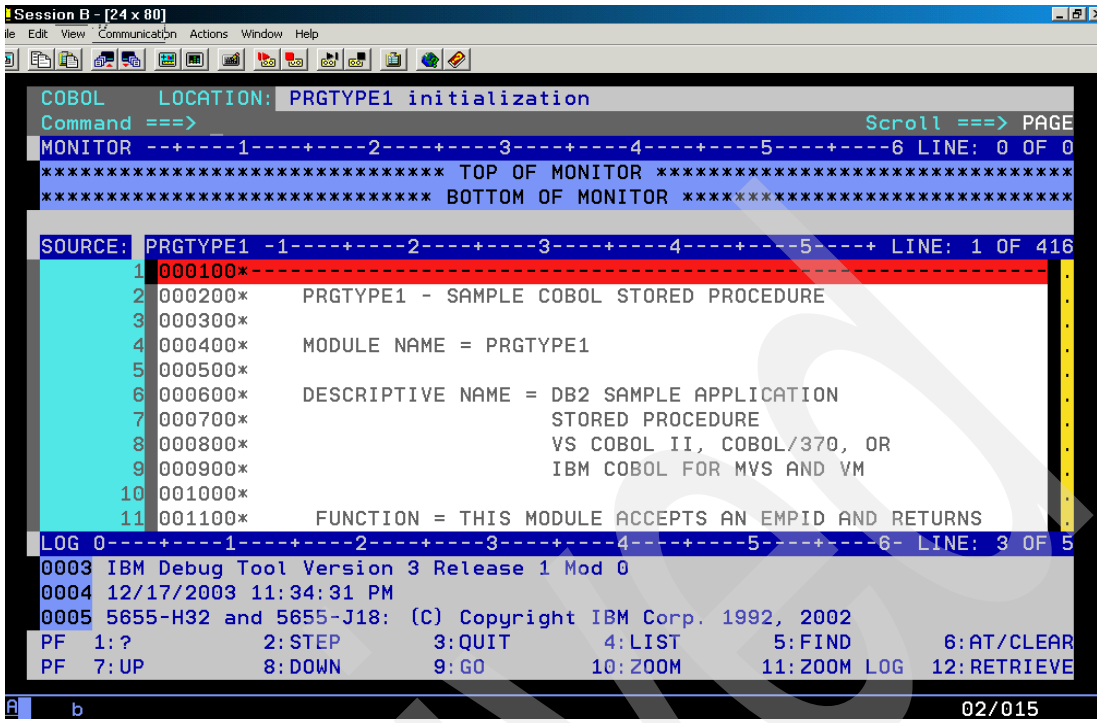


Figure 14-5 Initialization of DT stored procedure, PRGTYPE1

To start the stored procedure, press PF9, **GO**. Figure 14-6 shows the results of the **GO** command. DT stopped at statement 331.1 (statement 331, command 1) and reported that a Data Exception has occurred. The failing instruction is highlighted in red.

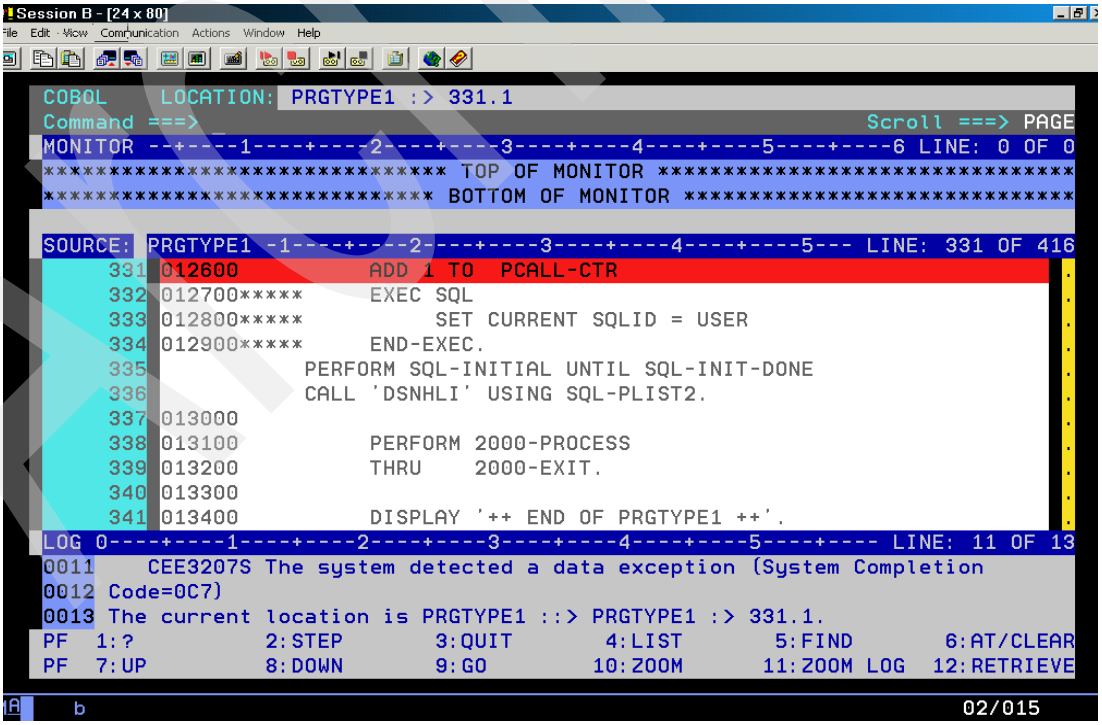


Figure 14-6 DT displays data exception error

We issued a LIST command to see the contents of PCALL-CTR. Figure 14-7 shows us that because the value is unprintable, we need to list the hex contents.

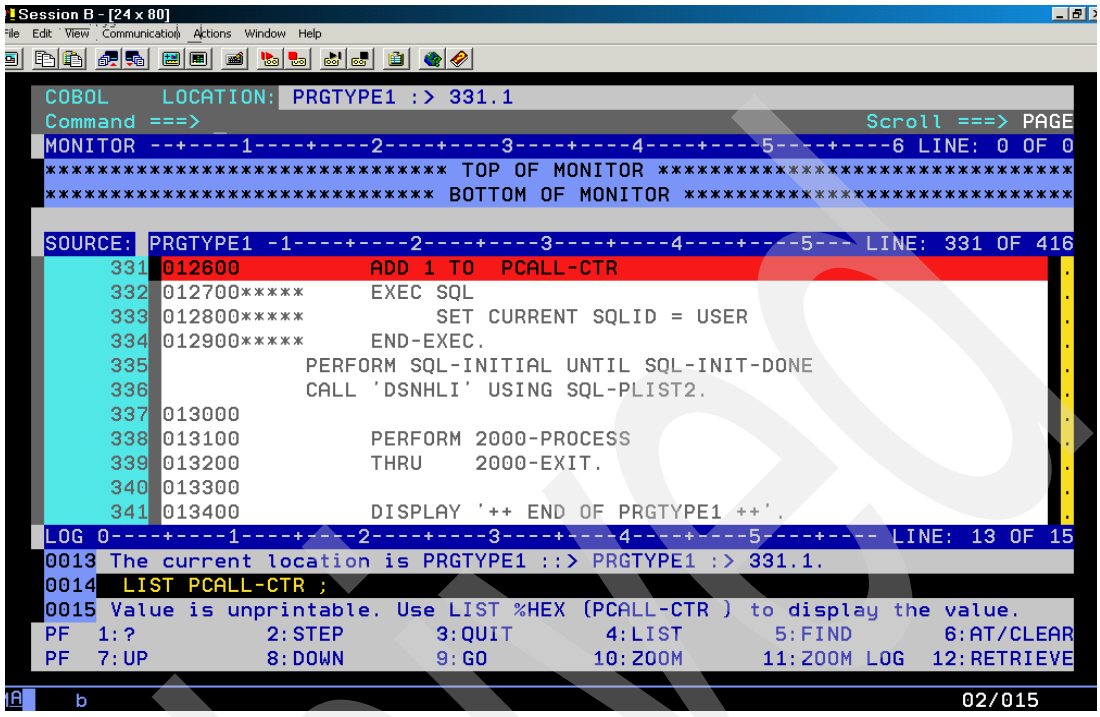


Figure 14-7 DT LIST command display

Figure 14-8 shows the results of the hex list. We see that PCALL-CTR contains invalid data.

Notice that with Debug Tool V4, the LIST command outputs the hex contents without the use of the **LIST %HEX** command.

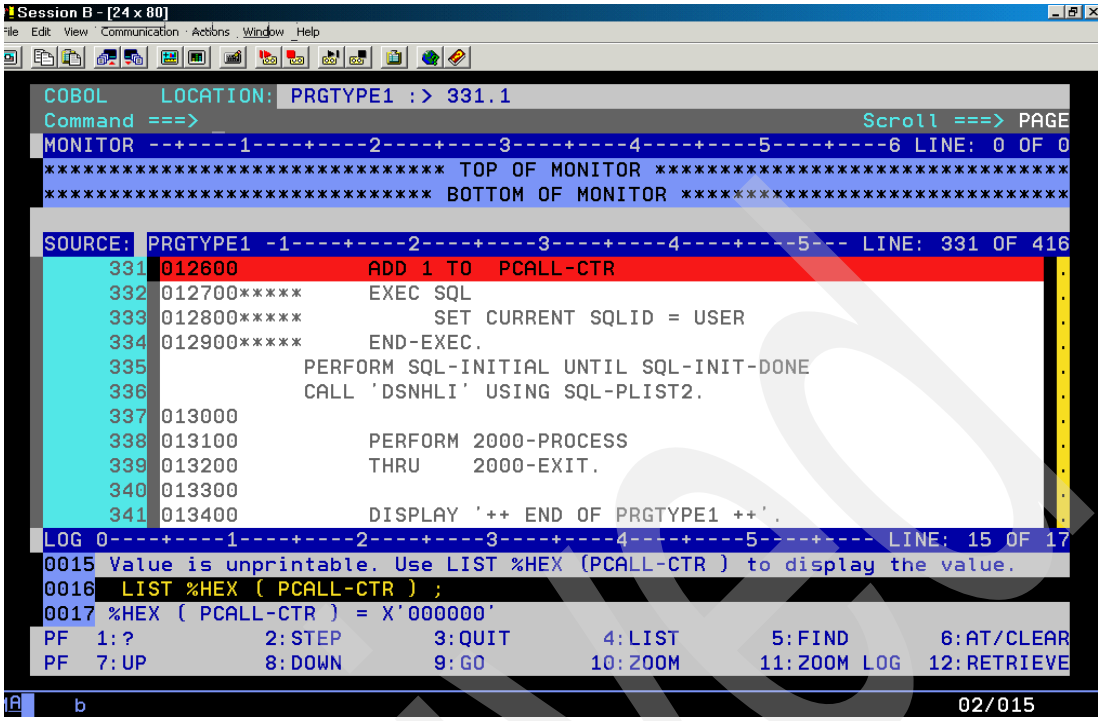


Figure 14-8 LIST %HEX

Figure 14-9 shows the entering of a **FIND** command. The **FIND** does not appear in the Log window. To repeat a find, press PF5.

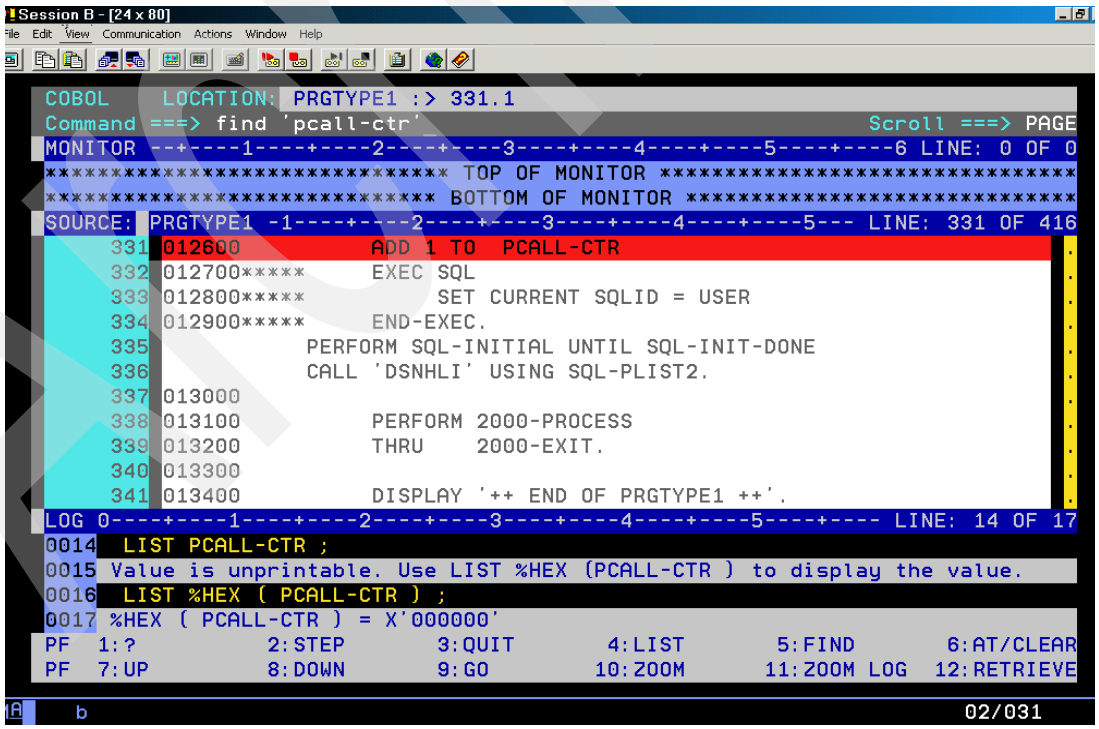


Figure 14-9 FIND command

As shown in Figure 14-10, we used PF5 to repeat the previous find until we reached the definition of the PCALL-CTR field.

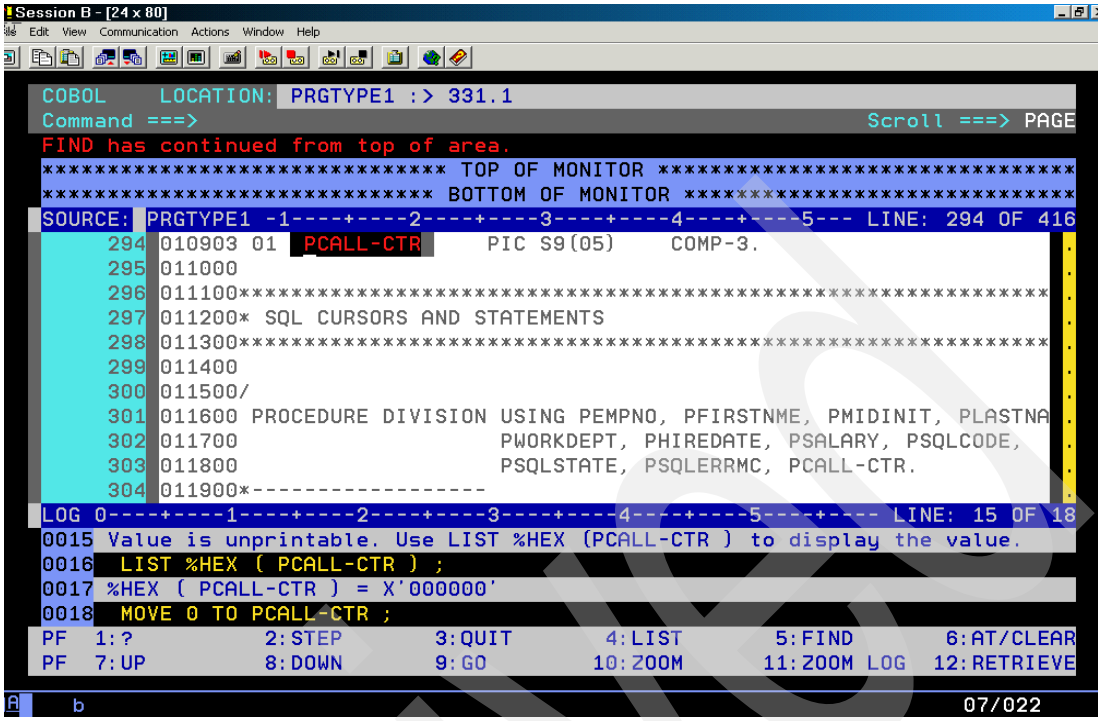


Figure 14-10 Results of repeated PF5 to locate definition

After quitting (F3) and restarting (submit JCL from first session TSO/ISPF screen), we use the AT command to set breakpoints that will cause execution to pause. Figure 14-11 shows that the AT command can be coded on the command line (at 330...) or as a line command next to a statement (statement 335) where we would like to set a breakpoint.

When the statements are reached, DT gets control and executes DT commands either coded with the AT or entered on the command line. Figure 14-11 demonstrates that we are able to initialize PCALL-CTR with zeroes before statement 331.1 and press PF 9 to begin execution, or PF 2 to step through the program, without an abend.

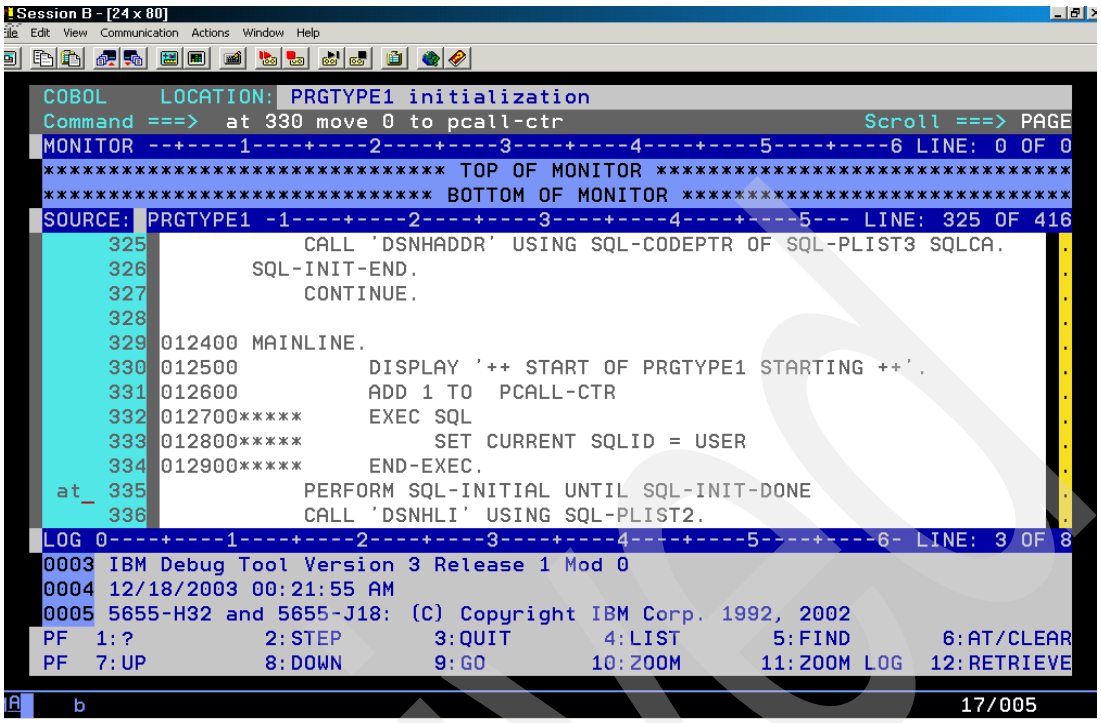


Figure 14-11 Line command AT

In Figure 14-12, the grey shading represents executed ATs, and the log shows the listing of the new PCALL-CTR value of +00001.

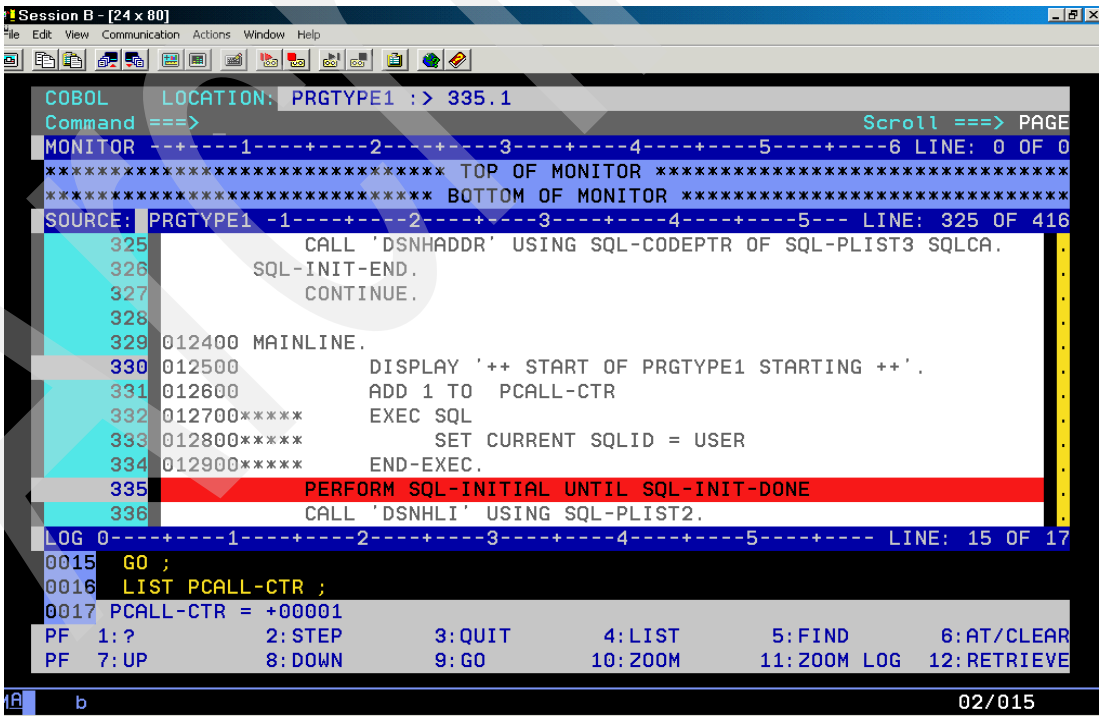


Figure 14-12 AT and LIST after initializing PCALL-CTR

Continuing with the finding of PCALL-CTR (Figure 14-13) allows us to see that PCALL-CTR is a parameter from the invoking program. We determined in Figure 14-8 that PCALL-CTR

was not initialized when we attempted to increment it in statement 331. The next step is to analyze if PCALL-CTR is defined as an input or output parameter. According to the CREATE statement in Example 14-2, it is an output parameter. As indicated in14.3, “Classical debugging of stored procedures” on page 190, DB2 did not move the output parameter’s data to the work area at the time of the call.

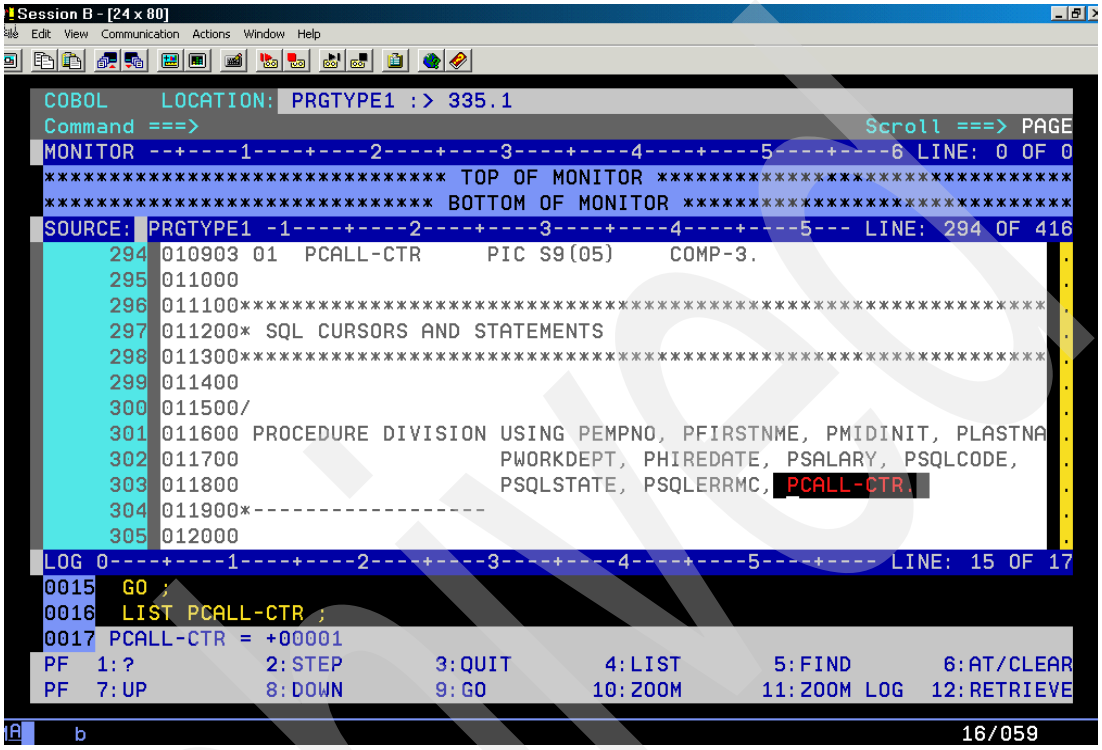


Figure 14-13 FIND again

Obviously, you have not seen everything that is available for debugging with the Debug Tool. We barely scratched the surface. For more information to help you get started with this debugging option, see the documentation available from the Debug Tool Web site:

http://www.ibm.com/software/awdtools/debugtool/

We found useful the following references to standard manuals:

- *Debug Tool for z/OS User’s Guide,* SC18-9012:
  - Chapter 10, “Linking DB2 programs for debugging” page 42
  - Chapter 15, “Example: TEST run-time options” pages 66 to 67
  - Chapter 19, “Starting a debugging session in full-screen mode through a VTAM terminal” page 85, also next section
  - Chapter 20, “Requesting an attention” page 119
  - Chapter 33, “Debugging DB2 programs in full-screen mode” page 243 to 244
  - Chapter 34, “Debugging DB2 stored procedures” page 245
- *Debug Tool for z/OS Customization Guide*, SC18-9008
  - Chapter 5
- *Debug Tool for z/OS Reference and Messages*, SC18-901
  - Chapter 1, “Syntax of the TEST run-time option” pages 1 to 6

14.6 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 14-14.

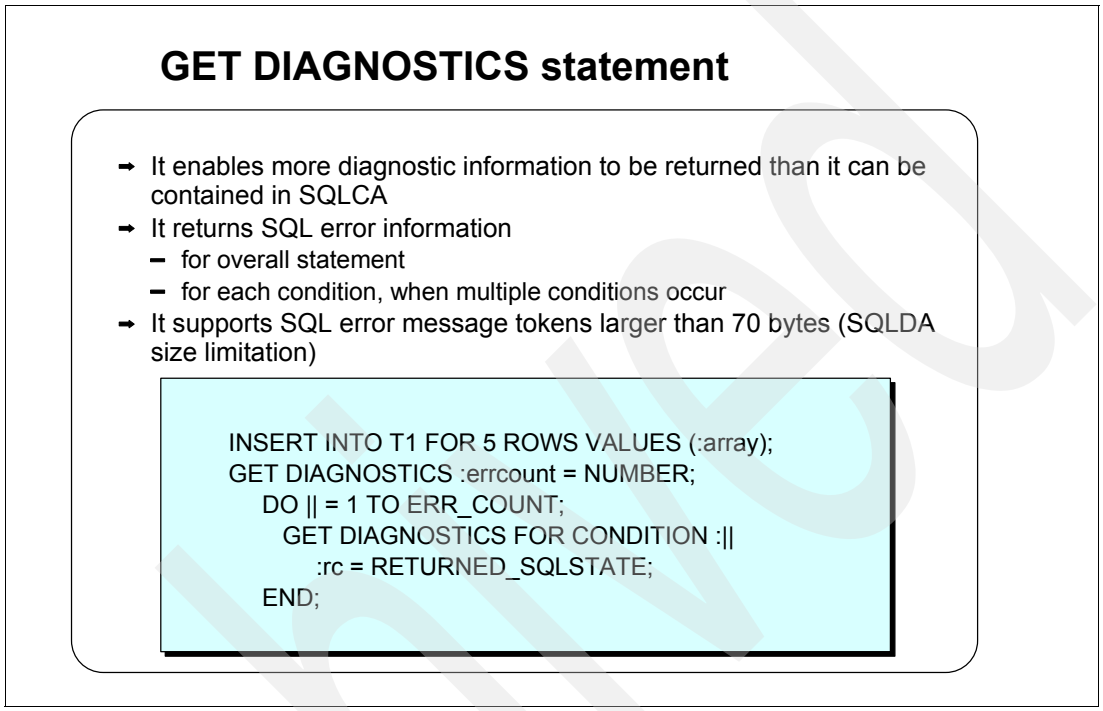


Figure 14-14 GET DIAGNOSTICS statement

In Figure 14-15 we show the syntax for the GET DIAGNOSTICS statement.

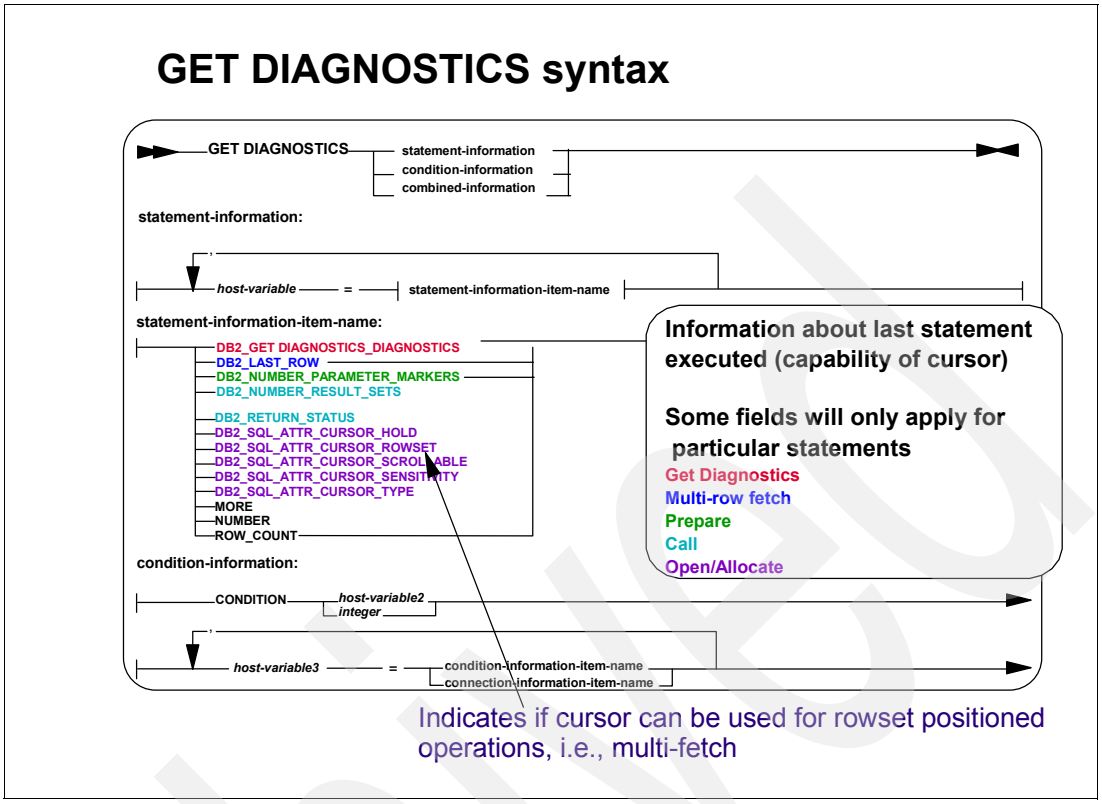


Figure 14-15 GET DIAGNOSTICS syntax

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 the following examples, where we attempt to fetch 10 rows with a single FETCH statement.

- Example 1:

  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 INTEGERS and :hva2 an array of DECIMAL(15,0) values. The data values for :hva1 and :hva2 are represented in Table 14-9.

Table 14-9 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

15

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

15.1 Remote stored procedures

A program which calls a stored procedure can be called as a client or requester. As shown in Figure 15-1 on page 214, if the client and the stored procedure both execute within the same DB2 subsystem, then the stored procedure is called a *local stored procedure*.

If the client executes in one DB2 subsystem (DB2L) and the stored procedure executes in another DB2 subsystem (DB2R), then the stored procedure is a *remote stored procedure*.

Stored procedures, being reusable programmed components, can be called both from local applications and remote applications. The subsystem where the client program executes can be referred as local subsystem and the subsystem where the stored procedure executes can be referred as remote subsystem. The remote DB2 subsystem can be on the same machine or thousands of miles away on a different machine. The remote DB2 subsystem is known to the local DB2 subsystem by its location name, which is recorded in the communications database (part of DB2 catalog).

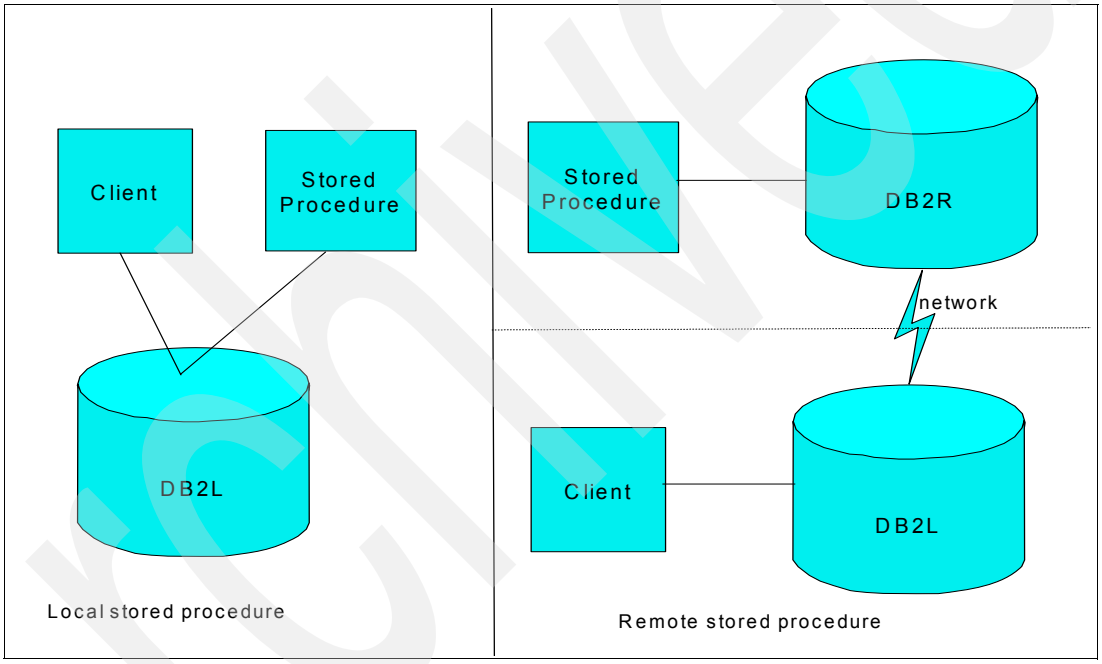


Figure 15-1 Local stored procedure vs. remote stored procedure

Somewhere along in its code, the client program logic decides to CALL a stored procedure. If it is a remote stored procedure, connectivity to the remote subsystem is either established with an explicit *CONNECT TO hostname* statement, or the stored procedure name is a *qualified three part name* that the local DB2 for z/OS can translate in a DB2 location. Either way, the information for the DRDA connectivity is found in the communication database.

Communications database

The communications database (CDB) is part of the DB2 catalog. The CDB consist of a set of tables that establish conversations with remote DBMS. 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 methods, DRDA or DB2 private protocol. However, invocation of stored procedure is supported only with DRDA.

The CDB consists of the following tables:

- SYSIBM.IPLIST (new table in V8)
- SYSIBM.IPNAMES
- SYSIBM.LOCATIONS
- SYSIBM.LULIST
- SYSIBM.LUMODES
- SYSIBM.LUNAMES
- SYSIBM.MODESELECT

In order to communicate with remote DB2 subsystems, the CDB has to be populated. If a DB2 subsystem is intended to act only as a server, then you do not need to populate the tables. The tables listed above have to be populated at the requestor's subsystem with details about the server subsystem. You do not have to populate all the tables. For more information on populating the tables, and in general how to establish DRDA connectivity, refer to the following manuals:

- *Distributed Functions of DB2 for z/OS and OS/390,* SG24-6952
- *DB2 UDB for z/OS Version 8 Installation Guide,* GC18-7418
- *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426

CONNECT statement

The CONNECT statement connects the application process to a designated server. There are two types of CONNECT statements both with same syntax but with different semantic. Figure 15-1 shows main differences between Type 1 and Type 2 CONNECT statements.

Table 15-1 Main differences between type 1 and type 2 CONNECT

CONNECT (Type 1)	**CONNECT (Type 2)**
CONNECT statements can be executed only when the application process is in the connectable state. 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. There are no rules about the connectable state.
If a CONNECT statement fails because the application process is not in the connectable state, the SQL connection status of the application process is unchanged. If a CONNECT statement fails for any other reason, the application process is placed in the unconnected state.	If a CONNECT statement fails, the current SQL connection is unchanged and any subsequent SQL statements are executed by that server, unless the failure prevents the execution of SQL statements by that server.
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.

CONNECT (Type 1)	CONNECT (Type 2)
A CONNECT to the current server is executed like any other CONNECT (Type 1) statement.	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 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.

Type 1 or Type 2 connection will be determined based on precompiler option. Type 2 connection is the default option and recommended by IBM. The connect rules apply to application process are determined by the first CONNECT statement that is executed (successfully or unsuccessfully). Programs containing 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 to authorized to connect to the identified server or local DB2.

15.2 Remote stored procedure preparation

There is no difference in the stored procedure preparation whether to be used for local applications or for remote applications. The following steps happen at server side:

1. Develop a program to be used as stored procedure, compile, linkedit.
2. Define the stored procedure.
3. Bind the DBRM as a package. The stored procedure packages does not have to be bound to a plan. They use a caller's plan. This is same for all stored procedures, local or remote.
4. Grant an execute privilege to the invoker of the stored procedure.

15.2.1 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. Make an unqualified call to a stored procedure.
- Compile, linkedit the program.
- Bind the program at your local server and at each remote server. If your client program has a need to access multiple servers then you have to bind the program at each sever. Specify DBPROTOCOL as DRDA for bind at your local server.
- Bind all packages into a plan at local server. Specify DBPROTOCOL as DRDA.

15.2.2 Sample scenarios of program preparations

Let us consider a few examples where the client program access stored procedures, local and remote. Let us begin with Example 15-1 where we deal with a local stored procedure, and then move towards more complex remote calls.

Example 15-1 Client program invoking local stored procedure

```
Client program looks like:

   EXEC SQL
     CALL SP (parameter_list)
   END-EXEC.

BIND for client looks like (only important options are shown).

   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(BIND)

   BIND PLAN(plan_name) -
        PKLIST(col1.drpgm)
```

Example 15-1 shows the steps for preparing a client program to invoke local stored procedure. In this case both stored procedure and client access the same DB2 subsystem.

We now have a local and a remote stored procedure. As shown in Example 15-2, the client program has to issue a connect statement to connect to a remote DB2 server. For detailed information on connectivity topics, refer to the redbook *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952.

There is no difference in the CALL statement. There are two package binds for the same DBRM, once under local location and another one under remote DB2 location. The package bind for under local location should have DBPROTOCOL(DRDA) option. Also, observe that local package is bound with VALIDATE(RUN). As we made unqualified call to stored procedure, the local DB2 does not know about the stored procedure defined at remote server. During run time, this gets resolved as a CONNECT statement is issued before the CALL. We can overcome this by making a qualified call to stored procedure with three part name as *location.schema.name.* However, hard coding the qualifier poses some challenges with code portability. Another approach is to pass qualifier as a parameter to the program and making dynamic call to stored procedure. This approach causes an incremental BIND the first time the program is executed, then a cached entry is used. By using VALIDATE(RUN), we can make an unqualified call and resolve the stored procedure name during run-time. The option VALIDATE(RUN) is recommended for DRDA applications. DB2 issues warning messages for unresolved objects during BIND with VALIDATE(RUN) option.

Example 15-2 Client program invoking remote stored procedure

```
Client program looks like:

   MOVE 'DB2RLOC' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

   EXEC SQL
     CALL SP (parameter_list)
```

```
   END-EXEC.

BIND for client looks like (only important options are shown).

   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        DBPROTOCOL(DRDA) -
        VALIDATE(RUN)
   BIND PACKAGE(db2rloc.col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(BIND)

   BIND PLAN(plan_name) -
        PKLIST(col1.drpgm, db2rloc.col1.drpgm) -
        DBPROTOCOL(DRDA).
```

Both packages, local and remote, have to be bound to a local plan. At 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.

As shown in Example 15-3, in this case the client program has some SQLs to be processed at local DB2 server.The preparation looks similar to Example 15-2 except the remote package bind also has VALIDATE(RUN) option. As the tables referred in local DB2 are not known to the remote DB2, VALIDATE(RUN) option is required to bypass checking. Another change is the QUALIFIER option for local package. QUALIFIER is be used to qualify unqualified DB2 objects. PATH is used to qualify stored procedures and user defined functions.

Example 15-3 Client program with local SQL and invoking remote stored procedure

```
Client program looks like:

   EXEC SQL
     SELECT c1, c2, c3, c4 from ....
   END-EXEC.

   MOVE 'DB2RLOC' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

   EXEC SQL
     CALL SP (parameter_list)
   END-EXEC.

BIND for client looks like (only important options are shown).

   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        QUALIFER(qual) -
        DBPROTOCOL(DRDA) -
        VALIDATE(RUN)
   BIND PACKAGE(db2rloc.col1) -
```

```
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        VALIDATE(RUN)

   BIND PLAN(plan_name) -
        PKLIST(col1.drpgm, db2rloc.col1.drpgm) -
        DBPROTOCOL(DRDA).
```

As shown in Example 15-4, the client program has a requirement to invoke stored procedures at multiple locations. In this case, CONNECT statements are issued to different remote servers before invoking the stored procedure. Also observe that client program is bound as package at all remote servers and local server. The local plan is bound with local package and remote packages. Again, note the VALIDATE(RUN) for the stored procedure that is not defined locally.

Example 15-4 Stored procedures at multiple remote servers

```
Client program looks like:

   MOVE 'DB2RLOC' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

   EXEC SQL
     CALL SP (parameter_list)
   END-EXEC.

   MOVE 'DB2SLOC' TO HV-LOCATION.
   EXEC SQL
     CONNECT TO :hv-location
   END-EXEC.

   EXEC SQL
     CALL SP1 (parameter_list)
   END-EXEC.

   MOVE 'DB2PLOC' 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).

   BIND PACKAGE(col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
        DBPROTOCOL(DRDA) -
        VALIDATE(RUN)
   BIND PACKAGE(db2rloc.col1) -
        MEMBER(drpgm) -
        LIBRARY(dbrm_library_name) -
        PATH(schema) -
```

```
     VALIDATE(BIND)
BIND PACKAGE(db2sloc.col1) -
     MEMBER(drpgm) -
     LIBRARY(dbrm_library_name) -
     PATH(schema) -
     VALIDATE(BIND)
BIND PACKAGE(db2ploc.col1) -
     MEMBER(drpgm) -
     LIBRARY(dbrm_library_name) -
     PATH(schema) -
     VALIDATE(BIND)

BIND PLAN(plan_name) -
     PKLIST(col1.drpgm, db2rloc.col1.drpgm, -
            db2sloc.col1.drpgm, db2ploc.col1.drpgm) -
     DBPROTOCOL(DRDA).
```

At any point of time, a DB2 special register CURRENT SERVER indicates the server to which the thread is connected to. DB2 statement CONNECT RESET can be used to reset the connection back to local server.

15.2.3 Other considerations on preparing

Precompiler options

The following precompiler options are relevant to preparing a package to be run using DRDA access:

- CONNECT

  CONNECT(2) is the default and recommended option. CONNECT(1) causes your CONNECT statements to allow only three 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

  SQL(DB2) is the default option and is acceptable only for DB2 for OS/390 and z/OS. The precompiler rejects any statement that does not obey the rules of DB2 for OS/390 and z/OS.

  SQL(ALL) is for servers other than DB2 for OS/390 and z/OS. The precompiler accepts any statement that obeys DRDA rules.

BIND PACKAGE options

Only the options relevant to program preparation are discussed here:

- DBPROTOCOL(DRDA)

  This is only option to invoke ac stored procedure; DBPROTOCOL(PRIVATE) is not acceptable.

- ENCODING

  The default ENCODING value for a package that is bound at a remote DB2 server is the system default for that server. For applications that execute remotely and use explicit CONNECT statements, DB2 uses the ENCODING value for the plan. In case of implicit CONNECT statements, DB2 uses the value of the package that is set at the site where a statement executes.

BIND PLAN options

- DISCONNECT

  For most flexibility, use DISCONNECT(EXPLICIT), explicitly or by default. That requires you to use RELEASE statements in your program to explicitly end connections.

  The other values of the option are also useful.

  - *DISCONNECT(AUTOMATIC)* ends all remote connections during a commit operation, without the need for RELEASE statements in your program.
  - *DISCONNECT(CONDITIONAL)* ends remote connections during a commit operation except when an open cursor defined as WITH HOLD is associated with the connection.

- DBPROTOCOL

  Only DBPROTOCOL(DRDA) is accepted for programs with stored procedures.

ENCODING

For applications that execute remotely and use explicit CONNECT statements, DB2 uses the ENCODING value for the plan. In case of implicit CONNECT statements, DB2 uses the value of a package that is set at the site where a statement executes.

16

Code level management

In this chapter we discuss two most common issues with the management of stored procedures: How to version stored procedures within a DB2 subsystem and how to promote stored procedures from development to production. This chapter focusses on external stored procedures built in COBOL, C, PL/I etc., and internal stored procedures built with the SQL Procedures language. For management of Java stored procedures, refer to Part 4, “Java stored procedures” on page 243.

This chapter contains the following:

- Environments and levels
- Versioning of stored procedures
- Promotion of stored procedures
- Notes on REXX execs

16.1 Environments and levels

In a typical IT world, it is possible to have multiple environments and each with multiple releases of application code and stored procedures. Most organizations maintain at least three environments: development, quality, and production. Each environment may again have multiple levels to support parallel development and testing needs of the customers. A level represents code at one particular application release.

Table 16-1 shows an example of different environments and levels for an application. As shown, each level is designated for a specific purpose.

Table 16-1 Sample environments and levels

Environment name	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. So, 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. Please note that entire discussion in this section is surrounding a single application (normally, DB2 subsystems are shared by multiple applications).

We look at some of the common configurations:

- One DB2 subsystem per environment, see Figure 16-1.

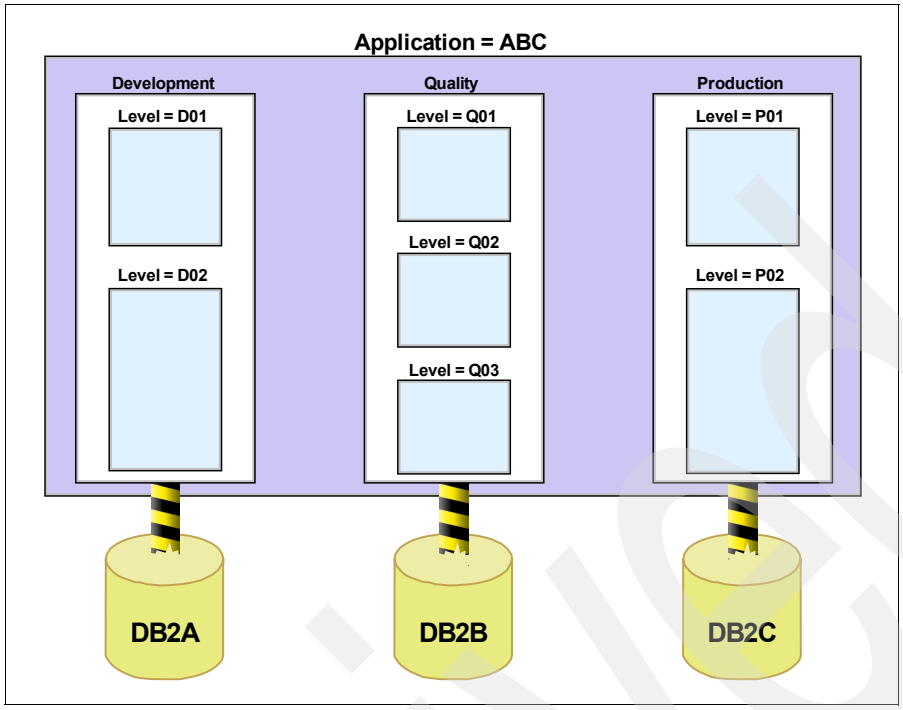


Figure 16-1 One DB2 subsystem for one environment

- One DB2 subsystem for two or more environments, see Figure 16-2.

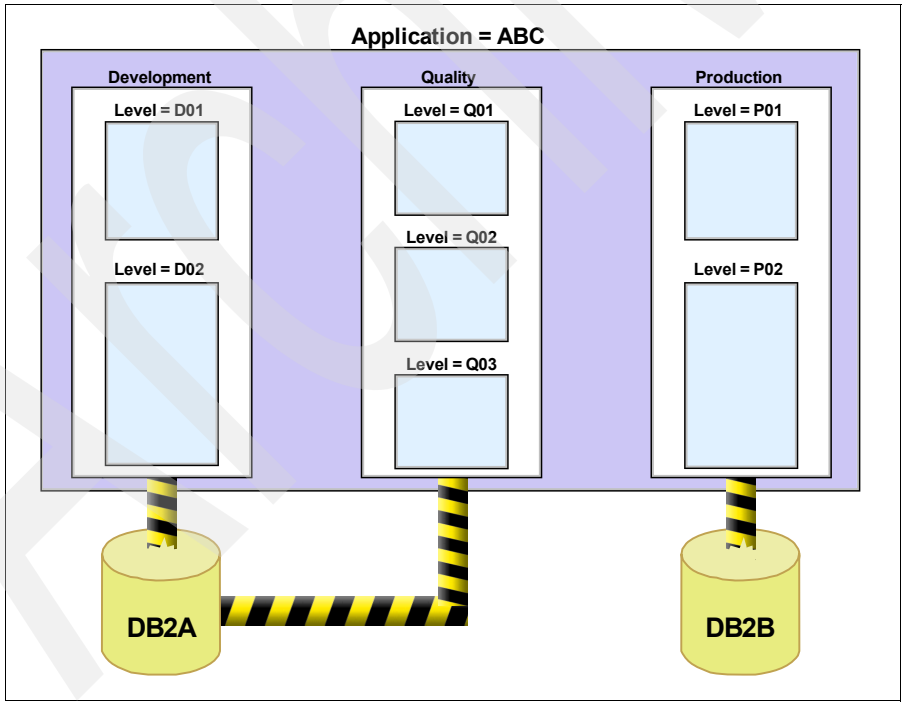


Figure 16-2 One DB2 subsystem for two environments

- One DB2 subsystem for one or more levels of an environment, see Figure 16-3.

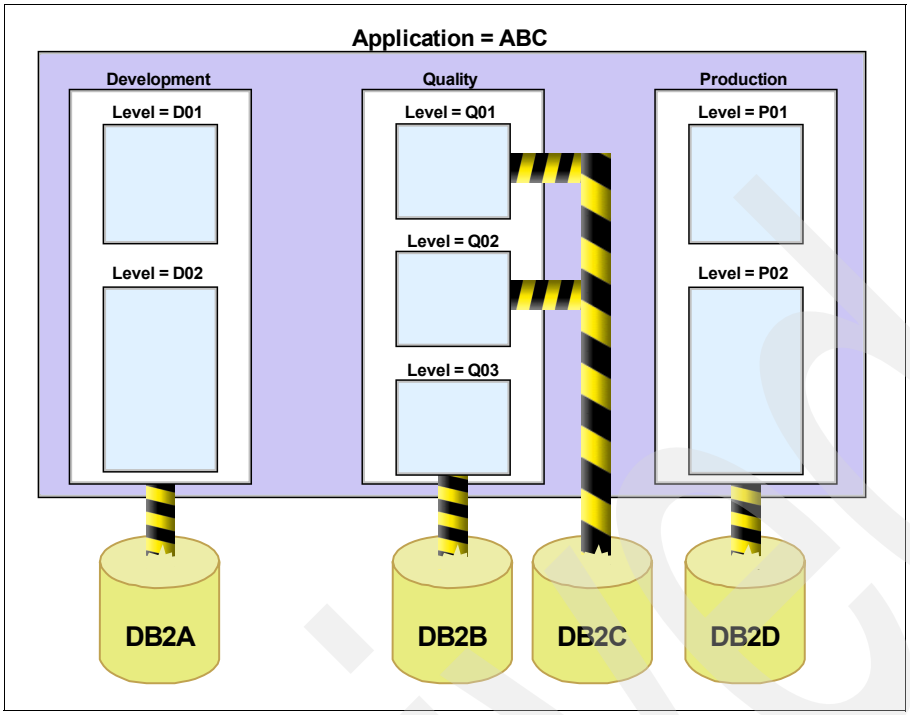


Figure 16-3 One or more levels of an environment within a DB2 subsystem.

Once the code is developed and tested, it will be promoted to production to serve the intended business functionality. Depending on the complexity of environments, configuration management of stored procedures can vary from simple process to most complex one. Each site should have a proper change management procedures in place for its smooth operation. Whatever the complexity of the environments, there are two challenges that exists with the configuration management of stored procedures:

- How to maintain different versions of stored procedures within a DB2 subsystem?
- How to promote the stored procedures from development environment to production environment?

The above two challenges gets complicated if your site supports different types of stored procedures like external (written in COBOL, C, PL/I, etc.), SQL Procedures language, and Java.

The following sections provide an approach to solve these two issues.

16.2 Versioning of stored procedures

DB2 uses three variables to identify and execute a stored procedure at run time, which are procedure name, package name, and load module name. In order to distinguish 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:

```
EXEC SQL
CALL PROC1 (:PARM1, :PARM2, :PARM3)
END-EXEC.
```

Figure 16-4 summarizes the connection among the variables at CALL time.

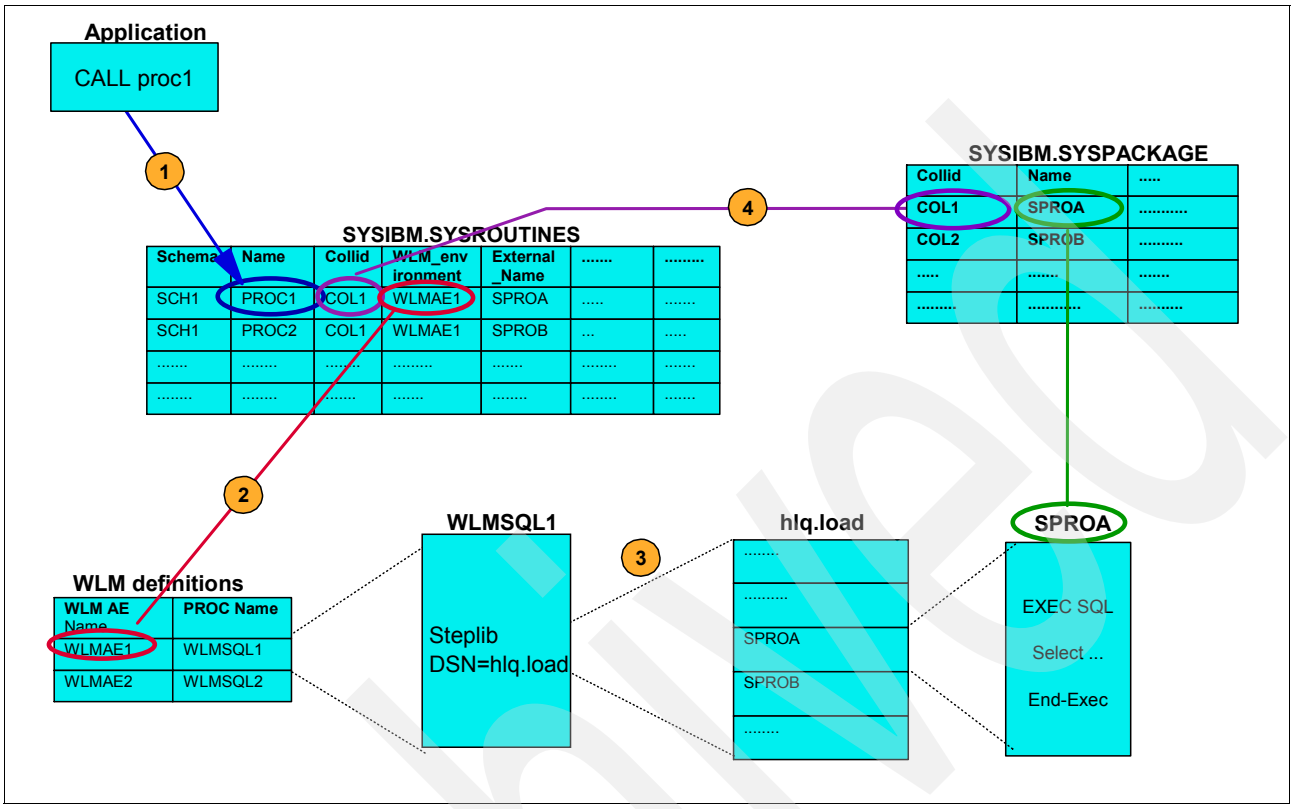


Figure 16-4 Relationship between schema, collid, and WLM AE at runtime

These are the steps:

1. DB2 searches for stored procedure PROC1 in SYSIBM.SYSROUTINES 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 run-time with SET CURRENT PATH statement within the caller.
2. DB2 sends a request to WLM to schedule stored procedure PROC1 at the WLM_ENVIRONMENT name retrieved from catalog table.
3. WLMSQL1, the WLM stored procedure address space, locates the load module SPROA from the data sets in 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. If possible, it is recommended to have the name of the stored procedure also within eight characters, and the same 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.

> **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 16-2 summarizes the relationship between the variables DB2 uses to locate a stored procedure and the qualifiers it uses.

Table 16-2 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 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.

16.2.1 Four release levels: Sample scenario

For example, consider a scenario where you have an application at four release levels, say Q01, Q02, Q03, and Q04, and all the levels are accessing same DB2 subsystem DB2Q. The stored procedure name is PROC1 and calling program name is CALLDR. At any point in time, the user should have a flexibility to invoke any version of the stored procedure. To make this possible, the following preparations are required.

Stored procedure preparation

These are the steps:

1. Make four definitions of stored procedure PROC1 with a unique combination of SCHEMA, COLLID, and WLM ENVIRONMENT name as shown in Example 16-1.

Example 16-1 Four uniquely names levels of stored procedures

```
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 linkedit the stored procedure PROC1. Ensure that the load module SPROA is stored in four different load libraries. For instance:

```
for level QO1, the stored procedure load library is 'hlq.ABC.DB2QWL1'
for level QO2, the stored procedure load library is 'hlq.ABC.DB2QWL2'
for level QO3, the stored procedure load library is 'hlq.ABC.DB2QWL3'
for level QO4, the stored procedure load library is 'hlq.ABC.DB2QWL4'
```

   The SPROA module exists in all the above libraries.

3. Bind the package SPROA with different collection IDs. The collection ID here should match with the collection ID specified in the CREATE PROCEDURE statement. If the stored procedure is defined with NO COLLID then the collection ID should match with the collection ID of 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 proc. In the JCL proc, concatenate the appropriate stored procedure load library. For instance:

```
JCL proc for DB2QWL1 will have 'hlq.ABC.DB2QWL1' in steplib.
JCL proc for DB2QWL2 will have 'hlq.ABC.DB2QWL2' in steplib.
JCL proc for DB2QWL3 will have 'hlq.ABC.DB2QWL3' in steplib.
JCL proc for DB2QWL4 will have 'hlq.ABC.DB2QWL4' in steplib.
```

Calling program preparation

These are the steps:

1. Compile and linkedit CALLDR program (which calls the PROC1 stored procedure). Ensure that unqualified reference is made to the stored procedure in CALLDR.
2. Bind CALLDR program with PATH option. Specify the schema name of the stored procedure in PATH option. Depending on the value specified in PATH option, DB2 invokes the corresponding stored procedure at runtime. It is also possible to set the value of PATH at run time using SET PATH statement in CALLDR.

.As shown in Table 16-3, the procedure name remains the same across all levels and its qualifier schema changes. Similar observations can be found for package name and external name.

Table 16-3 Sample versioning of 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:

```
Select Name, Schema, Collid, External_name, WLM_environment
from SYSIBM.SYSROUTINES
where name ='PROC1';
```

will extract from the catalog the different versions of a particular stored procedure.

Figure 16-5 shows how an application ABC at multiple releases levels (Q01 to Q04) can access the same DB2 subsystem(DB2Q) and distinguish multiple versions of a stored procedure based on the schema.

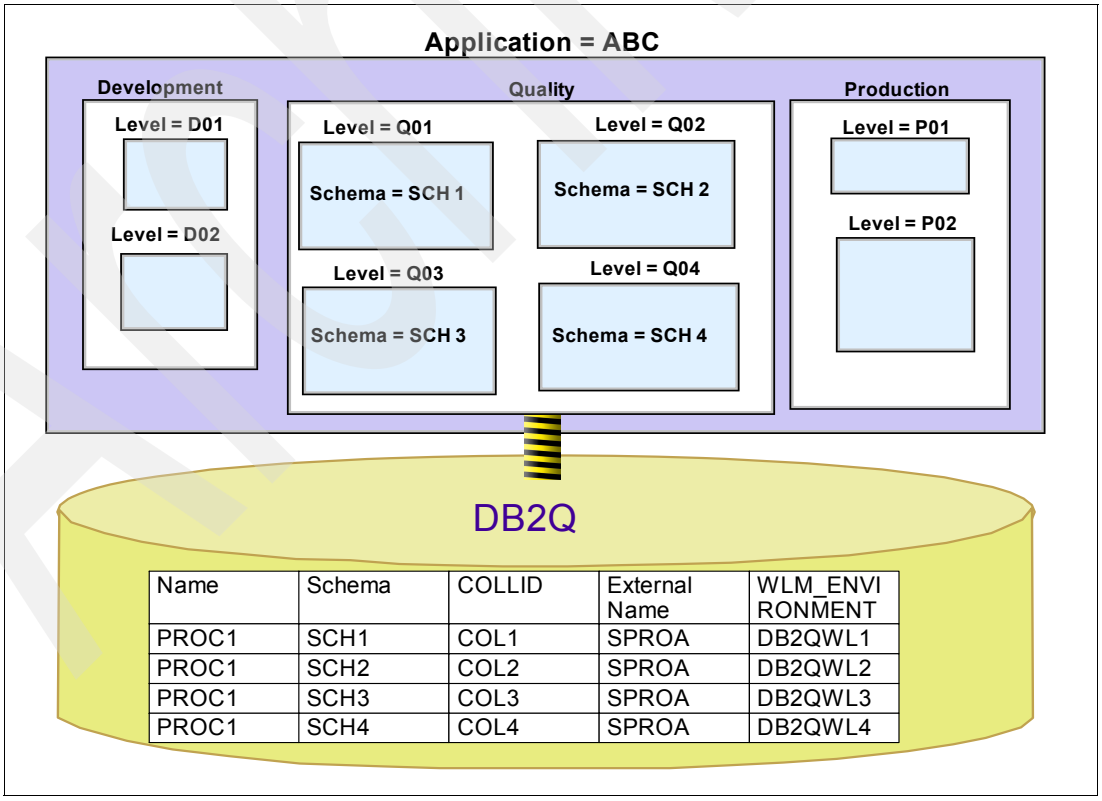


Figure 16-5 Sample versioning of stored procedures in a DB2 subsystem

16.3 Promotion of stored procedures

Once the code is developed and tested it will be 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 (create procedure statement). Depending on the type of stored procedure (external versus internal) the two parts may exist as two components or one component.

- **External**: Source code + DDL (create procedure statement).
- **Internal**: DDL (here source code is part of the CREATE PROCEDURE statement).

Promotion of stored procedures depends on the combination of two categories:

- Configuration management policy of your site
  - Compile only once in development
  - Compile in each environment (or level)
- Type of stored procedure
  - External
  - Internal

The following sections explain in detail the necessary steps involved in the promotion of stored procedures depending on the above two categories.

16.3.1 Compile just once

We differentiate between external and SQL stored procedures.

External stored procedures

Stored procedures that are developed in high level languages like COBOL, C, Java, PL/I etc., fall under this category.

Figure 16-6 shows the following steps.

Development activities

1. Define the stored procedure using the CREATE PROCEDURE statement.
2. Pre-compile, compile, and linkedit the source, which produces load module and 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/level

The steps are depicted in Figure 16-6.

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 level/environment.

> **Note:** A sample REXX DDLMOD and sample job DDLMODJB can be found in the Additional material. See the notes inside DDLMOD 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/level.

4. Copy the DBRM and 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.

5. Copy load module and refresh 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 hlq.SDSNSAMP data set to set up WLM_REFRESH job.

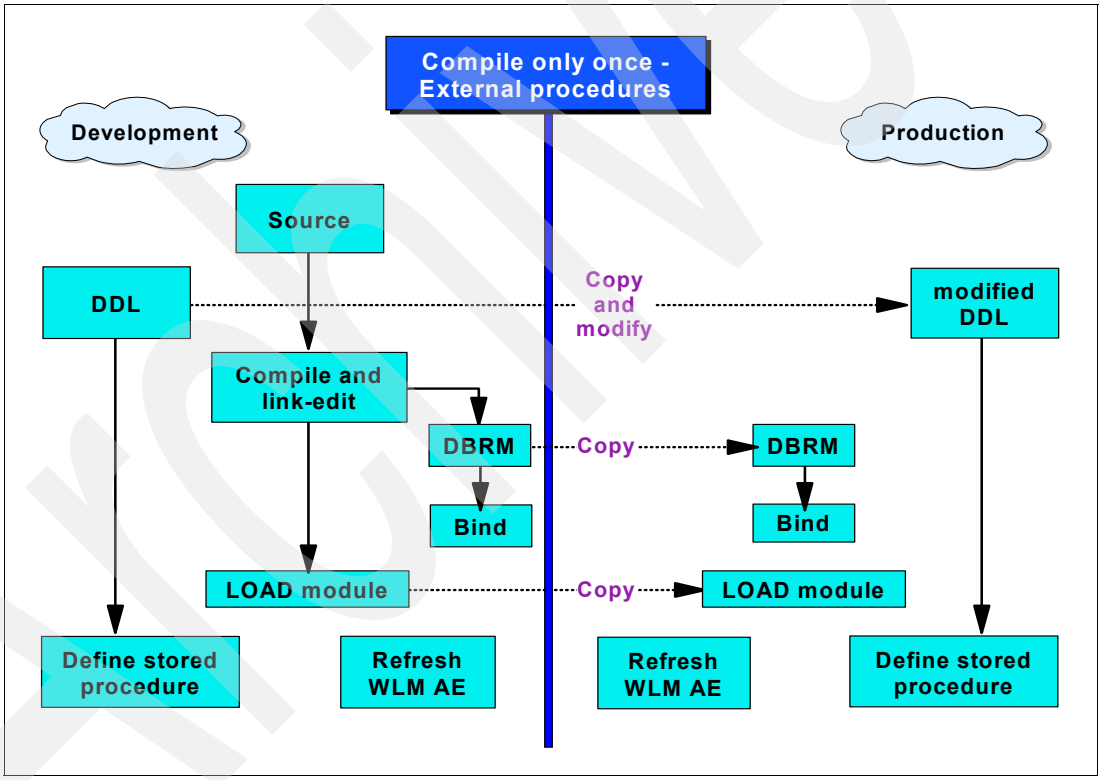


Figure 16-6 Promotion of external stored procedures - Compile only once

SQL stored procedures

Stored procedures built in the SQL Procedures language fall into this category. SQL stored procedures are mainly built either using Stored Procedure Builder (SPB) or its successor Development Center (DC) from workstations. It is also possible to build them on z/OS using DSNTPSMP stored procedure in batch mode. For traditional developers on z/OS server, it is possible to build them using ISPF editor and prepare, pre-compile, compile, and linkedit the process. The source of SQL stored procedures, built using either SPB or DC or DSNTPSMP, is stored in DB2 catalog tables (SYSROUTINES_SRC and SYSROUTINES_OPTS).

The DB2Build utility, which executes on the client side (UNIX and Windows), can be used to promote stored procedures. But this utility recompiles the program. Since your site's policy is to compile *just once in development*, you should have a procedure on z/OS to promote stored procedures without using SPB/DC or DB2Build. In this section we discuss an approach to solve this issue.

Figure 16-7 shows the following steps.

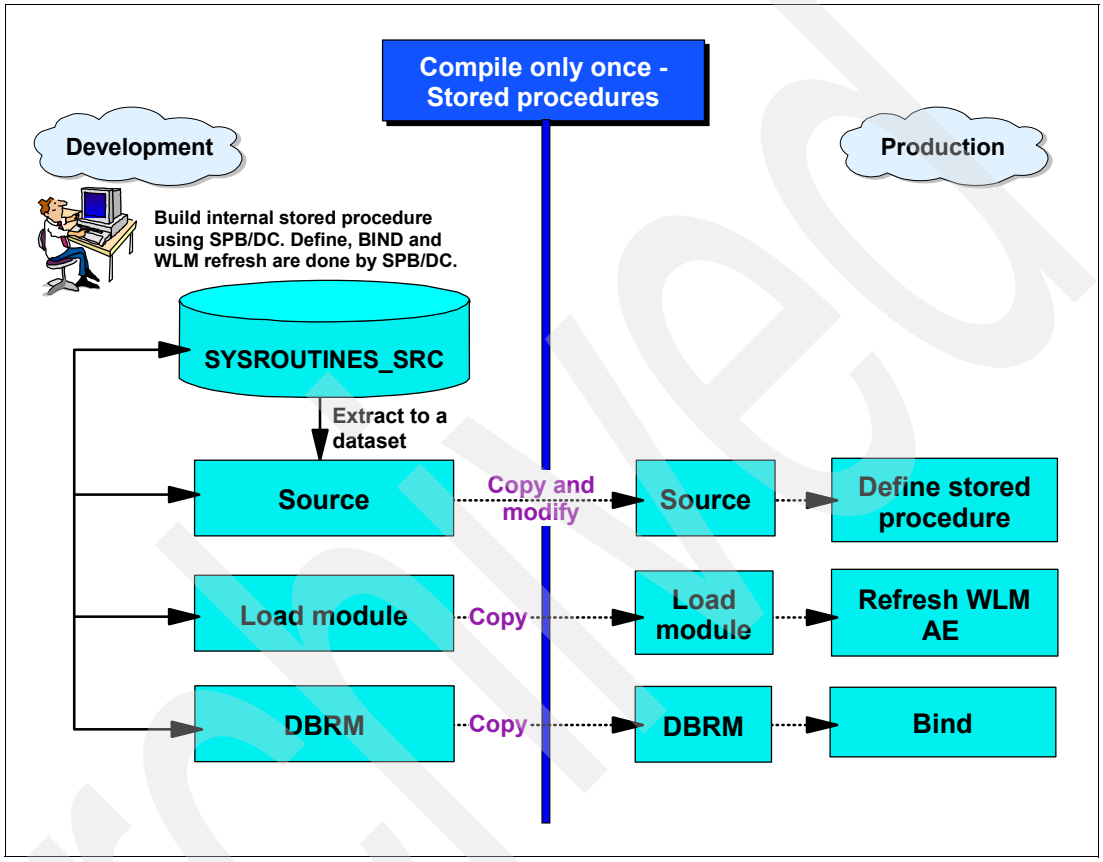


Figure 16-7 Promotion of SQL stored procedures - Compile only once

Development activities

1. Build SQL procedure using either SPB/DC or by calling DSNTPSMP in batch mode, or in a traditional way with the ISPF editor.
2. Test and verify the stored procedure functionality.

Promotion and installation activities in target environment/level

Note that the steps depicted in Figure 16-7 should all be executed on z/OS:

1. Extract DDL from the DB2 catalog (SYSROUTINES_SRC) to a data set[1], if you built the stored procedure with SPB/DC.
2. Modify DDL to reflect new schema, new collection ID, and the new application environment (WLM AE) corresponding to your promotion level/environment[2].
3. Define the stored procedure using the modified DDL. IBM supplied programs DSNTIAD or DSNTEP2 can be used. Note that DSNTIAD does not allow comment lines (--) in the

[1] A sample REXX 'GETSQLSP' and sample job 'GETSQLJB' can be found in additional material.
[2] A sample REXX 'DDLMOD' and sample job 'DDLMODJB' can be found in additional material.

input. For DSNTEP2, change the statement delimiter to other than '**;**' using SQLTERM parameter.

> **Note:** Ensure that SCHEMA, COLLID, and WLM AE corresponds to the new environment/level.

4. Copy DBRM and bind DBRM to create a DB2 package.

> **Note:** Ensure that the collection ID of BIND statement, and COLLID of the CREATE PROCEDURE statement are the same.

5. Copy load module and refresh 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 DSNTEJ6W job in hlq.SDSNSAMP data set to set up WLM_REFRESH job

16.3.2 Compile every time

We differentiate between external and SQL stored procedures.

External stored procedures

Stored procedures developed in high level languages like COBOL, C, Java, PL/I etc., fall under this category.

Figure 16-8 shows the following steps.

Development activities

1. Define the stored procedure using the CREATE PROCEDURE statement.
2. Pre-compile, compile, and linkedit the source, which produces load module and 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 next environment/level

1. Copy DDL (create procedure statement).
2. Modify DDL to reflect new schema, new collection ID, and new application environment (WLM AE) to reflect new environment/level[3].
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/level.

4. Copy the source.
5. Pre-compile, compile, and linkedit source, which produces DBRM and the load module.
6. Bind 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.

[3] A sample REXX 'DDLMOD' and sample job 'DDLMODJB' can be found in additional material.

7. Refresh 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.

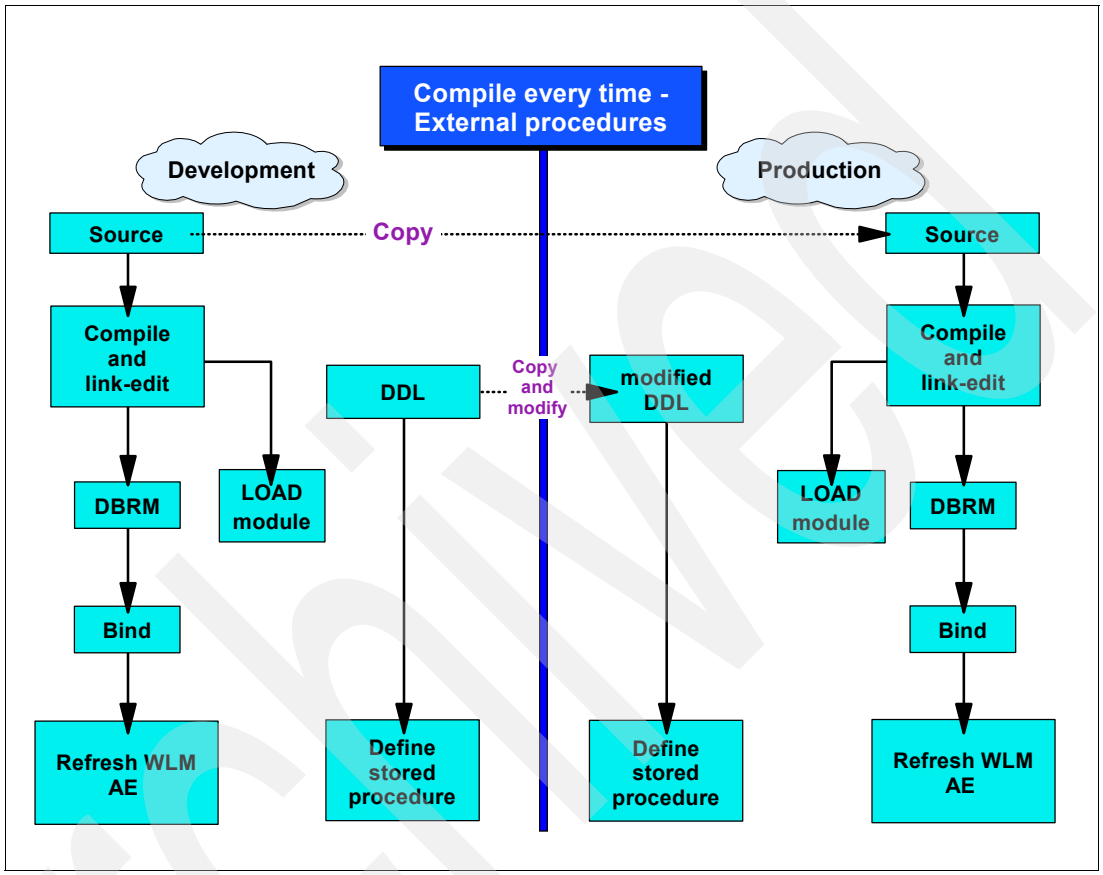


Figure 16-8 Promotion of external stored procedures - Compile every time

SQL stored procedures

For sites where compilation is allowed every time, two approaches can followed to promote code from development to production:

- DB2Build utility - complete promotion can be done from a workstation.
- A customized process on z/OS or OS/390 servers

DB2Build utility

An IBM supplied utility runs on workstation, either on UNIX or Windows, and can promote stored procedures between two z/OS servers. This is a great tool for sites where the development and testing of SQL stored procedures happen completely from a workstation, and DB2Build provides a way even to promote.

Some of the salient features of DB2Build utility are:

- Provides an online interface to migrate SQL stored procedures created on one DB2 for z/OS server to another DB2 for z/OS server.
- Can be run from a DB2 V7/V8 client on either UNIX or Windows.
- Uses the DSNTPSMP REXX stored procedure on both source and target DB2 servers.

- Available through FixPak 2 for DB2 UDB V8 on both Windows and UNIX. For V7.2, the code is available from a Web download.
- Multiple stored procedures can be promoted in one pass.

For a complete reference on usage of the DB2Build utility, refer to the IBM developerWorks® Web site and do search on DB2Build. The developerWorks Web site can be found at the following URL:

`http://www.ibm.com/developerworks`

Figure 16-9 shows the usage of the DB2Build utility.

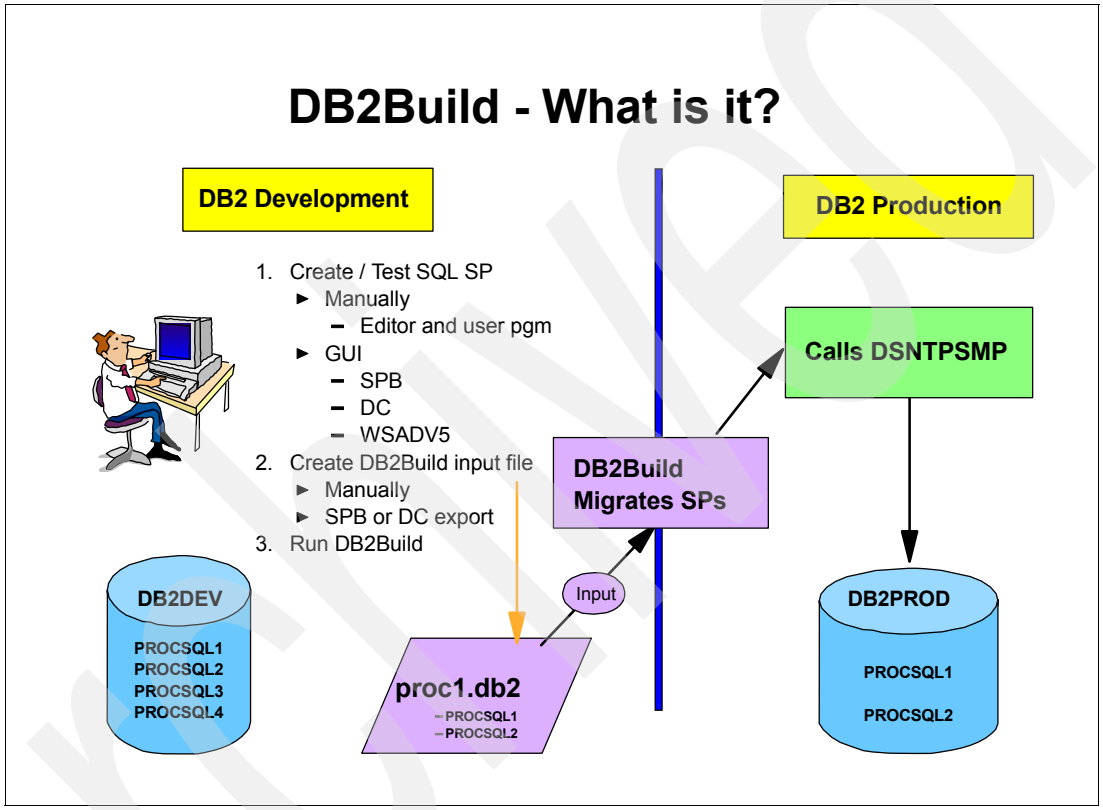


Figure 16-9 DB2Build Utility

For sites where version control of stored procedures happen on z/OS servers along with other components of the application (like batch programs, CICS programs etc.) the following process can be followed.

Customized process on z/OS or OS/390 server

Why is this process required?

SPB or DC is a great tool for development of stored procedures and for unit testing of the same. As you are aware, once the procedure is built, the source of the SQL stored procedure will be stored in SYSIBM.SYSROUTINES_SRC catalog table. Currently, it's not possible to version (for a particular SCHEMA.ROUTINENAME) stored procedures within the catalog table. Also each site has it's own tools or products to do configuration management built around data sets, which may or may not interact with DB2. As shown in Figure 16-10, a customized process has been developed to extract the source from the DB2 catalog table and store it in a data set. Once the source is available in a data set, it can be treated as any other programming language source like COBOL, PL/I, etc.

Development activities

1. Build SQL procedure using either SPB/DC or by calling DSNTPSMP in batch mode.
2. Test and verify the stored procedure functionality.

Promotion and installation activities in target environment/level

Note: These steps should happen on z/OS or OS/390.

These are the steps:

1. Extract DDL from DB2 catalog (SYSROUTINES_SRC) to a data set[4].
2. Modify DDL to reflect new Schema, new Collection ID and new Application environment (WLM AE) to reflect new environment/level[5].
3. Define the stored procedure using the modified DDL.
4. Pre-compile, compile and linkedit the SQL stored procedure using DSNHSQL procedure.
5. Bind DBRM to create a DB2 package.
6. Refresh WLM AE.

Tip: The sample job SQLSPCUS, described in Chapter D, “Additional material” on page 651, can be used for steps 2 to 6.

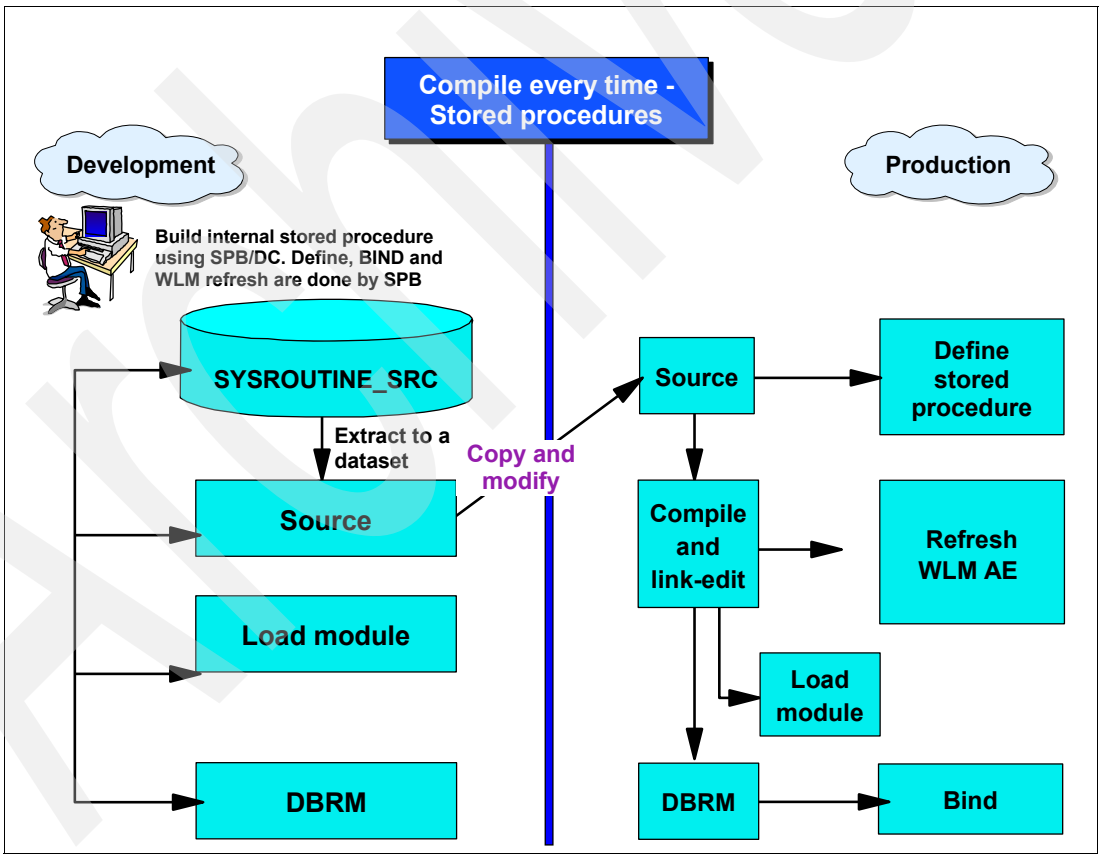


Figure 16-10 Promotion of SQL stored procedures - Compile every time

[4] A sample REXX ‘GETSQLSP’ and sample job ‘GETSQLJB’ can be found in additional material.
[5] A sample REXX ‘DDLMOD’ and sample job ‘DDLMODJB’ can be found in additional material.

16.4 Notes on REXX execs

Throughout this chapter there were several references to sample REXX execs used in configuration management. This section gives a high level overview of each of the REXX execs:

- GETSQLSP
- PUTSQLSP
- DDLMOD

16.4.1 GETSQLSP

Functionality

This REXX exec extracts the source code of an SQL stored procedure from the DB2 catalog table SYSROUTINES_SRC and stores it in a data set.

Input parameters

SSID: DB2 subsystem ID where the source of the stored procedure is stored

Schema: Schema of the stored procedure

Spname: Stored procedure name, whose source has to be retrieved

Output data sets

SQLSPOUT: The source code of the SQL stored procedure

Usage notes

- The version of GETSQLSP provided in the additional materials is what we used for our V7 procedures based on the version of DSNTPSMP that we used in our case studies. Depending on your DB2 version and maintenance level you may need to modify the sample GETSQLSP procedure to match the line formatting characters used in DSNTPSMP in your environment..
- Once the source is retrieved into a data set, it can be treated like any other source with respect to configuration management.

Sample job to invoke GETSQLSP

Example 16-2 provides a sample job for invoking GETSQLSP.

Example 16-2 Sample job to invoke GETSQLSP

```
//GETSQLJB JOB (999,POK),'GET SQL JOB',CLASS=A,MSGCLASS=A,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
//BATCREXX EXEC PGM=IKJEFT01
//STEPLIB  DD DISP=SHR,DSN=DB2G7.SDSNLOAD
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SQLSPOUT DD DISP=SHR,DSN=SG247083.TEST.SQLPROCS(SQLSP11)
//*
//*SYSEXEC contains REXX procedure.
//*
//SYSEXEC  DD DISP=SHR,DSN=SG247083.DEVL.CLIST
//SYSIN    DD DUMMY
//*   Arguments GETSQLSP are ssid schema spname
//*
```

```
//*   where     ssid = DB2 subsystem ID where SQL procedure is stored.
//*           schema = Schema of the stored procedure.
//*           spname = Name of the stored procedure
//SYSTSIN  DD *
%GETSQLSP DB2G PAOLOR9 SQLSP6
/*
```

16.4.2 PUTSQLSP

Functionality

This REXX exec retrieves the source code for an SQL stored procedure from a data set and inserts the source code into the DB2 catalog table SYSROUTINES_SRC.

Input parameters

SSID: DB2 subsystem ID where the source of the stored procedure has to be stored

Input data set

SQLSPINP: The source code of an SQL stored procedure

Output

None

Usage notes

- The version of PUTSQLSP provided in the additional materials is what we used for our V7 procedures based on the version of DSNTPSMP that we used in our case studies. Depending on your DB2 version and maintenance level you may need to modify the sample PUTSQLSP procedure to match the line formatting characters used in DSNTPSMP in your environment..
- It provides a way to store source of SQL procedures, developed using ISPF interface, in the SYSROUTINES_SRC table. Once the source is available in the DB2 catalog table, SPB or DC can access it.
- For sites where the main development of stored procedures happen on SPB or DC, the GETSQLSP and PUTSQLSP execs help to provide a complete change management life cycle; for example, a build SQL procedure using SPB. Extract the source into a data set and use it for version control purposes within your configuration management tools and products. Promote it to production. If you have to enhance or bug fix the SQL procedure, then using PUTSQLSP insert the source into the SYSROUTINES_SRC table. Using SPB or DC, enhance the source or fix the bugs.

Sample job to invoke PUTSQLSP

Example 16-3 provides a sample job for invoking PUTSQLSP.

Example 16-3 Sample job to invoke PUTSQLSP

```
//PUTSQLJB JOB (999,POK),'PUT SRC JOB',CLASS=A,MSGCLASS=A,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
//BATCREXX EXEC PGM=IKJEFT01
//STEPLIB  DD DISP=SHR,DSN=DB2G7.SDSNLOAD
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SQLSPINP DD DISP=SHR,DSN=SG247083.TEST.SQLPROCS(SQLSP6)
//*
```

```
//SYSEXEC  DD DISP=SHR,DSN=SG247083.DEVL.CLIST
//SYSIN    DD DUMMY
//*  Argument to PUTSQLSP is ssid
//*
//*   where     ssid = DB2 subsystem ID where SQL procedure has to be
//*                    stored.
//SYSTSIN  DD *
%PUTSQLSP DB2G
/*
```

16.4.3 DDLMOD

Functionality

- Modifies the DDL based on the values specified in the configuration file
- Generates SYSIN cards, which can be used for WLM refresh job, the DROP PROCEDURE statement and the SET CURRENT SQLID statement

Input parameter

Level: An unique ID representing an environment or level

Input data sets

- DDLINPSP: Input data set with DDL
- CFGFILE: Configuration file with environment/level details

Example 16-4 shows the sample contents of the configuration file

Example 16-4 Sample contents of the configuration file

```
Level/Env SSID      Schema   CollID    WLMAE     SQLID
--------- -----    -------- -------  ---------  ---------
D01       DB2G      DSCH1    DCOL1     DB2GWL1   DB2GUSR1
D02       DB2G      DSCH2    DCOL2     DB2GWL2   DB2GUSR2
Q01       DB2Q      QSCH1    QCOL1     DB2QWL1   DB2QUSR1
Q02       DB2Q      QSCH1    QCOL1     DB2QWL2   DB2QUSR2
Q03       DB2Q      QSCH1    QCOL1     DB2QWL3   DB2QUSR3
Q04       DB2Q      QSCH1    QCOL1     DB2QWL4   DB2QUSR4
P01       DB2P      PSCH1    PCOL1     DB2PWL1   DB2PUSR1
P02       DB2P      PSCH1    PCOL1     DB2PWL2   DB2PUSR2
```

Output data sets

- DDLOUTSP: Modified DDL
- SETSQLID: Contains SET CURRENT SQLID statement
- DROPSP: Contains DROP PROCEDURE statement
- WLMRFRSH: Contains SYSIN cards for WLM refresh job

Usage notes

- DDLMOD REXX modifies the input DDL for schema, the collection ID, and the WLM application environment name.
- DDLMOD can read the output of the GETSQLSP exec and modify it in preparation for running PUTSQLSP on another system, or it can read a file that you have created that contains SQL procedure source code.

- If your site requires some more parameters to be modified between environments/levels, the above REXX can be customized.
- The output produced in 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 16-5 provides a sample job for invoking DDLMOD.

Example 16-5 Sample job to invoke DDLMOD

```
//DDLMODJB JOB (999,POK),'DDL MOD JOB',CLASS=A,MSGCLASS=A,
// 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.TEST.SQLPROCS(EMPDTLSS)
//*CONFIGURATION FILE WITH ENVIRONMENT/LEVEL DETAILS
//CFGFILE  DD DISP=SHR,DSN=SG247083.DEVL.CLIST.CONFIG(APPLABC)
//DDLOUTSP DD DISP=SHR,DSN=SG247083.DEVL.DDLMODO(EMPDTLSS)
//SETSQLID DD DISP=SHR,DSN=SG247083.DEVL.DDLMODO(SETSQLID)
//DROPSP   DD DISP=SHR,DSN=SG247083.DEVL.DDLMODO(DROPSP)
//WLMRFRSH DD DISP=SHR,DSN=SG247083.DEVL.DDLMODO(WLMRFRSH)
//*
//SYSEXEC  DD DISP=SHR,DSN=SG247083.DEVL.CLIST
//SYSIN    DD DUMMY
//SYSTSIN  DD *
%DDLMOD DO2

/*
```

For external 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

Java stored procedures

Java and DB2 for z/OS work together in order to allow you to run mission critical applications written in Java. Several enhancements have been introduced by IBM to Java functions, such as 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, and the dismissal of compiled Java.

This part is dedicated to Java stored procedures. First, we discuss the Java stored procedures environment set up, and their definition with examples using JDBC and SQLJ; then we introduce the new Java Universal Driver, and provide some recommendations on usage and migration.

This part contains the following chapters:

17

Building Java stored procedures

In this chapter we describe how to set up the environment for Java stored procedures, and how to code and debug them. We assume that you have a basic understanding of the Java language.

This chapter contains the following:

- Overview of Java stored procedures
- Setting up the environment for Java stored procedures
- Persistent Reusable JVM
- Considerations on static variables
- Preparing Java stored procedures
- DDL for defining a Java stored procedure
- Debugging JDBC and SQLJ
- Java sample JDBC stored procedures
- Java sample SQLJ stored procedures

17.1 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 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 and OS/390 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 redbook 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. A source of reference information is *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java,* SC18-7414. In this redbook we concentrate on *Java stored procedures*.

A Java stored procedures can come in two flavors: compiled Java stored procedures and interpreted Java stored procedures. While DB2 V7 supports development of both types of procedures, DB2 V8 only supports the development of *interpreted Java stored procedures*. The Development Center application development tool only supports creating interpreted Java stored procedures on DB2 V7 and DB2 V8. It is strongly recommended to move away from compiled Java stored procedures. In this book we only discuss interpreted Java stored procedures, which do not use HPJ compilers or Visual Age for Java Enterprise Toolkit libraries on the host. You can develop interpreted Java stored procedure 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 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 that make dynamic SQL calls.

We developed several JDBC and SQLJ Java stored procedures. Please refer to Chapter 3, "Our case study" on page 21 for a comprehensive list of JDBC and SQLJ stored procedures.

17.2 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 Java stored procedures.

17.2.1 Prerequisite software for Java stored procedure

The major prerequisites are:

- DB2 for z/OS at Version 7.
- OS/390 Version 2 Release 8.
- IBM Developer Kit for OS/390, Java 2 Technology Edition, SDK 1.3.1 or 1.4.1 level (5655-A46). For information on installing the Java SDK refer to the following URL:

  `http://www-1.ibm.com/servers/eserver/zseries/software/java/`

Consider also applying the PTFs for the following APARs:

- APAR PQ46673 - In case you plan to use the Jar Support
- APAR PQ51847.- Incase you plan to use LOBs support

If you plan to use the Development Center for developing Java stored procedures you need to check the APARs:

- APAR II13277 - For Unicode Support for Development Center (DC) and Stored Procedure Builder (SPB)
- APAR PQ65125 - Development Center SQLJ support
- APAR PQ62695 - SQL Assist support

17.2.2 Checking that the Java SDK is at the right level

To check that you have the appropriate Software Developers Kit (SDK) release, log on to UNIX System Services (USS) and issue the UNIX commands below. You need to set the PATH environment variable to the Java bin directory. The Java bin directory can normally be found in /usr/lpp/java/IBM/J1.3. Sometimes your systems programmer could have loaded the Java libraries into a different directory. Issue the export command to assign the PATH variable to the Java bin directory.

```
=> export PATH=/usr/lpp/java/IBM/J1.3/bin:$PATH
=> java -version
```

You should get the following messages on your terminal. First, you are looking for the *Persistent Reusable VM* portion in the message. Second, you should look for a Java version 1.3.1 or 1.4.1:

```
# java -version
      java version "1.3.1"
      Java(TM) 2 runtime Environment, Standard Edition (build 1.3.1)
      Classic VM (build 1.3.1, J2RE 1.3.1 IBM OS/390 Persistent Reusable VM build cm13
      1s-20030913 (JIT enabled: jitc))
```

17.2.3 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 *IBM DATABASE 2 Universal Database Server for OS/390 and z/OS Program Directory* 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/db2710, and issue the list directory command. In our case the DB2 home directory was /usr/lpp/db2/db2g:

```
=> cd /usr/lpp/db2/db2g
=> ls
```

You should see the following directories:

```
IBM             bin             installVAJDLLs  samples
README          classes         lib :
```

17.2.4 Checking the build level of SQLJ/JDBC driver

New diagnostics have been added to determine the SQLJ/JDBC driver build level. In support of this, a new Java class, COM.ibm.db2os390.sqlj.util.DB2DriverInfo, has been added. The DB2DriverInfo class includes a Java *main,* which prints the SQLJ/JDBC Driver build version. This can be run from the USS command line with the command:

```
java COM.ibm.db2os390.sqlj.util.DB2DriverInfo
```

Example 17-1 shows the output of the above command.

Example 17-1 Checking the driver version

```
/u/paolor7:>java COM.ibm.db2os390.sqlj.util.DB2DriverInfo
DB2 for OS/390 SQLJ/JDBC Driver build version is: DB2 7.1 PQ69861 JDBC 2.0
/u/paolor7:>
```

17.2.5 Setting up the WLM procedure

You need to have a WLM JCL proc corresponding to the WLM application environment defined for executing the Java stored procedures. The sample procedure that we used is shown in Example 17-2.

Example 17-2 WLM proc 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.
//*
//*************************************************************
//DB2GWEJ1 PROC APPLENV=DB2GWEJ1,DB2SSN=DB2G,NUMTCB=1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=0M,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=DB2G7.SDSNEXIT
//         DD  DISP=SHR,DSN=DB2G7.SDSNLOAD
//         DD  DISP=SHR,DSN=DB2G7.SDSNLOD2
//*        DD  DISP=SHR,DSN=CEE.SCEERUN
//*   NEED UNAUTHORIZED DATASET
//         DD  DISP=SHR,DSN=CBC.SCBCCMP
//JAVAENV  DD  DSN=DB2G7.JAVAENV
//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 proc for Java:

- Keep the NUMTCB to a low number, an optimum number can be 6 to 7 for running Java stored procedures in production. For every NUMTCB, WLM would create a JVM in the WLM address space. Having a very high value of NUMTCB can result in a high usage of memory. Also, a WLM refresh of the address space would take longer. We set the NUMTCB=1 in our WLM address spaces used for test to allow for faster refreshing of the WLM environment after we made code changes. The lower the NUMTCB value, the faster the refresh time.
- 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 run-time 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 LLA to reduce I/O.
- Apart from including SDSNEXIT, SDSNLOAD libraries in the STEPLIB, do not forget to include the SDSNLOD2 library. 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 mentioned in 17.2.6, “Setting up the JAVAENV data set for Java stored procedure execution” on page 249.
- JSPDEBUG DD statement specifies a data set into which DB2 puts information that you can use to debug your stored procedure. 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. You should comment out this DD statement during production. Information regarding each and every invocation of a Java stored procedure is written to the JSPDEBUG data set by DB2. Example 17-3 shows the sample JSPDEBUG output produced when a stored procedure was invoked.

Example 17-3 JSPDEBUG output from an invocation of a stored procedure

```
Just entered pq71286 version at time:  Fri Dec  5 15:11:43 2003

Default encoding is 37; as CCSID char: 'Cp037'; as iconv char: 'IBM-037'
Generated signature before convert: (Ljava/lang/String;ÝLjava/lang/String;ÝLjava
ang/String;ÝLjava/math/BigDecimal;ÝLjava/lang/String;)V
Processing IN and INOUT parameters of the Java method
  parm 1 is String: '000010' CCSID: 37
invoking class: EmpDtlsJ, method: GetEmpDtls
Back from Call: Processing time was 0.101600
Processing OUT and INOUT parameters of the Java method
  parm 2 is String: 'CHRISTINE' CCSID: 37
  parm 3 is String: 'I' CCSID: 37
  parm 4 is String: 'HAAS' CCSID: 37
  parm 5 is String: 'A00' CCSID: 37
  parm 6 is BigDecimal: 52750.01, size: 9, precision: 2
  parm 7 is String: ' ' CCSID: 37
Number of result sets is 0
Return parm is 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 17.7, “Debugging JDBC and SQLJ” on page 266. They are only required if you plan to debug your Java stored procedures using the System.out.println method.

17.2.6 Setting up the JAVAENV data set for Java stored procedure execution

The WLM proc 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 17-1.

Table 17-1 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 17-4.

Example 17-4 Contents of JAVAENV - DB2G.JAVAENV file

```
ENVAR("DB2_HOME=/usr/lpp/db2/db2g",
"JAVA_HOME=/usr/lpp/java/IBM/J1.3_V030510A",
"CLASSPATH=/SC63/sg247083/spjava"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

All the environment variables need to be included in this file. Ensure that the total length of all the entries put together does not exceed 245 bytes (exclude the blanks). In case your entries exceed the 245 byte limit, you need to take a different approach as shown in Example 17-5. Here we show an alternate form of JAVAENV definitions.

Example 17-5 Contents of JAVAENV having _CEE_ENVFILE variable

```
ENVAR("_CEE_ENVFILE=/usr/lpp/db2/db2g/envfile.txt",
"JAVA_HOME=/usr/lpp/java/IBM/J1.3_V030510A"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

The _CEE_ENVFILE variable points to an HFS file that contains most of the environment variables, as 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 17-6. This is a standard UNIX file where each line must start in column 1 and the continuation character is a \.

Example 17-6 Contents of the _CEE_ENVFILE - `/usr/lpp/db2/db2g/envfile.txt`

```
DB2_HOME=/usr/lpp/db2/db2g
CLASSPATH=/SC63/sg247083/spjava
JVMPROPS=/usr/lpp/db2/db2g/jvmsp
```

You can use the _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.

17.2.7 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 17-2.

Table 17-2 Contents of a JAVAENV data set

Environment variable	Description
DB2_HOME	This environment variable indicates to DB2 that the WLM environment is for interpreted Java routines. The value of DB2_HOME is the highest-level directory in the set of directories that contain the JDBC/SQLJ driver. For example: /usr/lpp/db2/db2710. In our case we had our DB2 libraries in /usr/lpp/db2/db2g.
JCC_HOME	The contents of the JAVAENV file decide if you are running the new JCC driver or using the Legacy Driver. If you want to use the JCC driver, you need to set the JCC_HOME variable to the location of the JCC driver. You should not have the definitions of JCC_HOME and DB2_HOME in the same JAVAENV file. See 18.2.1, “JAVAENV for DB2 stored procedures” on page 279.
JAVA_HOME	This environment variable indicates to DB2 that the WLM environment is for interpreted 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/J1.3
CLASSPATH	The directory where you place your compiled stored procedures. A detailed discussion of CLASSPATH can be found in “Making the stored procedure class files available to DB2” on page 260
TMSUFFIX	A list (similar to CLASSPATH) where anything appearing in TMSUFFIX is treated as Trusted Middleware class. See “TMSUFFIX” on page 251.
JVMPROPS	The Persistent Reusable JVM improves transaction processing throughput by providing the ability to run multiple JVMs within an z/OS address space. See “JVMPROPS” on page 252.
RESET_FREQ	This variable overrides the default frequency of JVM reset operations, which by default is every 256 procedure invocations. The primary reason for overriding this value is for testing purposes. For example: "RESET_FREQ=5"

TMSUFFIX

On OS/390 and z/OS, Java stored procedures can reference classes that are categorized into the following groups:

- **nonShared Application classes**

  Mentioned for completeness, no other description included

- **Sharable Application classes**

  These are normally the user’s application classes included in the WLM AE JAVAENV statement CLASSPATH environment variable or included in an jar installed into the DB2

catalog that is referenced in the EXTERNAL NAME clause of the CREATE PROCEDURE/FUNCTION.

- **Trusted Middleware classes**

  These include other environment variables used by Java stored procedures including JAVA_HOME, DB2_HOME. Additionally, CICS-support, IMS-support, and any application designed to do actions that would cause the JVM to become unresettable are included in this group.

- **The JVMs own classes**

  Self-explanatory

Any class defined as a Trusted Middleware class is allowed to break all rules that would cause a Sharable Application class to have an unresettable event problem. This includes breaking things in such a way that unrelated Java stored procedures, executing in the same WLM AE, start experiencing odd behavior. Allowing classes to be defined as a Trusted Middleware class should be done with caution.

For the complete set of rules regarding what makes a JVM unresettable, see *New IBM Technology featuring Persistent Reusable Java Virtual Machines*, at the Web site:

`http://www-1.ibm.com/servers/eserver/zseries/software/java`

TMSUFFIX is similar to CLASSPATH. It specifies a "**:**" separated list of directories and jar files that will be searched for class references. The difference is that anything appearing in TMSUFFIX will be treated as a Trusted Middleware class. If a directory or jar appears in both CLASSPATH and TMSUFFIX, it will not be loaded as Trusted Middleware.

Example 17-7 describes the JAVAENV data set including the TMSUFFIX envar where the directory of the contained classes is */u/TrustedSP*.

Example 17-7 Contents of JAVAENV including TMSUFFIX envar - DB2G JAVAENV file

```
ENVAR("DB2_HOME=/usr/lpp/db2/db2g",
"JAVA_HOME=/usr/lpp/java/IBM/J1.3_V030510A",
"CLASSPATH=/SC63/sg247083/spjava",
"TMSUFFIX=/u/TrustedSP"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

The Java stored procedure can use the methods defined by the new Trusted Middleware classes the same way that they use methods defined by the Trusted Middleware known as the JDBC driver. They just use the new classes like any other classes. One restriction associated with this approach will be that the Java stored procedure can only be run in a WLM AE that has specified the TMSUFFIX referencing the needed trusted classes. This makes distributing the Java SP and set up a little more difficult than Java stored procedures that do not need the new Trusted Middleware, though it is not impossible.

Trusted classes cannot come from installed jars. They must come from the TMSUFFIX identified in the JAVAENV statement of the WLM AE.

For more information on using TMSUFFIX and *Trusted Middleware* classes in the Java Virtual Machine, see *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201, and the *New IBM Technology featuring Persistent Reusable Java Virtual Machines*, SC34-6034.

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. See *New IBM Technology Featuring Persistent Reusable Java Virtual Machines*, SC34-6034.

Example 17-8 shows the contents of the HFS file.

Example 17-8 Contents of 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 application class system heap
-Xinitacsh512K
#initial size of system heap
-Xinitsh512K
#initial size of transient heap
-Xinitth32M
```

For information about JVM startup options, see *New IBM Technology featuring Persistent Reusable Java Virtual Machines*, available at:

http://www.ibm.com/servers/eserver/zseries/software/java

17.2.8 Binding the JDBC packages

Submit job DSNTJJCL from <hlq>.SDSNSAMP to bind the four DSNJDBC packages. This job binds the packages into a collection named DSNJDBC and also binds a plan DSNJDBC. These packages are required to be bound in order to run both JDBC and SQLJ Java stored procedures making JDBC calls:

- DSNJDBC1 - must be bound with transaction isolation level = UR
- DSNJDBC2 - must be bound with transaction isolation level = CS
- DSNJDBC3 - must be bound with transaction isolation level = RS
- DSNJDBC4 - must be bound with transaction isolation level = RR

The default transaction level for the DSNJDBC plan is CS. To change the transaction level of a connection in a JDBC program, use the Connection.setTransactionIsolation method.

17.3 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 an 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 an 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, which is reset

between transactions or whenever necessary. 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. 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.

Java stored procedures have no awareness of transaction boundaries, the Java stored procedures environment drives JVM resets (primarily to spur *garbage collection)* periodically. By default this happens once every 256 procedure invocations.

17.4 Considerations on static variables

We advise against using static 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.

This advise 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.
- Static variables are reset whenever the JVM is driven through a reset cycle.
- It is difficult to guarantee that a sequence of CALLs will be processed by the same JVM.

  For instance:

  - CALL INITIALIZE, sets a static variable to a known state
  - CALL PROCESS, goes to a different JVM and finds the value of the static variable to be uninitialized

However, there are good, valid reasons that static variables be used, and we do not prevent a Java stored procedure or UDF from doing so. Two different techniques can be applied:

- Static variables associated with normal sharable application classes (which is what the classes associated with a Java SP or UDF are) are reset during each JVM reset. In this technique, any use of static variables must first test whether or not the variable is set to its default or uninitialized value, and if so, it initializes that variable and that newly initialized value will persist until the next JVM reset.
- Trusted Middleware classes can be declared with what JVM documentation refers to as “tidy-up” methods. In this technique, during JVM reset each Trusted Middleware class’s tidy-up method will be invoked, and the class both can be aware of the JVM reset and is not subject to the default re-initialization of its static variables. It is up to a Trusted Middleware class to manage its own static variables. You may or may not need such tidy-up methods in your application. It may be that simply avoiding the resetting of the

static variables is sufficient. But tidy-up methods give Trusted Middleware classes the opportunity to manage their own clean-up during the JVM reset process.

As a concluding cautionary note, with either of the above techniques, setting NUMTCBs to one is not guaranteed to solve the same-JVM problem, as multiple WLM address spaces can be initiated for a single WLMENV. So, the question of whether or not static variables can be used is a question specific to the application's design. With either technique it is recommended that testing be done with a lower than normal RESET_FREQ, but not a RESET_FREQ of one. Many problems that are likely to be encountered will only be found following JVM resets, but a frequency of one may not test the code as thoroughly as, say, a frequency of three.

17.5 Preparing Java stored procedures

In this section we discuss the most important steps in preparing the Java stored procedures for execution.

17.5.1 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 file for user ID PAOLOR7 would be /u/paolor7/.profile. Example 17-9 shows the contents of user profile data set /u/paolor7/.profile.

Example 17-9 /u/paolor7/.profile data set

```
PATH=/usr/lpp/java/IBM/J1.3_V03O510A/bin:$PATH
PATH=/usr/lpp/db2/db2g/bin:$PATH
export PATH
LIBPATH=/usr/lpp/db2/db2g/lib:$LIBPATH
export LIBPATH
LD_LIBRARY_PATH=/usr/lpp/db2/db2g/lib
export LD_LIBRARY_PATH
CLASSPATH=/usr/lpp/db2/db2g/classes/db2j2classes.zip:.
export CLASSPATH
STEPLIB=DB2G7.SDSNEXIT:DB2G7.SDSNLOAD:DB2G7.SDSNLOD2:$STEPLIB
STEPLIB=CEE.SCEERUN:$STEPLIB
export STEPLIB
export DB2SQLJPROPERTIES=/usr/lpp/db2/db2g/classes/db2sqljjdbc.properties
```

The names of the directory path could vary across installations. Refer to Table 17-3 for a general description of various environment variables. Please 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 run-time environment and properties. The JAVAENV data set controls the run-time environment and behavior of Java stored procedures.

Table 17-3 Environment variables

Variable	Typical value	Value used at the ITSO site
LIBPATH	/usr/lpp/db2/db2710/lib	/usr/lpp/db2/db2g/lib

Variable	Typical value	Value used at the ITSO site
CLASSPATH	/usr/lpp/db2/db2710/classes/db2j2classes.zip	/usr/lpp/db2/db2g/classes/db2j2classes.zip
LD_LIBRARY_PATH	/usr/lpp/db2/db2710/lib	/usr/lpp/db2/db2g/lib
PATH	It needs to be set to the bin sub directory in Java and DB2 directories /usr/lpp/db2/db2710/bin /usr/lpp/java/IBM/J1.3	/usr/lpp/db2/db2g/bin /usr/lpp/java/IBM /usr/lpp/java/IBM/J1.3_V030510A/bin

17.5.2 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 17-10 shows the foreground invocation of the **javac** command.

Example 17-10 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 the STDIN. Example 17-11 shows the JCL for compiling the Java program with AOPBATCH.

Example 17-11 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 accomodate a (default) 64 MB maximum heap size, the 40+ MB for the JIT, and leaves 24 MB for application and system storage requirements.

17.5.3 Preparing SQLJ stored procedures

See Figure 17-1 for 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 a part of its preparation process. All these steps are mentioned below.

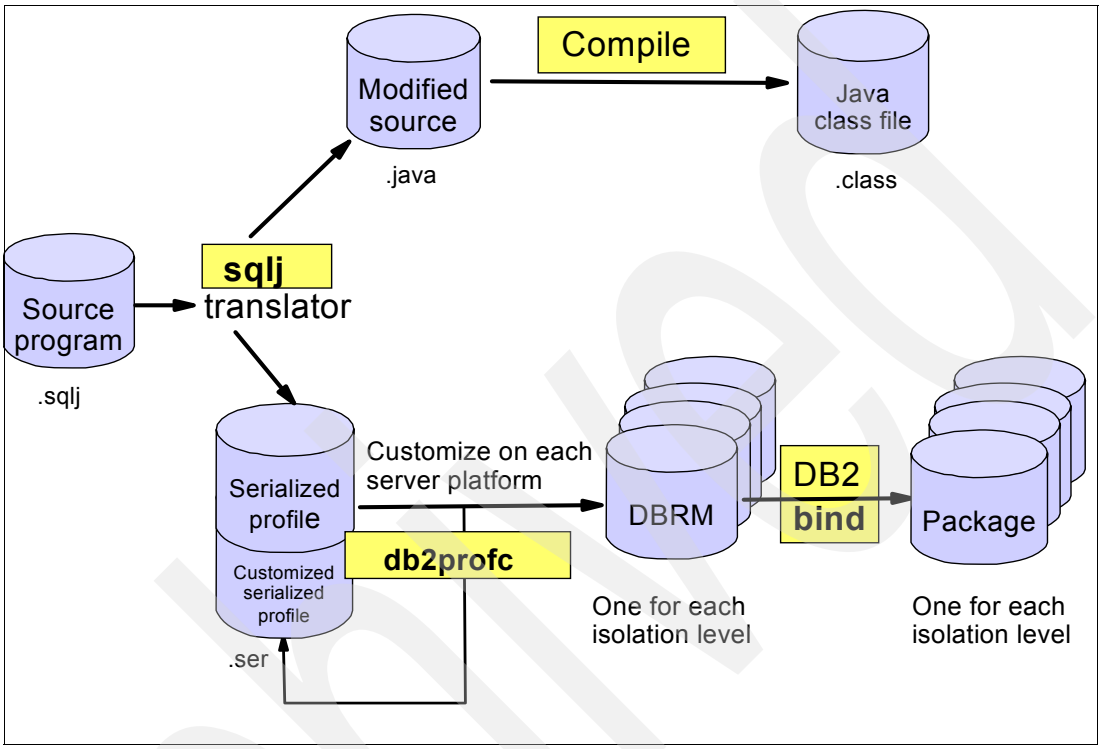


Figure 17-1 SQLJ preparation process

Translation and compilation

All SQLJ stored procedures 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
```

By issuing 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 17-12.

Example 17-12 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 needs to be customized further by using the db2proc command (discussed in the next section).
EmpDtl1J_SJProfileKeys.class	

Customizing the profile

Example 17-13 shows the command used for customizing the server profile. The **db2profc** command produces four DBRMs (one for each transaction isolation) and also updates the serialized profiles:

-online=<db2location name>, specifies that the SQLJ customizer connects to DB2 to do online checking of data types in the SQLJ program. location-name is the location name that corresponds to a DB2 subsystem to which the SQLJ customizer connects to do online checking

-pgmname=EMPDTL1, specifies the common part of the names for the four DBRMs that the SQLJ customizer generates. DBRM-name must be seven or fewer characters in length, and must conform to the rules for naming members of MVS partitioned data set.

Example 17-13 Sample db2profc command

```
/SC63/sg247083/spjava:>db2profc -online=DB2G -schema=DEVL7083 -pgmname=EMPDTL1
EmpDtl1J_SJProfile0.ser
```

Example 17-14 shows the output of the **db2profc** command. Notice the names of the four DBRM members that the **db2profc** command produces. Read the output for further instruction on binding the packages.

Example 17-14 Output of the db2profc command

```
-------------------------------------------------------
-> DB2 7.1: Begin Customization of SQLJ Profile
-------------------------------------------------------
-> SQLJ profile name is: EmpDtl1J_SJProfile0

-> Number of sections is:       1
-> Number of HOLD   cursors is: 0
-> Number of NOHOLD cursors is: 1
-------------------------------------------------------
-> Profile <<EmpDtl1J_SJProfile0.ser>> has been customized for DB2 for z/OS
-> This profile must be present at runtime
-------------------------------------------------------

 ->INFORMATIVE<-
 ->INFORMATIVE<-    ***************** IMPORTANT *****************
```

```
 ->INFORMATIVE<- -> The following DBRMs must be bound as specified into the
 ->INFORMATIVE<- -> packages with the associated Transaction Isolation:
 ===>
 ->INFORMATIVE<- -> packages with the associated Transaction Isolation:
 ->INFORMATIVE<- ->  Bind DBRM in SG247083.DEVL.DBRM(EMPDTL11)
 ->INFORMATIVE<- ->  into Package EMPDTL11 with Transaction Isolation UR
 ->INFORMATIVE<- ->  Bind DBRM in SG247083.DEVL.DBRM(EMPDTL12)
 ->INFORMATIVE<- ->  into Package EMPDTL12 with Transaction Isolation CS
 ->INFORMATIVE<- ->  Bind DBRM in SG247083.DEVL.DBRM(EMPDTL13)
 ->INFORMATIVE<- ->  into Package EMPDTL13 with Transaction Isolation RS
 ->INFORMATIVE<- ->  Bind DBRM in SG247083.DEVL.DBRM(EMPDTL14)
 ->INFORMATIVE<- ->  into Package EMPDTL14 with Transaction Isolation RR
 ->INFORMATIVE<-
 ->INFORMATIVE<- -> These packages must then be bound into the plan to be used f
or SQLJ applications.
 ->INFORMATIVE<- -> This plan must be specified at run time via the properties f
ile
 ->INFORMATIVE<-    ***************** IMPORTANT *****************

------------------------------------------------------
-> DB2 7.1: End   Customization of SQLJ Profile
------------------------------------------------------
/SC63/sg247083/spjava:>
 ===>
                                                                    RUNNING
```

Binding the DBRMs into packages

In Example 17-15 we show the sample job to bind the packages into the collection DEVL7083. You need to specify the collection name in the DDL definition of the stored procedure. You also need to be aware that the collection must also have the four DSNJDBC packages supplied with DB2 as 17.2.8, “Binding the JDBC packages” on page 253.

Example 17-15 Binding the DBRM packages for SQLJ stored procedure.

```
//BINDPKG EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(0,LT)
//DBRMLIB  DD DSN=SG247083.DEVL.DBRM,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//REPORT   DD SYSOUT=*
//SYSTSIN DD *
 DSN SYSTEM(DB2G)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL11) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(UR) ACT(REP)  RELEASE(COMMIT)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL12) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(CS) ACT(REP)  RELEASE(COMMIT)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL13) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(RS) ACT(REP)  RELEASE(COMMIT)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL14) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(RR) ACT(REP)  RELEASE(COMMIT)
END
//*
```

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 17-16. Make sure that the user who submits the job has the appropriate

profile data set as discussed in 17.5.1, “Profile data set” on page 255. Notice a back slash in the **db2profc** command. It is used as a continuation character for the **db2profc** command.

Example 17-16 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
  sqlj EmpDtl1J.sqlj
db2profc -online=DB2G -schema=DEVL7083  \
-pgmname=EMPDTL1 EmpDtl1J_SJProfile0.ser
/*
//BINDPKG EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(0,LT)
//DBRMLIB  DD DSN=SG247083.DEVL.DBRM,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//REPORT   DD SYSOUT=*
//SYSTSIN DD *
 DSN SYSTEM(DB2G)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL11) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(UR) ACT(REP)  RELEASE(COMMIT)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL12) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(CS) ACT(REP)  RELEASE(COMMIT)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL13) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(RS) ACT(REP)  RELEASE(COMMIT)
BIND PACKAGE(DEVL7083) MEMBER(EMPDTL14) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(RR) ACT(REP)  RELEASE(COMMIT)
END
//*
```

Making the stored procedure class files available to DB2

We differentiate between with and without jars.

Without jars

If you are not using jars, all you need to do is to place the stored procedure class files in the CLASSPATH directory. In case of SQLJ stored procedures, you also need to place the .ser files, context classes in the CLASSPATH directory.

Table 17-4 shows the relationship between classpath and the location of the class files. It has two examples, one each for JDBC and SQLJ stored procedure.

Table 17-4 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

JDBC Stored Procedure: EMPDTLSJ	
CLASSPATH (In the JAVAENV data set)	/SC63/sg247083/spjava
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 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 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.".

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.INSTALL_JAR. If you define the jar to DB2, you should not have the class files in the CLASSPATH. In case the class files appear at both places, DB2 jar and the CLASSPATH directory, you may get unpredictable results.

In case you do not define the jar file to DB2, then you could mention the jar file in the CLASSPATH.

Example 17-17 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.

Example 17-17 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:>
```

Defining jars to DB2

Once you have created the jar file, you need to define it to DB2. DB2 provides you with the INSTALL_JAR stored procedure to load the jar file to DB2. You could invoke the INSTALL_JAR stored procedure from the Development Center (DC), the Stored Procedure Builder (SPB), or from an application program:

```
CALL SQLJ.INSTALL_JAR(file:/SC63/sg247083/spjava/Employee.jar,DEVL7083.EMPDTL,0)
```

The first argument, known as the *url*, specifies the fully qualified path name for the jar file.

The second argument is the *SCHEMANAME.JARNAME*. The JARNAME that you specify can be anything.

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.

17.6 DDL for defining a Java stored procedure

Example 17-18 shows a sample DDL definition used for defining the JDBC-based Java stored procedure.

Example 17-18 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 DB2GWEJ1
```

Refer to Table 17-5, for a detailed discussion on the various options while defining a DB2 stored procedure.

Table 17-5 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 on the parameters can be found in 17.6.1, “INPUT/OUTPUT parameters” on page 263
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 17.6.2, “EXTERNAL NAME” on page 264 for further details.
PROGRAM TYPE	**SUB**. For Java stored procedures PROGRAM TYPE should always be SUB.
WLM ENVIRONMENT	Java stored procedures can only run in a WLM Environment. We used a WLM Environment DB2GWEJ1. Steps for defining the environment can be found in Chapter 4, “Setting up and managing Workload Manager” on page 33**.**

Note: Using 1.4.1 JVM requires adding XPLINK(ON) to JAVAENV and APAR PQ76769.

17.6.1 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. 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. In Table 17-6, FIRSTNAME is defined as an output variable, and the Java program declares it as a String[] array. Also, ensure that the Java method is defined as public, static, and void.

Table 17-6 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 17-7 shows a stored returning a result set.

Table 17-7 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 DB2GWEJ1	public static void GetEmpResult((String workDept, String[] outputMessage, **ResultSet[] rs**)

17.6.2 EXTERNAL NAME

The external name specifies the location and the 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 below:

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/.

```
EXTERNAL NAME 'EmpDtlsJ.GetEmpDtls'
```

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

Java stored procedure EmpDtlsJ.java has been compiled into EmpDtlsJ.class. There is a package statement `package itso.ibm` specified in the first line of 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/.

EXTERNAL NAME ‘itso.ibm.EmpDtlsJ.GetEmpDtls’

LOCATION for EmpDtlsJ.class file should be in a directory named /SC63/sg247083/spjava/itso/ibm/.

Notice that we need to create the subdirectories 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

Java stored procedure EmpDtlsJ.java has been compiled into EmpDtlsJ.class. The EmpDtlsJ.class resides in a directory /SC63/abc/pqr/xyz/. A package statement `package abc.pqr.xyz` is specified in the first line of the Java code. A jar file Employee.jar is created to hold the EmpDtlsJ.class. by issuing the commands shown in Example 17-19.

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 run-time, 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 17-19 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 EmpDtlsJ.java has been compiled into EmpDtlsJ.class. The EmpDtlsJ.class resides in a directory /SC63/abc/pqr/xyz/. A package statement `package abc.pqr.xyz` is specified in the first line of the Java code. A jar file Employee.jar is created to hold the EmpDtlsJ.class. by issuing the commands shown in Example 17-19. We registered the Employee.jar file to DB2 using the IBM supplied stored procedure SQLJ.INSTALL_JAR. 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.

The external name clause should be defined as follows:

```
EXTERNAL NAME 'DEVL7083.EMPLJAR:abc.pqr.xyz.EmpDtlsJ.GetEmpDtls'
```

CLASSPATH variable can be set to blank.

17.7 Debugging JDBC and SQLJ

Presently there is no direct way for debugging stored procedures on the host. You are required to convert the Java stored procedure into a Java application and then use a workstation tool such as WebSphere Studio Application Developer (WSAD) to debug the Java application from a Windows platform. The steps of converting a Java stored procedure to a Java application with a minimal effort is documented in 17.7.1, “Changing Java stored procedure to enable debugging in WSAD” on page 266, which shows the procedure to debug a Java application using WSAD.

In this section we show how to change your Java stored procedure to enable debugging using WSAD, and how to code System.out.println and System.err.printl to document errors on z/OS.

17.7.1 Changing Java stored procedure to enable debugging in WSAD

Table 17-8 shows the conversion.

Table 17-8 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 WSAD or any Java Development tool.
	The Java application can be invoked from the command line:
	
	Java EmpDtlsJ ‘000010’

Changes to the connection String	

Stored procedure code	Converted Java application
Changes to the Stored Procedure Method	
Connection conndb2 = null; conndb2 = DriverManager.getConnection("jdbc:default:connection");	The connection statements in the stored procedure needs to be changed, depending on where the Java application needs to run. In either case it access data from DB2 for OS/390 and 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 code needs to be imported into WSAD. Instruction for debugging a Java JDBC or SQLJ application can be found in Chapter 30, “Using WSAD to debug Java stored procedures converted to Java applications” on page 553.

17.7.2 Debugging Java stored procedures on z/OS

You can code System.out.println and System.err.println lines in your Java code. The output is directed to STDOUT and STDERR DD cards in WLM address space.

Example 17-20 shows the DD cards that you need to include in your WLM address space.

Example 17-20 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.

17.8 Java sample JDBC stored procedures

In this section we show how to implement JDBC routines.

17.8.1 Sample Java stored procedure code: EmpDtlsJ using JDBC

The sample Java stored procedure shown in Example 17-21 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 17-21 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();
        }
    }
}
```

17.8.2 DDL for Java stored procedure EmpDtlsJ

Example 17-22 shows the DDL for creating the EMPDTLSJ stored procedure.

Example 17-22 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
```

17.8.3 Deploying JDBC stored procedures on z/OS

The way you deploy your Java stored procedure 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 17-23.

Example 17-23 FTP the Java source code

```
C:\>ftp wtsc63.itso.ibm.com
Connected to wtsc63.itso.ibm.com.
220-FTPMVS1 IBM FTP CS V1R4 at wtsc63oe.itso.ibm.com, 01:51:14 on 2003-11-08.
220 Connection will close if idle for more than 5 minutes.
User (wtsc63.itso.ibm.com:(none)): paolor7
331 Send password please.
Password:
230 PAOLOR7 is logged on. Working directory is "PAOLOR7.".
ftp> cd /SC63/sg247083/spjava
250 HFS directory /SC63/sg247083/spjava is the current working directory
```

```
ftp> put EmpDtlsJ.java
200 Port request OK.
125 Storing data set /SC63/sg247083/spjava/EmpDtlsJ.java
250 Transfer completed successfully.
ftp: 1525 bytes sent in 0.01Seconds 152.50Kbytes/sec.
ftp>
```

17.8.4 Sample Java stored procedure returning a result set - EmpRsetJ

Example 17-24 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 17-24 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 DB2GWEJ1
```

Example 17-25 shows the code for Java stored procedure EmpRsetJ. Notice that a result set argument is included in the method signature for GetEmpResult.

Example 17-25 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();
```

```
}
```

17.9 Java sample SQLJ stored procedures

In this section we show how to implement SQLJ routines.

17.9.1 Sample code for SQLJ stored procedure - EmpDtl1J.sqlj

Example 17-28 shows an SQLJ stored procedure EmpDtl1J.sqlj. In the current section we discuss the 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 mentioned below 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 a context class EmpDtl1J_Ctx. At the time of compilation the sqlj translator creates a new Java class by the name of 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 a 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 any other 3-GL language like COBOL, C etc. You need to declare your host variables before you can use them in an SQL statement. Example 17-26 shows the host variable declarations that was used in the example.

Example 17-26 Host variable declaration

```
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 variables. Example 17-27 shows a sample SQL statement that was used.

Example 17-27 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 17-28 shows the sample code for SQLJ stored procedure.

Example 17-28 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();
          }
    }
```

```
}
```

17.9.2 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. A result set iterator 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 17-30 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, then 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 17-29 shows the external file that contains the iterator declaration, EmpTst2J_UpdByPos. The iterator specifies that you intend to update the SALARY column.

Example 17-29 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 illustrates the use of a positioned update.

Example 17-30 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 17-31 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 17-31 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
db2profc -online=DB2G -schema=DEVL7083 -pgmname=EMPRST2 EmpRst2J_SJProfile0.ser
/*
//*-------------------------------------------------------
//*-------------------------------------------------------
//BINDPKG EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(0,LT)
//DBRMLIB  DD DSN=SG247083.DEVL.DBRM,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//REPORT   DD SYSOUT=*
//SYSTSIN DD *
 DSN SYSTEM(DB2G)
 BIND PACKAGE(DEVL7083) MEMBER(EMPRST21) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(UR) ACT(REP)  RELEASE(COMMIT)
 BIND PACKAGE(DEVL7083) MEMBER(EMPRST22) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(CS) ACT(REP)  RELEASE(COMMIT)
 BIND PACKAGE(DEVL7083) MEMBER(EMPRST23) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(RS) ACT(REP)  RELEASE(COMMIT)
 BIND PACKAGE(DEVL7083) MEMBER(EMPRST24) VALIDATE(BIND) -
    OWNER(DEVL7083) ISOLATION(RR) ACT(REP)  RELEASE(COMMIT)
 END
//*
```

DDL definition for EMPRST2J

Example 17-32 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
```

18

Using the new Universal Driver

Before you read this chapter, it is recommended that you read the previous, and get familiar with the setup requirements for Java stored procedures. For reference information see *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java,* SC18-7414.

In this chapter we describe the usage of the Universal Driver for Java. This driver has been announced with DB2 V8, but it is also made available to DB2 V7 through APAR PQ80841.

This chapter contains the following:

- JCC setup for DB2 stored procedures
- SQLJ preparation process using the new JCC
- Migrating stored procedures to use the new JCC driver

18.1 JCC - Universal Driver

The DB2 Universal JDBC Driver is a new JDBC and SQLJ driver implementation that is being provided on multiple DB2 platforms and that can execute as a locally-connected (Type 2) driver, or as a standalone, remote client (Type 4) driver. JDBC drivers of various types have previously been provided with DB2 on each of the supported hardware platforms. The Universal JDBC Driver will consolidate all of those drivers into a single common implementation.

DB2 for OS/390 and z/OS has provided a JDBC driver as an optional feature (under FMID JDB8812) for the past several releases. That driver provided both a JDBC 1.2 and a JDBC 2.0-based implementation. In DB2 Version 8, the JDBC 2.0-based implementation will continue to be provided and supported. That driver is referred to in this book as the legacy JDBC driver.

The Universal JDBC Driver will be provided under the same FMID JDB8812, but will reside in a different directory. It provides JDBC 2.0 and most JDBC 3.0 functionality. Do not confuse with the *JDBC Version* and *type number* for the *driver*.

The legacy type 2 driver can be found in /usr/lpp/db2/db2810. All existing Java stored procedures (JDBC and SQLJ) can continue to run without any change using the Legacy Driver.

The new Universal Driver (JCC) can be found in /usr/lpp/db2/db2810/jcc. All the files and directories corresponding to the Universal Driver are placed under a sub directory named jcc.

The Legacy Driver is a Type 2 implementation while the JCC driver implements a Type 2 and a Type 4 driver. While using the JCC driver, the connection string specified in the Java code determines the appropriate driver that is used. Please be aware that the Type 2 implementation of a JCC driver is different than the Type 2 implementation of a Legacy JDBC driver provided in DB2 V8 and previous DB2 releases. DB2 V8 provides both the drivers: Legacy Type 2 and JCC Type 4, and Type 2.

18.2 JCC setup for DB2 stored procedures

Java stored procedures running on S/390® can use the Type 2 implementation of the JCC driver or the Type 2 implementation of the Legacy Driver. They cannot use the JCC Type 4 driver. The connection string along with JAVAENV properties file determines the driver that is being used.

To summarize the above discussion, please refer to Table 18-1.

Table 18-1 JDBC Driver Types and DB2 versions

Feature	DB2 V7	DB2 V8 - Legacy Driver	DB2 V8 - JCC driver (Universal)
JDBC Driver Type	Type 2	Type 2	Type 4 and Type 2
JDBC Driver Version	JDBC 1.2 and JDBC 2	JDBC 1.2 and JDBC 2	JDBC 2.0 and JDBC 3.0
location of the driver	/usr/lpp/db2/db2710	/usr/lpp/db2/db2810	/usr/lpp/db2/db2810/jcc

Feature	DB2 V7	DB2 V8 - Legacy Driver	DB2 V8 - JCC driver (Universal)
JAVAENV Setup	DB2_HOME=/usr/lpp/db2/db2710	DB2_HOME=/usr/lpp/db2/db2810	DB2 HOME no longer used. JCC_HOME=/usr/lpp/db2/db2810/jcc.
Connection String in a Java Stored Procedure	Connection condb = DriverManager.getConnection("jdbc:default:connection"); Connection condb = DriverManager.getConnection("jdbc:db2os390:*location-name*"); Connection condb = DriverManager.getConnection("jdbc:db2os390sqlj:*location-name*");	Connection condb = DriverManager.getConnection("jdbc:default:connection"); Connection condb = DriverManager.getConnection("jdbc:db2os390:*location-name*"); Connection condb = DriverManager.getConnection("jdbc:db2os390sqlj:*location-name*");	Connection condb = DriverManager.getConnection("jdbc:default:connection"); Connection condb = DriverManager.getConnection("jdbc:db2:*location-name*"); Please note: Stored Procedure only supports Type 2 implementation of JCC driver.You cannot use a Type 4 implementation Also note the changes in the connection string.

18.2.1 JAVAENV for DB2 stored procedures

In DB2 V8, you have a choice while running your stored procedures. You can either use the existing Legacy Driver (type 2) or use the new type 2 driver that comes with the Universal Driver (JCC). You require to have separate WLM application environments for each of the above case.

The contents of the JAVAENV file decide if you are running the new JCC driver or using the Legacy Driver. If you want to use the JCC driver, you need to set the JCC_HOME variable to the location of the JCC driver. In case you want to use the Legacy Driver you need to have DB2_HOME variable set to the location of Legacy Driver. You should not have the definitions of JCC_HOME and DB2_HOME in the same JAVAENV file.

In case you want to use the Legacy Driver, the setup and customization is the same as that described in Chapter 18, “Using the new Universal Driver” on page 277.

In case you want to use the new JCC driver, then you need to follow the steps mentioned in this chapter.

In our environment we created two WLM application environments. DB8ADJC2 uses the JCC driver, and DB8ADJ1 uses the Legacy Driver. Example 18-1 shows the WLM procedure for the JCC driver. We set NUMTCB to 1 for our Java development WLM environment, since during development we needed to refresh the WLM environment regularly to have our changes take affect. The NUMTCB is set between 5-7 for the production WLM environment for our Java stored procedures.

Example 18-1 WLM procedure for JCC driver

```
//DB8ADJC2 PROC APPLENV=DB8ADJC2,DB2SSN=DB8A,NUMTCB=1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=0M,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD  DISP=SHR,DSN=DB8A8.SDSNEXIT
```

```
//         DD  DISP=SHR,DSN=DB8A8.SDSNLOAD
//         DD  DISP=SHR,DSN=DB8A8.SDSNLOD2
//*        DD  DISP=SHR,DSN=CEE.SCEERUN
//*   NEED UNAUTHORIZED DATASET
//         DD  DISP=SHR,DSN=CBC.SCBCCMP
//JAVAENV  DD  DSN=DB8AU.JCC2.JAVAENV,DISP=SHR
//JSPDEBUG DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
```

The contents of JAVAENV are shown in Example 18-2. Notice that most of the parameters are mentioned in an HFS file `/SC63/sg247083/DB8AU/jcc2_nolimit.txt`. By following this approach your environment file does not cross 245 bytes.

Example 18-2 Contents of JAVAENV

```
ENVAR("_CEE_ENVFILE=/SC63/sg247083/DB8AU/jcc2_nolimit.txt",
"JAVA_HOME=/usr/lpp/java/IBM/J1.3_V030510A"),
MSGFILE(JSPDEBUG,,,,ENQ)
```

The contents of the HFS file `/SC63/sg247083/DB8AU/jcc2_nolimit.txt` are shown in Example 18-3. Notice that there is no mention of DB2_HOME. JCC_HOME refers to the JCC directory.

Example 18-3 Contents of HFS file

```
CLASSPATH=/SC63/sg247083/DB8AU/jcc/spjava
JCC_HOME=/usr/lpp/db2/db8a/jcc
WORK_DIR=/SC63/sg247083/DB8AU/tmp
```

18.2.2 USS profile data set

You need to have the following statements included into the /etc/profile file or in the .profile file of the user that invokes the SQLJ or binder utilities. Please be aware that the environment variable DB2JCCPROPERTIES does not have much significance in a stored procedure environment. Notice that all the libraries corresponding to the Universal Driver are located under a subdirectory /jcc. See Example 18-4.

Example 18-4 Contents of the profile data set

```
PATH=/usr/lpp/java/IBM/J1.3_V030510A/bin:$PATH
PATH=/usr/lpp/db2/db8a/jcc/bin:$PATH
export PATH
LIBPATH=/usr/lpp/db2/db8a/jcc/lib:$LIBPATH
export LIBPATH
LD_LIBRARY_PATH=/usr/lpp/db2/db8a/jcc/lib
export LD_LIBRARY_PATH
CLASSPATH=/usr/lpp/db2/db8a/jcc/classes/db2jcc.jar:$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:$CLASSPATH
export CLASSPATH
STEPLIB=DB8A8.SDSNEXIT:DB8A8.SDSNLOAD:DB8A8.SDSNLOD2:$STEPLIB
STEPLIB=CEE.SCEERUN:$STEPLIB
export STEPLIB
```

18.2.3 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 *DB2 Installation Guide* for information on setting DESCSTAT. This step is necessary for SQLJ support.

18.2.4 Binding the packages for Universal JDBC Driver

To bind the packages for the DB2 Universal JDBC Driver, run the DB2binder utility. This utility binds the packages and grants EXECUTE authority on the packages to PUBLIC.

Universal Driver requires a set of packages to be bound at the target DB2 system. From the USS shell you need to run the IBM provided binder utility.

Before running the utility, ensure that you have the file /usr/lpp/db2810/jcc/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:12345/DB8A
-user PAOLOR7 -password BHAS11
-collection DEVL7083
```

The URL options are: -url jdbc:db2://server_name:port_number/database_name.

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 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 command needs to be the same ID under which your stored procedure will execute. You need to issue this command for every collection. The command can also be issued from the PC through a DOS prompt.

Example 18-5 shows an alternate way to bind the JDBC packages.

Example 18-5 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:12345/DB8A  \
-user PAOLOR7 -password BHAS11 \
-collection DEVL7083
/*
```

18.2.5 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:

```
SQLCOLPRIVILEGES
SQLCOLUMNS
SQLFOREIGNKEYS
SQLGETTYPEINFO
SQLPRIMARYKEYS
SQLPROCEDURECOLS
SQLPROCEDURES
SQLSPECIALCOLUMNS
SQLSTATISTICS
SQLTABLEPRIVILEGES
SQLTABLES
SQLUDTS
SQLCAMESSAGE
```

These procedures are provided in DB2 for z/OS Version 8. They are created by installation job DSNTIJSG when created 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 Part 2 of the *V8 DB2 Installation Guide* for more information.

18.3 SQLJ preparation process using the new JCC

To prepare an SQLJ application to run in a JVM, and with the Universal Driver follow these steps:

1. Translate the source code to produce generated Java source code and serialized profiles, and compile the generated source code to product Java byte codes.
2. Customize the serialized profiles to produce customized serialized profiles and DB2 packages. Instead of using the **db2profc** command (for Legacy Driver) you need to use the new **db2sqlcustomize** command.

   In Figure 18-1 we show the preparation process.

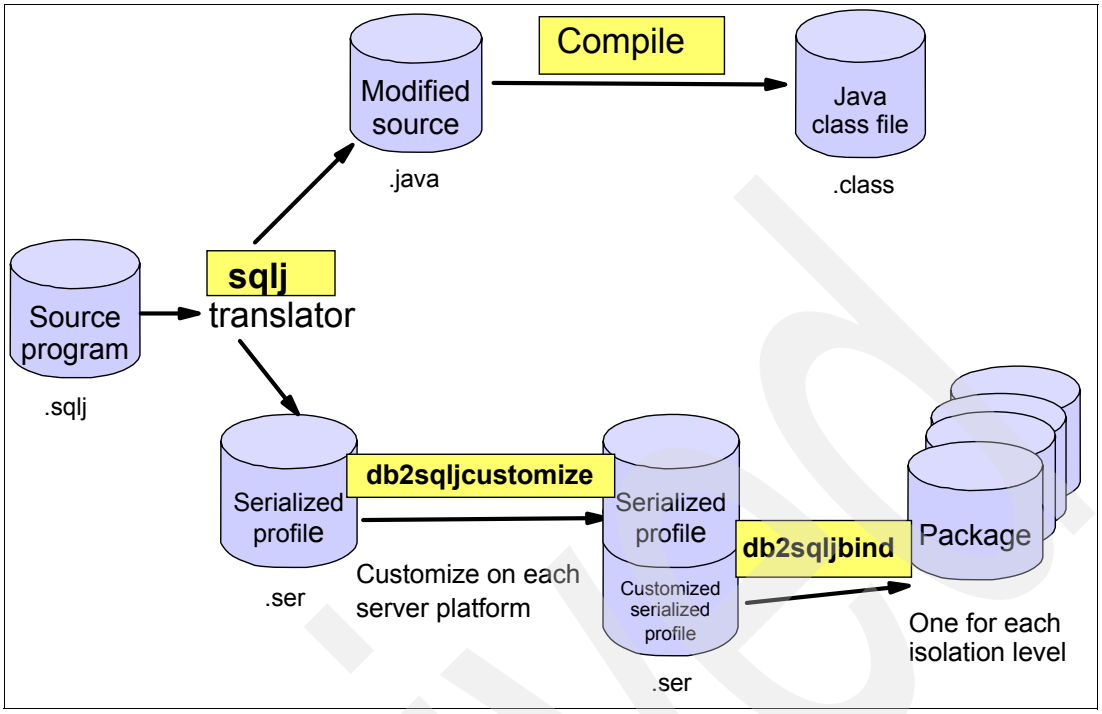


Figure 18-1 Shows the SQLJ preparation process with the new driver

Translating and compiling

On the UNIX prompt issue the commands shown in Figure 18-6.

Example 18-6 Translating and compiling the sqlj stored procedure

```
/SC63/sg247083/DB8AU/jcc/spjava:>sqlj EmpDtl1J.sqlj

after the completion of the command you should have the following files:
EmpDtl1J.class                EmpDtl1J_Ctx.class
EmpDtl1J.java                 EmpDtl1J_SJProfile0.ser
EmpDtl1J.sqlj                 EmpDtl1J_SJProfileKeys.class
```

Customize the profile and bind the packages

After you use the SQLJ translator to generate serialized profiles for an SQLJ program, you need to customize each serialized profile. The command that you use, and the output that you receive depends on the driver that you use. In this chapter we deal with the Universal Driver. We created a script file as shown in Example 18-7 and executed the script file.

Example 18-7 Script file to prepare an sqlj application

```
sqlj EmpDtl1J.sqlj
db2sqljcustomize  -url jdbc:db2://wtsc63.itso.ibm.com:12345/DB8A \
-user PAOLOR7 -password BHAS11 -automaticbind YES \
-bindoptions " QUALIFIER(DEVL7083) RELEASE(COMMIT) " \
-rootpkgname EMPDTL1 \
-collection DEVL7083  -onlinecheck YES -qualifier DEVL7083  \
EmpDtl1J_SJProfile0.ser
```

Commands for executing the script file is shown in Example 18-8. The **db2sqljcustomize** utility customizes the serialized profiles and binds the packages directly into the DB2 subsystem.

If you set the automatic bind property as yes, the customizer utility binds the packages, otherwise you need to run the binder as a separate extra step.

The root package name should be less than or equal to seven characters in length; the customizer creates four packages, and puts a suffix number to the package name. In our case it created EMPDTL1, EMPDTL2, EMPDTL3, and EMPDTL4 packages, with one for each isolation level.

Example 18-8 Executing the script file sqljcomp.sj

```
/:>cd /SC63/sg247083/DB8AU/jcc/spjava
/SC63/sg247083/DB8AU/jcc/spjava:>sqljcomp.sh
```

Using a batch job for preparation

The batch job shown in Example 18-9 prepares the sqlj stored procedure EmpDtl1J.sqlj.

Example 18-9 Job for preparing EmpDtl1J.sqlj stored procedure

```
//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/DB8AU/jcc/spjava
  sqlj EmpDtl1J.sqlj
  db2sqljcustomize  -url jdbc:db2://wtsc63.itso.ibm.com:12345/DB8A \
  -user PAOLOR7 -password BHAS11 -automaticbind YES \
  -bindoptions " QUALIFIER(DEVL7083) RELEASE(COMMIT) " \
  -rootpkgname EMPDTL1 \
  -collection DEVL7083  -onlinecheck YES -qualifier DEVL7083  \
  EmpDtl1J_SJProfile0.ser
//* The backslash character is used as a command continuation character.
```

Creating the jar files and defining to DB2

The steps for creating the jar file and defining it to DB2 are the same as those explained in Chapter 18, “Using the new Universal Driver” on page 277. There is no change while using the new driver.

18.4 Migrating stored procedures to use the new JCC driver

Assuming that you currently have Java stored procedures running on your 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 mentioned below.

Presently, the JCC driver is only shipped with DB2 V8; IBM will be shipping the JCC driver for DB2 Version 7 customers in APAR PQ80841. In order to use it with Java routine processing, PQ76769 is also required.

We explain the steps that you need to take for migrating your stored procedures that currently use the old legacy SQLJ/JDBC driver. We assume that you are currently on DB2 V8 and

running your stored procedure with the Legacy Driver. The same procedure can also be used if you were running your stored procedures in DB2 V7, and wish to migrate to JCC driver in DB2 V7 (once made available).

18.4.1 Migrating JDBC stored procedures

The steps are:

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 the 18.2.1, “JAVAENV for DB2 stored procedures” on page 279 for further details on setting up the WLM proc 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 18-2 shows the various setup steps that you need to perform to move to the JCC driver.

Table 18-2 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/J1.3_V030510 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/J1.3 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/J1.3_V030510 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/J1.3 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

	DB2 V8 using Legacy Driver	DB2 V8 using Universal Driver (JCC)
DDL	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 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
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 run-time 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/J1.3_V030510A PATH=/usr/lpp/java/IBM/J1.3_V030510A/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:$STEPLIB export STEPLIB export DB2SQLJPROPERTIES=/SC63/sg247083/DB8AU/db2sqljjdbc.properties Notice that there is no mention of the jcc directories	JAVA_HOME=/usr/lpp/java/IBM/J1.3_V030510A PATH=/usr/lpp/java/IBM/J1.3_V030510A/bin:$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

18.4.2 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 executable 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 the 18.2.1, “JAVAENV for DB2 stored procedures” on page 279 for further details on setting up the WLM proc 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 18-10 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 18-12 on page 289.

Example 18-10 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/J1.3_V03051OA/bin.....................................5.
  PATH=/usr/lpp/db2/db8a/jcc/bin:$PATH  .......................................6
  cd /SC63/sg247083/DB8AU/jcc/spjava/..........................................7
  db2sqljupgrade EmpDtl1J_SJProfile0.ser.......................................8.
/*
```

Example 18-11 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 18-11 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 18-3 summarizes the changes for migrating to the JCC driver.

Table 18-3 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/J1.3_V030510 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/J1.3 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/J1.3_V030510 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/J1.3 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 18-10.
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 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 run-time 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/J1.3_V030510A PATH=/usr/lpp/java/IBM/J1.3_V030510A/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:$STEPLIB export STEPLIB export DB2SQLJPROPERTIES=/SC63/sg247083/DB8AU/d2sqljjdbc.properties Notice that there is no mention of the jcc directories	JAVA_HOME=/usr/lpp/java/IBM/J1.3_V030510A PATH=/usr/lpp/java/IBM/J1.3_V030510A/bin:$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 upmessage shown in Example 18-12 on page 289

Example 18-12 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
```

18.4.3 Extracting a .ser file from a jar file defined to DB2

There are times when the sqlj run time files are packaged into a jar file. Let us assume that we have a stored procedure EmpRst1J.sqlj

1. The run-time files for the above stored procedure 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 18-13 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 the third the output file.

Example 18-13 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 18-14 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 can be found in 26.4.4, “Handling large BLOB columns” on page 445.

The Java application shown in Example 18-13 used a JCC Type 4 Driver. Notice the connection string; we specify the domain name and port number of target DB2 subsystem.

Example 18-14 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. Next step is to upgrade the EmpRst1J_SJProfile0.ser using the db2sqljupgrade utility. Example 18-10 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 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.

Part 5

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 295

  An introduction to performance characteristics, tools, and capacity planning of DB2 stored procedures
- Chapter 20, “Server address space management” on page 319

  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 331

  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 procedures 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 the client through the SQL CALL statement. These are some of the advantages of using stored procedures:

- In a distributed environment, performance often can 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 is only available with WLM managed address spaces. Figure 19-1 shows the types of address spaces that are typically active when stored procedures are being executed.

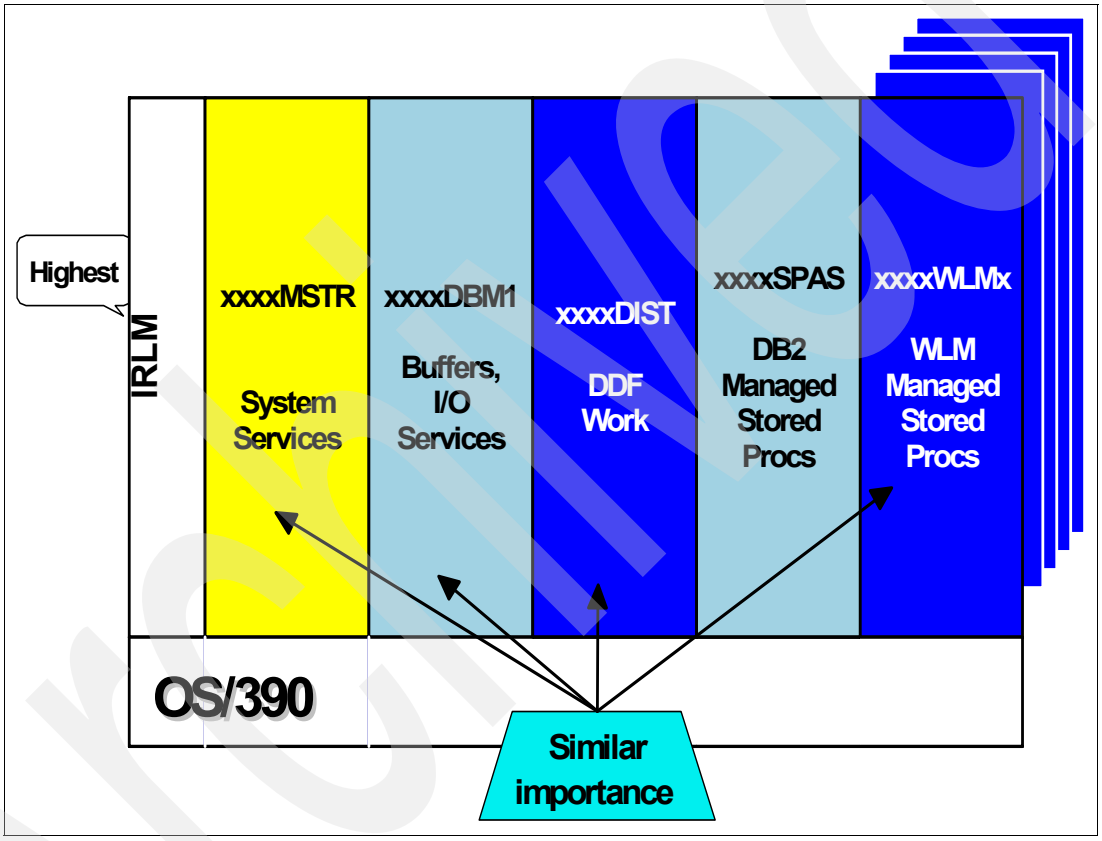


Figure 19-1 The DB2 address spaces with stored procedures

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 a similar priority, just below the IRLM address space priority. 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.

Next, the single DB2-managed stored procedures address space (SPAS) should be given a goal/priority typical of applications. All stored procedures that run in this address space will all have the same priority and could be higher or lower than the calling application's goal or priority.

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 323 for more information on classifying your workloads.

Stored procedures that run in the DB2-established stored procedure address spaces (DB2 SPAS) run at the priority established for the DB2 SPAS. Therefore, you should set the priority of the DB2 SPAS similarly to the priority of the calling application. You can run into performance problems if you set the priority of DB2 SPAS at the same level as some other address space, and then you call a stored procedure from an application running in a different address space that has a different priority. Since all stored procedures that run in the DB2 SPAS have to share the same priority, you do not have the flexibility to manage your workload according to your desired priority for each task. This is one of the main reasons that we recommend you run all your stored procedures in WLM-managed address spaces.

Note: In this chapter we primarily discuss performance considerations for WLM-managed stored procedures. DB2-managed stored procedure address spaces are only mentioned because you may still have some existing stored procedures running in the DB2-managed address space. You should consider migrating those stored procedures to WLM-managed address spaces as soon as possible. DB2 V8 only allows the creation of goal mode WLM-managed stored procedures.

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.

Assuming an application 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 319 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.

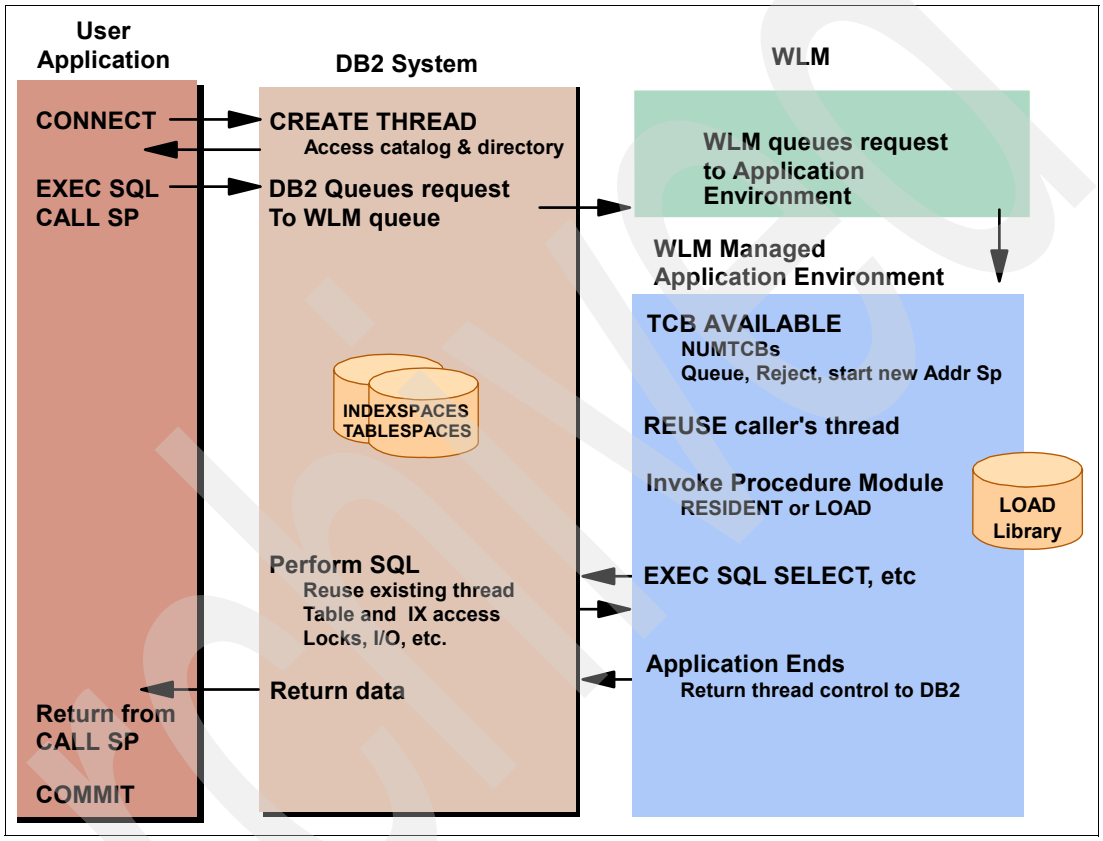


Figure 19-2 Stored procedure application life cycle

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 run-time 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 application 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.

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.

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

There are many components that 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.

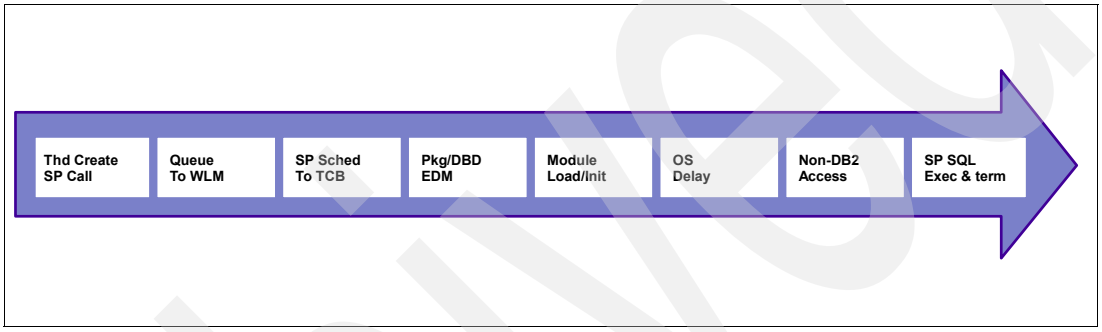


Figure 19-3 Where the execution time goes

Here is a breakdown of each execution time component:

- Thread create on stored procedure call

  This is handled differently depending on whether the stored procedure is called locally or from a distributed client. If the stored procedure is called locally, it runs under the same thread as the calling program. If the stored procedure is called from a distributed client, then there is some wait time for the thread to be created, unless the client already had connected to DB2; in that case, the stored procedure runs under the same thread as the client.

- 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 304 for more details on DB2 accounting time for stored procedures.

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.

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, thread reuse and release deallocate = 80 µs 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 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 depend 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.

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, which you can use to monitor and measure a stored procedure's performance. It is important to be current on maintenance. Table 19-1 shows the most relevant APARs.

Table 19-1 Accounting related APARs

APAR No.	DB2 Version	Description
PQ69983	DB2 V7	Corrections to accounting values with nested activity such as stored procedures, UDFs, and triggers.
PQ74772	DB2 V7	Various corrections to batch accounting reports and traces, including CLASS 3 SUSPENSIONS for stored procedures and UDF schedule.
PQ75064	DB2 V7	Correction to reflect the actual count of incremental binds on CALL :HV or CALL :HV USING DESCRIPTOR statements.

In the next sections we discuss the following topics:

- 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:

```
-DIS PROCEDURE (DEVL7083.*)
```

The report produced from this command is shown in Figure 19-4.

```
DSNX940I  -DB2G DSNX9DIS DISPLAY PROCEDURE REPORT FOLLOWS -
------- SCHEMA=DEVL7083
PROCEDURE          STATUS ACTIVE QUEUED MAXQUE TIMEOUT WLM_ENV
EMPAUDTS           STOPREJ     0      0      1       0 DB2GDEC1
EMPDTLSC           STARTED     0      0      1       0 DB2GDEC1
EMPDTLSJ           STARTED     0      0      1       0 DB2GWEJ1
EMPDTLSR           STOPABN     0      0      0       0 DB2GDER1
EMPDTL1J           STARTED     0      0      1       0 DB2GWEJ1
EMPODB1C           STOPQUE     0      0      1       0 DB2GODBA
EMPRSETC           STARTED     0      0      1       0 DB2GDEC1
EMPRST1J           STARTED     0      0      1       0 DB2GWEJ1
DSNX9DIS DISPLAY PROCEDURE REPORT COMPLETE
DSN9022I  -DB2G DSNX9COM '-DISPLAY PROC' NORMAL COMPLETION
***
```

Figure 19-4 Output of -DISPLAY PROCEDURE command

You can see from the report that all the procedures are started except EMPAUDTS, EMPDTLSR, and EMPODB1C. EMPAUDTS 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. EMPODB1C 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. EMPDTLSR is in STOPABN status, which indicates that the procedure experienced an abend and the maximum number of abends as defined by zparm STORMXAB had been reached. If you see many procedures with a STOPABN status, you may want to increase STORMXAB to minimize the need to start your procedure each time you experience an abend. This can be especially helpful in a test environment. In DB2 V8 you can specify the maximum abend count at the individual stored procedure level, so you can set the abend count to a higher value for stored procedures that abend frequently during testing. See Chapter 8, “Operational issues” on page 67 for more information on 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; and the number of times an SQL CALL statement timed out waiting for the procedure to be scheduled. 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.

19.2.2 Reporting on DB2 accounting class 7 and 8 data

DB2 for OS/390 and 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 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. See Table 19-2 for a description of the accounting classes.

Table 19-2 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

For more details on starting traces and the appropriate trace classes to choose, see *DB2 Performance Monitor for z/OS Version 7.2, Reporting User's Guide*, SC27-1651-02. Another source of information on this tool is the redbook *DB2 Performance Expert for z/OS*, SG24-6867. 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 DB2 Performance Expert (DB2 PE) or DB2 Performance Monitor (DB2 PM) 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 running on 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

See *DB2 Performance Monitor for z/OS Version 7.2, Report Reference*, SC27-1647-02 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 procedures 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 19-5.

```
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 19-5 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 19-6. 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 19-6 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 19-7 and Figure 19-8.

```
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 19-7 Thread Times panel of DB2 PE (page 1 of 2)

```
0  ________________________________________________________________________
  |                              Thread Times                              | _
  | Command ===> _________________________________________________________ |
  |                                                                        |
  |                                                          More:   - +   | +
  |   Other write I/O . . . . . . . . . . . :   0.000000               0   |
  |   Services task switch  . . . . . . . . :   0.000000               0   |
  |   Archive log (quiesce) . . . . . . . . :   0.000000               0   |
  |   Archive log read  . . . . . . . . . . :   0.000000               0   |
  |   Drain lock  . . . . . . . . . . . . . :   0.000000               0   |
  |   Claim release . . . . . . . . . . . . :   0.000000               0   |
  |   Page latch  . . . . . . . . . . . . . :   0.000000               0   |
  |   Stored procedures . . . . . . . . . . :  26.492065               3   |
  |   Notify messages . . . . . . . . . . . :   0.000000               0   |
  |   Global contention . . . . . . . . . . :   0.000000               0   |
  |                                                                        |
  | DB2 entry/exit events                                                  |
  |   Non stored procedures  . . . . . . . . :         2                   |
  |   Stored procedures  . . . . . . . . . . :        10                   |
  |                                                                        |
  | Class 5 (IFI)                                                          |
  |   Elapsed time . . . . . . . . . . . . . :       N/P                   |
  |   TCB time . . . . . . . . . . . . . . . :       N/P                   |
```

Figure 19-8 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 can 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 19-9.

```
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 19-9 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 19-10, Figure 19-11, and Figure 19-12 on page 311. 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 19-12) and there were nine rows fetched from cursors (Fetch = 9 on Figure 19-11).

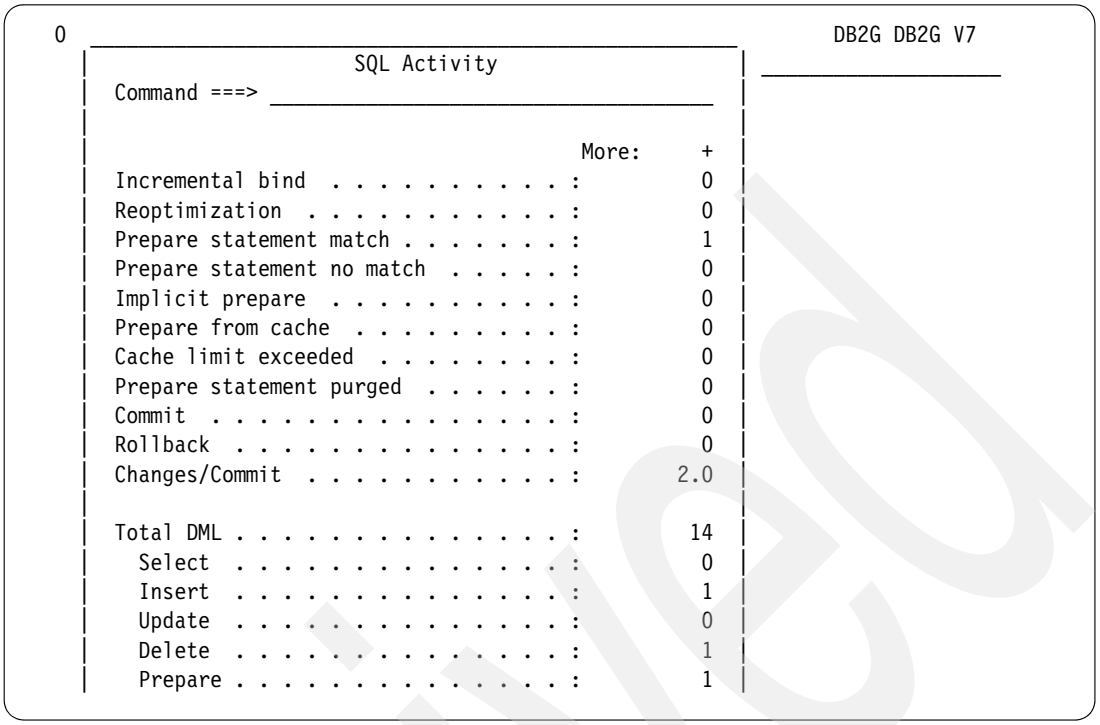


Figure 19-10 SQL Activity panel of DB2 PE (page 1 of 3)

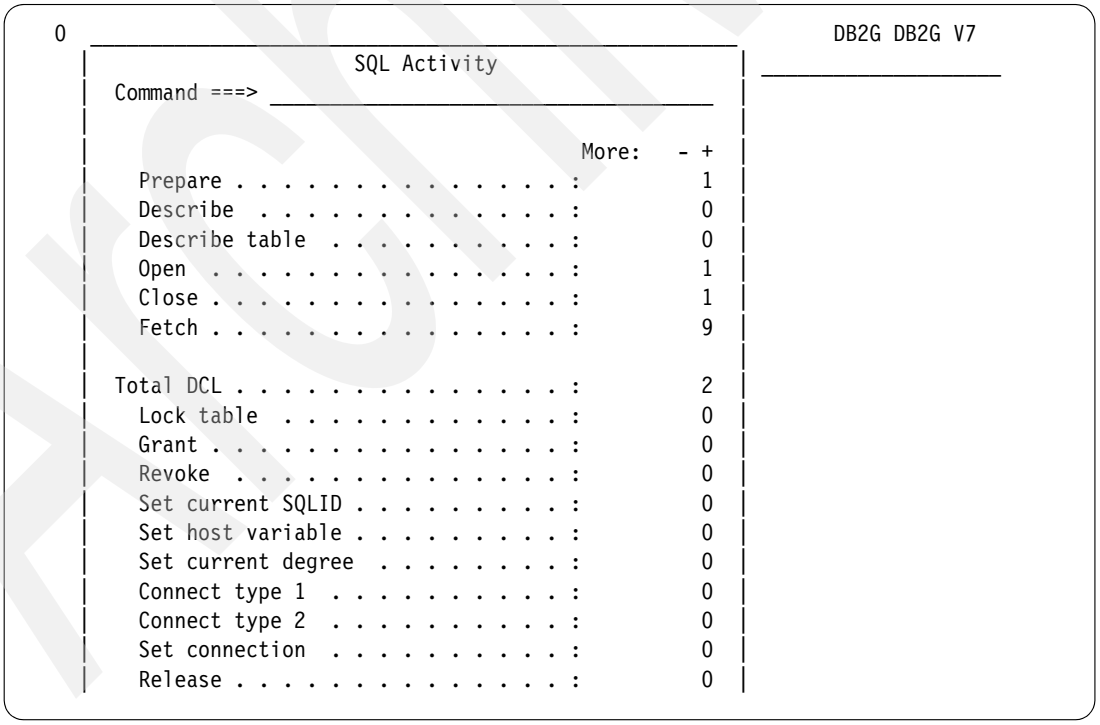


Figure 19-11 SQL Activity panel of DB2 PE (page 2 of 3)

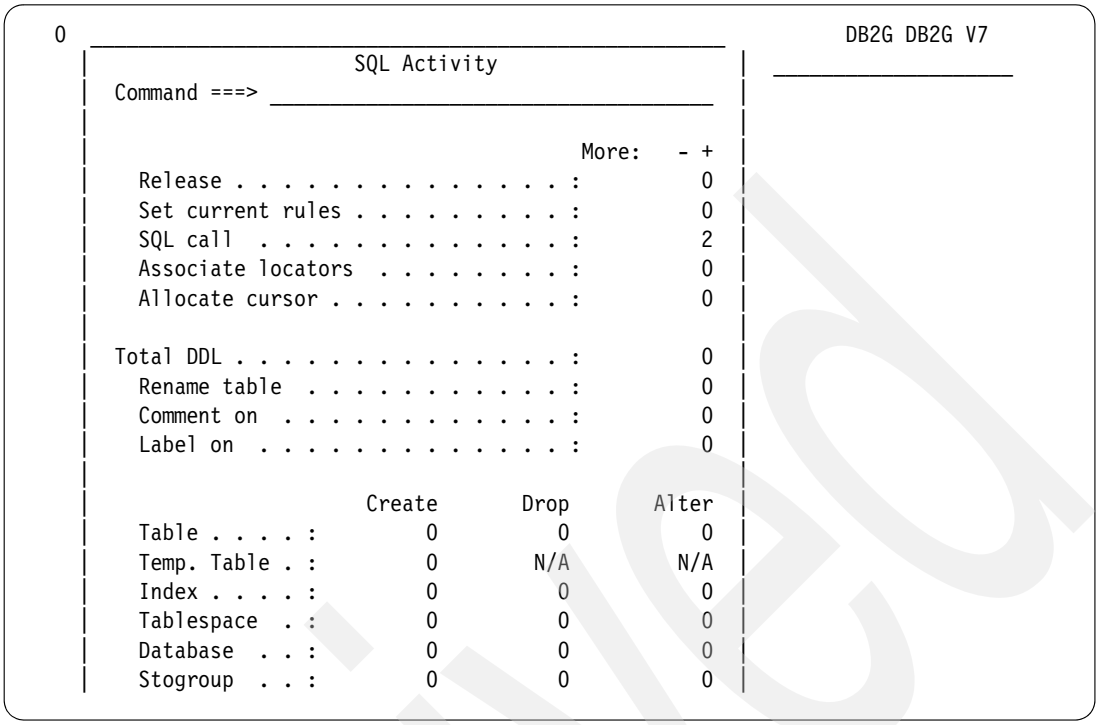


Figure 19-12 SQL Activity panel of DB2 PE (page 3 of 3)

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 19-4 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 19-4 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 19-5 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 19-5 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 Resource Management Facility User's Guide*, SC33-7990-01 for more 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 3 (suspension) time (see Figure 19-7 on page 308, and Figure 19-8 on page 308). You can adjust the NUMTCB value of each application environment to meet the performance needs for those environments. See 20.1.3, “NUMTCB” on page 322 for more details on the implications of your NUMTCB settings.

CREATE PROCEDURE

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.
  - 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.
- PARAMETER STYLE (GENERAL, GENERAL WITH NULLS, DB2SQL)
  - 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 DB2SQL, 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.

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. If you do not provide any classification rules for your DDF workload, then default service classes will 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 run-time library access and options

Language Environment (LE) is a component of the OS/390 and z/OS operating systems. LE establishes a common run-time environment for stored procedures that may be written in

many different high level languages. The run-time library access needs to be facilitated for performance, and the run-time 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 regards 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 run-time options when you create your stored procedures. See 19.3, “Recommendations” on page 316 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 within 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 run-time options. When stored procedures that are defined with PROGRAM TYPE SUB with one set of LE run-time options execute in an application environment, and are followed by a stored procedure with a different set of LE run-time options in the same environment, all the stored procedures with the initial set of LE run-time options are invalidated, and must be reloaded at the next execution. When the stored procedures with the original run-time 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.

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 re-created 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

DSNTRACE is a facility that can be used to capture all trace messages for offline reference and diagnosis. We do not recommend that you 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.

19.3 Recommendations

Here, we group some recommendations for parameter definition of:

- 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 76 for more details on each option of the CREATE PROCEDURE statement:

- 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 recommend that you specify NO for stored procedures that are called locally.

- PARAMETER STYLE (GENERAL WITH NULLS or DB2SQL)

  We recommend that you specify either GENERAL WITH NULLS or DB2SQL. 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.

- 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. 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 319 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) run-time options to minimize storage usage.

There are a number of LE run-time options that you can specify to minimize storage usage below the 16 MB line. They are documented in Chapter 25, “*DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415. 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 D, “Additional material” on page 651 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 proc. 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 proc that provide access to specific data sets. There is a wide variety of other possible dependencies for a particular WLM proc.

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 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 325.

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). 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 further 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.
- When the stored procedure finishes, it relinquishes 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.

In this case the top-level stored procedure requires a TCB, and each of stored procedures it calls requires its own TCB. These nested (or second-level) stored procedures run at different times, so the most stored procedures 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 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 20.1.3, "NUMTCB" on page 322 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. See 20.1.4, "WLM management of server address spaces" on page 323 for why this might be the case. Similarly, WLM might not immediately terminate a server address space when all of its TCBs become unused.

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 elongated, 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. 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 are created in different server address spaces.

20.1.3 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 despatched 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 24 bit virtual storage line and above the 24 bit virtual storage line. Each stored procedure allocates and uses its own virtual storage, 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 as each other, 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 as 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. Otherwise, the server address space might exhaust virtual storage.
- Many DB2-supplied stored procedures, such as DSNUTILS must be run in a server address space with a NUMTCB of 1 to ensure correct serialization of access to resources.

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 accumulates service, mainly CPU. If you define periods for the service class, you define durations for all periods except the last. When the transaction's accumulated service exceeds the duration for period one, the transaction transitions to period two. When, in turn, the transaction accumulates enough additional service to exceed the duration of period two, the transaction transitions to period three, and so on. Because the final period has no duration, any transaction transitioning to the final period terminates in this 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 such terms 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: An installation assigns SCs to individual pieces of work, but not to individual stored procedures. It is the caller of the stored procedure that determines the SC the stored procedure runs under.

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 ten 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 ten 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 stored procedures is delayed by this additional wait for a server address space (and hence a TCB). The ten second minimum interval introduces latency. The ten 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 20.1.3, "NUMTCB" on page 322, 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 reduces the likelihood server address spaces will be stopped.

In general, 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 if 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 STOR.PRC SCHED field shows this application was delayed for 0.783497 seconds. As the ELAPSED TIME field is only 5.919586 seconds, this delay is a significant contributor. In short, 13% 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 5.810414 seconds. Most of the time was caused by a deliberately introduced wait of 5 seconds in the stored procedure itself.

Example 20-2 Sample DB2 Performance Monitor Accounting Report listing

```
1   LOCATION: DB2G                          DB2 PERFORMANCE EXPERT (V1)
       GROUP: N/P                            ACCOUNTING REPORT - LONG                    REQ
      MEMBER: N/P
   SUBSYSTEM: DB2G                           ORDER: PRIMAUTH-PLANNAME                     IN
 DB2 VERSION: V7                                SCOPE: MEMBER

 PRIMAUTH: PAOLOR7  PLANNAME: DISTSERV

 ELAPSED TIME DISTRIBUTION                                    CLASS 2 TIME DISTRIBUTION
 ---------------------------------------------------------    ------------------------------
 APPL   |================================================> 100%   CPU    |=====================
 DB2    |                                                        NOTACC |
 SUSP   |======> 13%                                            SUSP   |

 AVERAGE       APPL(CL.1)  DB2 (CL.2)  IFI (CL.5)    CLASS 3 SUSPENSIONS   AVERAGE TIME  AV.EVENT
 ------------  ----------  ----------  ----------    --------------------  ------------  --------
 ELAPSED TIME    5.919586    0.003098         N/P    LOCK/LATCH(DB2+IRLM)      0.000006      0.04
  NONNESTED      0.109173    0.001251         N/A    SYNCHRON. I/O             0.000596      0.01
  STORED PROC    5.810414    0.001847         N/A     DATABASE I/O             0.000596      0.01
  UDF            0.000000    0.000000         N/A     LOG WRITE I/O            0.000000      0.00
  TRIGGER        0.000000    0.000000         N/A    OTHER READ I/O            0.000000      0.00
                                                     OTHER WRTE I/O            0.000000      0.00
 CPU TIME        0.006272    0.001066         N/P    SER.TASK SWTCH            0.000000      0.00
  AGENT          0.006272    0.001066         N/A     UPDATE COMMIT            0.000000      0.00
   NONNESTED     0.000870    0.000577         N/P     OPEN/CLOSE               0.000000      0.00
   STORED PRC    0.005401    0.000489         N/A     SYSLGRNG REC             0.000000      0.00
   UDF           0.000000    0.000000         N/A     EXT/DEL/DEF              0.000000      0.00
   TRIGGER       0.000000    0.000000         N/A     OTHER SERVICE            0.000000      0.00
  PAR.TASKS      0.000000    0.000000         N/A    ARC.LOG(QUIES)            0.000000      0.00
                                                     ARC.LOG READ              0.000000      0.00
 SUSPEND TIME         N/A    0.784099         N/A    STOR.PRC SCHED            0.783497      1.00
  AGENT               N/A    0.784099         N/A    UDF SCHEDULE              0.000000      0.00
  PAR.TASKS           N/A    0.000000         N/A    DRAIN LOCK                0.000000      0.00
                                                     CLAIM RELEASE             0.000000      0.00
 NOT ACCOUNT.         N/A         N/C         N/A    PAGE LATCH                0.000000      0.00
 DB2 ENT/EXIT         N/A        4.04         N/A    NOTIFY MSGS               0.000000      0.00
 EN/EX-STPROC         N/A       12.04         N/A    GLOBAL CONTENTION         0.000000      0.00
 EN/EX-UDF            N/A        0.00         N/A    COMMIT PH1 WRITE I/O      0.000000      0.00
 DCAPT.DESCR.         N/A         N/A         N/P    ASYNCH IXL REQUESTS       0.000000      0.00
 LOG EXTRACT.         N/A         N/A         N/P    TOTAL CLASS 3             0.784099      1.05
```

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.

You need to understand how the Class 1 and Class 2 times are inter-related when processing SMF 101 Accounting Trace.

Table 20-1 shows how to handle the Class 1 and Class 2 times when stored procedures are involved. Performing the calculations this way allows you to sensibly calculate a Class 2 Other Waits time by deducting *all* the Class 3 wait times, including QWACCAST and QWACUDST, from the total Class 2 elapsed time.

Note: Stored procedures schedule wait and UDF schedule wait times are not calculated at the package level. The fields QPACCAST and QPACUDST always contain zeroes.

Table 20-1 Suggested major DB2 Accounting Trace time buckets with stored procedures

Time bucket	Calculation	Notes
Class 1 total elapsed time	QWACESC-QWACBSC	Total time, including application time and in DB2 time. Includes all class 1 components
Class 1 stored procedure elapsed time	QWACSPEA	Total stored procedure time, including application time and in DB2 time. Also includes schedule wait time.
Class 1 User Defined Function elapsed time	QWACUDEA	Total UDF time, including application time and in DB2 time. Also includes schedule wait time.
Class 1 non-nested elapsed time	(QWACESC -QWACBSC) - QWACSPEA - QWACUDEA	Total time not spent in stored procedures or UDFs, including application time and in DB2 time.
Class 2 total elapsed time	QWACASC + QWACSPEB + QWACUDEB + QWACCAST + QWACUDST	In this calculation you need to add in the scheduling waits so that you can take all the Class 3 waits out and still have a meaningful Class 2 Other time. This is in DB2 time.
Class 2 stored procedure elapsed time	QWACSPEB	Does not included schedule wait time. This is in DB2 time.
Class 2 user defined function time	QWACUDEB	Does not include schedule wait time. This is in DB2 time.
Class 2 non-nested elapsed time	QWACASC	Total in DB2 time, not spent in stored procedures or UDFs or schedule waits.

Time bucket	Calculation	Notes
Class 1 total CPU time	QWACEJST - QWACBJST) + QWACSPCP +QWACUDCP	Total CPU time, including application CPU time and in DB2 CPU time
Class 1 stored procedure CPU time	QWACSPCP	Total stored procedure CPU time, including application CPU time and in DB2 CPU time
Class 1 User Defined Function CPU time	QWACUDCP	Total UDF CPU time, including application CPU time and in DB2 CPU time
Class 1 non-nested CPU time	QWACEJST - QWACBJST	Total CPU time not spent in stored procedures or UDFs, including application CPU time and in DB2 CPU time
Class 2 total CPU time	QWACAJST + QWACSPTT + QWACUDST	Total in DB2 CPU time, including time spent in stored procedures and UDFs
Class 2 stored procedure CPU time	QWACSPTT	Total stored procedure in DB2 CPU time
Class 2 User Defined Function CPU time	QWACUDTT	Total UDF in DB2 CPU time
Class 2 non-nested CPU time	QWACAJST	Total CPU time in DB2, not spent in stored procedures or UDFs

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 325, 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 further 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 how the application behaves or the stored procedure is set up, so that each stored procedure execution consumes fewer resources, and is said to be lighter weight, leading to the server address spaces that service the stored procedure being 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 is discussed in 20.2.3, “Reducing the resource profile of stored procedures” on page 329.

- 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.

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 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.

- 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, despite the increased real memory cost.

These techniques often interact with each other, sometimes negatively. So select the techniques carefully.

To help analyze use of resources by different types of stored procedures name server address spaces so that it is clear which AE they serve. With this naming convention SMF Type 30 Subtypes 2 and 3, Accounting Interval records can be used to determine the resource consumption by each server address space. These records include CPU time and virtual storage usage.

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 as 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. The chapter's 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, which 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 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) run time 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 other books that provide data set tuning guidance.

Part 6

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

Enhancements to stored procedures with DB2 V8

In this part we introduce the major enhancements that DB2 for z/OS Version 8 is providing for the users of stored procedures. First, we provide a general overview, and then we illustrate with examples some of the programming enhancements.

This chapter contains the following:

- IBM DB2 Universal Driver for SQLJ and JDBC features
- DDF communication database enhancements
- Enhancements for stored procedures and UDFs:
  - Maximum failures
  - Exploit WLM server task thread management
  - Nested stored procedure result sets for JDBC and ODBC applications
  - Enhancements to SQL stored procedure language
  - COMPJAVA stored procedures no longer supported
  - DB2 established stored procedures
- CURRENT PACKAGE PATH special register
- New LOB parameters

Please refer to the recently available DB2 V8 documentation for details. The available manuals are listed in “Related publications” on page 657 and are downloadable from the DB2 for z/OS and OS/390 library Web page:

```
http://www-306.ibm.com/software/data/db2/os390/library.html
```

22.1 IBM DB2 Universal Driver for SQLJ and JDBC features

Prior to DB2 V8, the client for Linux, UNIX, Microsoft Windows, and OS/2 platforms provides the basis for three distinct products:

- DB2 Run-Time Client

  Also known as the Client Application Enabler (CAE)

- DB2 Application Development Client

  Formerly known as the Software Development Kit (SDK)

- DB2 Connect Personal Edition

  DB2 Connect Enterprise Edition is based on a combination of the UNIX, Windows, OS/2 engine infrastructure, and DB2 Connect Personal Edition.

Each product is positioned as either the client for a Linux, UNIX, Windows application server, or the client for a z/OS database server.

Currently, access to a Linux/UNIX/Windows (LUW) server and a z/OS server use different database connection protocols, for example, DB2RA, DRDA, "net driver." Each protocol defines a different set of methods to implement the same functions. To provide transparent access across the DB2 Family, the database connection protocols are now standardized, and all of them use the Open Group's DRDA Version 3 standard, which provides an open, published architecture that enables communication between applications, application servers, and database servers on platforms with the same or different hardware and software architectures. This new architecture is called the Universal Driver. The first deliverable of this new architecture is the *IBM DB2 Universal Driver for SQLJ and JDBC Version 1.0*, also known as the IBM Java Combined Client.

The Universal Driver is architected as an abstract JDBC processor that is independent of driver-type connectivity or target platform. (More more information on driver types, see 22.1.1, "IBM JDBC Type 4 driver" on page 340.) The IBM DB2 JDBC Universal Driver is an architecture-neutral JDBC driver for distributed and local DB2 access.

Since the Universal Driver has a unique architecture as an abstract JDBC state machine, it does not fall into the conventional driver-type categories as defined by Sun. Because the Universal Driver is an abstract machine, driver types become connectivity types.

This abstract JDBC machine architecture is independent of any particular JDBC driver-type connectivity or target platform, allowing for both all-Java connectivity (Type 4) or JNI-based connectivity (Type 2) in a single driver. A single Universal Driver instance is loaded by the driver manager for both Type 4 and Type 2 implementations. Type 2 and 4 connections may be made (simultaneously if desired) using this single driver instance.

Important: The new Universal Driver for SQLJ and JDBC is planned to be made available in DB2 V7 as well through the maintenance stream. At the time of writing, the APAR number mentioned for the Universal Driver Version 7 is PQ80841.

Objectives

The new common run-time environment fulfills the following key requirements:

- Have a single driver for Linux, UNIX, Windows, and z/OS. This eliminates the major cause of today's Java porting problems, and also enables to deliver performance and functional enhancements quicker (since they only need to be developed once).

- Enhance the current API to provide a fully compliant JDBC 2.0 driver, for both a Type 2 and Type 4 JDBC driver. The enhanced functionality will not be in the current Type 2 driver. It is only made available in the new Universal Client based Type 2 driver. The current Type 2 driver will be shipped for compatibility reasons, but will not be enhanced.
- Reduce the client footprint. Footprint® reduction is achieved by eliminating the multiple layers of processing, which reduces both disk and memory consumption on the client. An additional gain is made by partitioning the client into three distinct components:
  - A C based client supporting all SQL and CLI/ODBC access (this will not be discussed any further in this section, as we are focusing on the Java part).
  - A Java based client supporting all SQLJ and JDBC access
  - An administrative client providing a consistent set of administrative function, including replication across all platforms. This is not discussed any further, as we concentrate on the Java client.

  Each of the components above can be installed separately, or as any combination of the three, to allow consistent access to both a LUW server, and a z/OS server without any additional installation steps.
- Provide a full Java application development process for SQLJ, by:
  - Providing a fully portable customized SQLJ profile
  - Enabling the bind of DB2 packages from the client (using the Type 4 driver)
- Trace improvements, by allowing:
  - Turning traces on and off dynamically, and
  - Allowing multiple levels of tracing, with different levels of detail

Benefits for DB2 UDB for z/OS

At first glance this change might look like something that only impacts DB2 UDB for LUW and DB2 Connect users. This is certainly not the case, as explained here:

- First, and most importantly, because of a common code base, the functions provided on DB2 UDB for LUW and DB2 UDB for z/OS is exactly the same, not just similar. This largely improves DB2 Family compatibility. For example, it enables users to develop on LUW, and deploy on z/OS without having to make any change.
- As mentioned before, there are also many functionality enhancements to the Java API for the (Universal Driver based) Type 2, and the new Type 4 JDBC driver to make it fully compliant with the JDBC 2.0 standard.
- With the elimination of the “private protocols” used by the LUW clients in previous versions, using DRDA will render better performance.
- Ease of installation and deployment. The Java type 4 driver is 100% Java code, without dependencies on a run-time or DLL. Installation is merely a copy operation of a .jar and .zip file. Deployment on z/OS can now be completely done from the workstation.

Functional enhancements

Not only will the new DB2 Universal Driver bring more consistent application behavior, it also introduces several new functions, making it fully JDBC 2.0 compliant, such as:

- Java API enhancements
- Nested stored procedure result sets for JDBC and ODBC applications
- Extended DESCRIBE
- SQLcancel
- LOB streaming

22.1.1 IBM JDBC Type 4 driver

Not only does DB2 V8 provide a new JDBC driver architecture, known as the IBM DB2 Universal Driver for SQLJ and JDBC, IBM also delivers a JDBC Type 4 driver for the first time. This driver is also shipped with DB2 UDB for LUW Version 8.

As a reminder, we give a brief description of the JDBC driver architectures based upon the JDBC 3.0 specification:

Type 1: Drivers that implement the JDBC API as a mapping to another data access API, such as ODBC. Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC bridge driver is an example of a Type 1 driver.

Type 2: Drivers that are written partly in the Java programming language, and partly in native code. The drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited. Notice that a Type 2 has a native component that is part of the driver and is separate from the database access API.

Type 3: Drivers that use a pure Java client and communicate with a middleware server using a database independent protocol. The middleware server then communicates the client's requests to the data source.

Type 4: Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source. In case of the Universal Driver, the DRDA protocol is used to talk directly to the data source.

Note: The number in the driver type has no meaning whatsoever. Do not assume that because 4 is greater than 2, that a Type 4 driver is better than a Type 2 driver. In fact, a Type 2 driver is almost certain to outperform a Type 3 or Type 4 driver, because it does not have to route through a network layer. Normally, a Type 2 driver is the best suitable driver from the point of view of performance and scalability.

Using the Type 4 driver, a client application can now talk directly to DB2 UDB for z/OS, without going through DB2 Connect (although a DB2 Connect licence is required to be able to use the Type 4 driver).

The support of a Type 4 driver, combined with a fully portable SQLJ customized profile, will allow WebSphere to provide better tooling to support the development of SQLJ applications.

22.1.2 New IBM JDBC Type 2 driver

In addition to a brand new Type 4 driver, DB2 V8 delivers a new Type 2 driver as well. It is also based on the same common code base of the Universal Driver for SQLJ and JDBC. The current Type 2 driver (available since DB2 V5) will still be shipped with V8 for compatibility reasons, but will not be enhanced. All enhancements described hereafter are only available with the new Type 2 driver delivered with Universal Driver for JDBC and SQLJ.

22.1.3 Java API enhancements

The current API has been enhanced to provide a fully compliant JDBC 2.0 driver, for both the Type 2 and Type 4 JDBC driver. The enhanced functionality will not be made available in the current Type 2 driver. It will only by made available in the new DB2 Universal Driver based, Type 2 driver. Among these enhancements are:

- Scrollable cursor support (exploiting DB2 engine scrolling)

- Batch updates support
- Improved security for DB2 authentication
- Improved Java SQL error information through the `DB2Diagnosable` class. This class allows reporting of the contents of the SQLCA and SQL error message text.
- *Native* DB2 server SQL error messages can be returned when explicitly requested by the `getSQLErrorMessage()` method. EachDB2 server provides an “error message” stored procedure to allow a DB2 Client to retrieve the “native” message text from the target DB2 server.
- Java API for Set Client Information (SQLESETI).

In the meantime, development is underway to deliver a JDBC 3.0 compliant Universal Driver.

22.1.4 SQLJ

SQLJ support has been around for a while now. It provides superior performance, because it uses static SQL (in contrast to JDBC that uses dynamic SQL), and uses a powerful authorization model (like static SQL in other programming languages). But prior to the Universal Driver, the development and deployment of SQLJ applications was somewhat cumbersome.

Existing SQLJ program preparation process

Figure 22-1 shows the existing SQLJ program preparation process. After creating the serialized profile by means of the SQLJ translator, you have to execute the **db2profc** utility to create a DBRM, and then bind the DBRM into a set of packages (one package for each isolation level, UR, CS, RS, and RR). Even if you prefer to develop your Java applications on a workstation, the (uncustomized) serialized profile has to be shipped to the host before you can run the **db2profc** utility. This is because **db2profc** creates DBRMs, which are a DB2 UDB for z/OS and OS/390 only feature.

The **db2profc** utility also creates a customized serialized profile. Unfortunately, after customization, the profile is no longer portable.

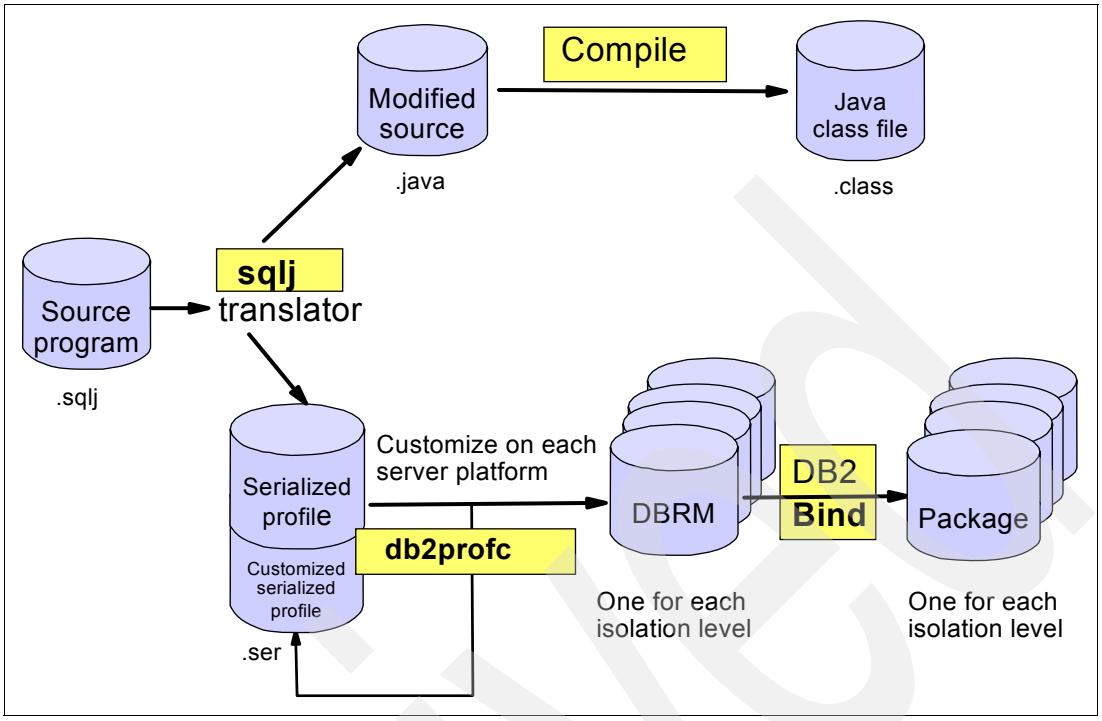


Figure 22-1 Existing SQLJ preparation process

Universal Driver SQLJ program preparation process

Using the new Universal Driver, DBRMs (or .bnd files) are no longer required, as shown in Figure 22-2. Using the **db2sqljcustomize** command, you can customize the serialized profile and bind the packages at the same time against the target DB2 system. With the Type 4 driver, we connect from any platform directly to the target DB2 system, do the online checking (highly recommended), and bind the packages on the target DB2 system.

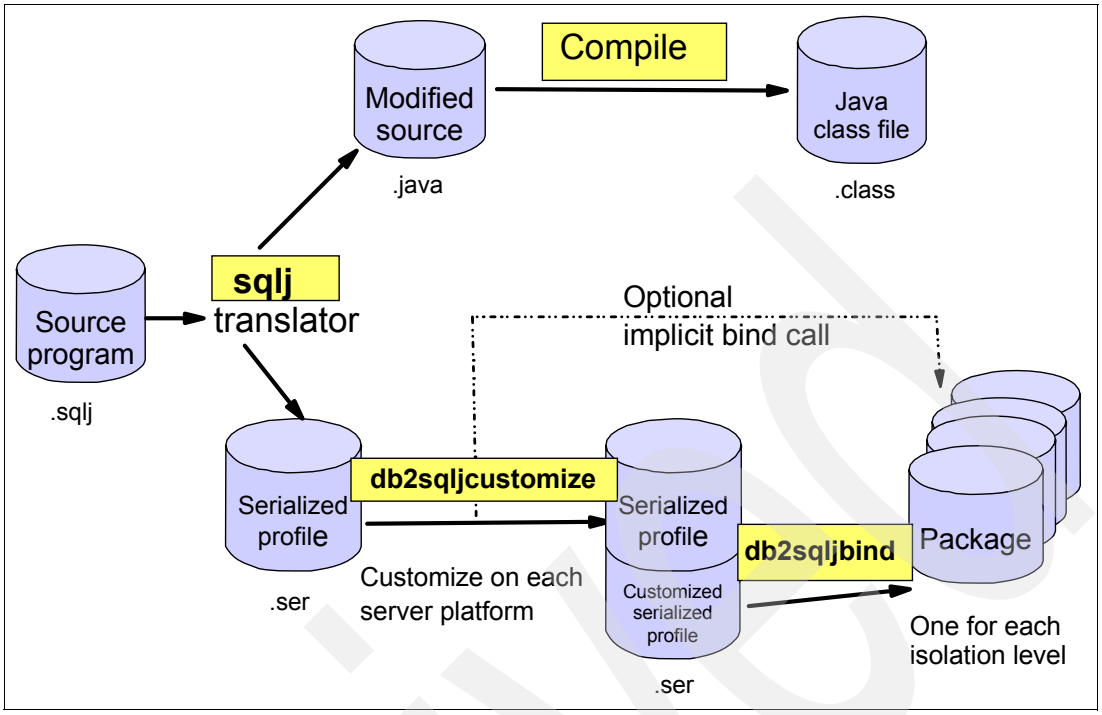


Figure 22-2 Universal Client SQLJ preparation process

For example, when you develop on the workstation using WebSphere Studio Application Developer (WSAD), you may now use the Type 4 driver to bind the packages against the DB2 UDB for z/OS system. You no longer have to ship the uncustomized profile to the z/OS system for customization.

In addition, the new Universal Driver customizes the serialized profile in such a way that it remains portable. You can execute using the same customized program files against any platform, as long as the **db2sqljbind** utility was used to connect to the new location and bind the correct program packages.

WSAD Version 5 will provide support for this new application development scheme used by the Universal Driver for SQLJ and JDBC.

22.1.5 Extended DESCRIBE

Some applications require a great deal of descriptive information to be returned from the server. With this enhanced function, the requester can control the amount and the type of information returned from a prepare, a describe, a query, or an execution of a stored procedure.

ODBC/CLI (on LUW) and JDBC applications can request database metadata through a standard set of APIs. In order for ODBC/CLI and JDBC drivers to accurately return this information, they must be sensitive to the underlying database server, which the application is requesting the information about.

The extended describe feature is enabled by the DESCSTAT DSNZPARM (as well as the enhancements to the DRDA flow). It provides:

- Additional descriptive information for a cursor or a result set
- Information about whether or not a column can be updated

- Information about whether or not a column is a primary key or a preferred candidate key member
- Information about whether or not a column is an expression or an actual column
- Information about whether or not a column is a generated column or a real table column
- The fully qualified view or table name, location.schema.name
- The fully qualified column name, location.schema.name

For example, a DB2 server can provide extended descriptive information to support the JDBC 2.0 updateRow and deleteRow methods.

22.1.6 SQLcancel

The SQLcancel() statement allows an ODBC/CLI or JDBC application to cancel an SQL request *long* running on a DB2 server. Note that SQL cancel() is at a more granular level than the DB2 -CANCEL THREAD command. SQLcancel() only rolls back the currently executing SQL statement, not the entire unit of work. In addition, the thread is not destroyed in the process, but is allowed to continue processing.

If the database server is not in an interruptible state, the request completed before DB2 can interrupt, or the request is not interruptible, then DB2 returns a DRDA reply message back to the client confirming the attempt. A new sqlcode -952 is returned if the SQL statement was interrupted (rollback worked). If the SQL statement just reads the data (not write), then we issue -952 even if the rollback did not work.

Currently, only dynamic SQL statements are interruptible. Stored procedures cannot be interrupted. Transaction level SQL statements like connect, commit, and rollback cannot be interrupted. Even bind package cannot be interrupted.

22.2 DDF communication database enhancements

In this section we examine the following enhancements:

- Requester database ALIAS
- Server location alias
- Member routing in a TCP/IP network

22.2.1 Requester database ALIAS

When connecting to a DB2 UDB for z/OS and OS/390 system through DRDA, you address the entire DB2 subsystem by using its *location name*. A DB2 UDB for LINUX/UNIX/Windows database is known in the network by its *database name*. If the requester is a DB2 UDB for z/OS, you must specify the database name of the DB2 UDB for LUW system you want to connect to, in the LOCATION column of the SYSIBM.LOCATION catalog table.

Up to DB2 V7, there is always a one-to-one mapping between location name and database name, as there is a unique index on the LOCATION column. As you can see in Figure 22-3, prior to DB2 V8, there is no way to access multiple DB2 UDB for LUW databases that have the same name (even when they reside on different machines).

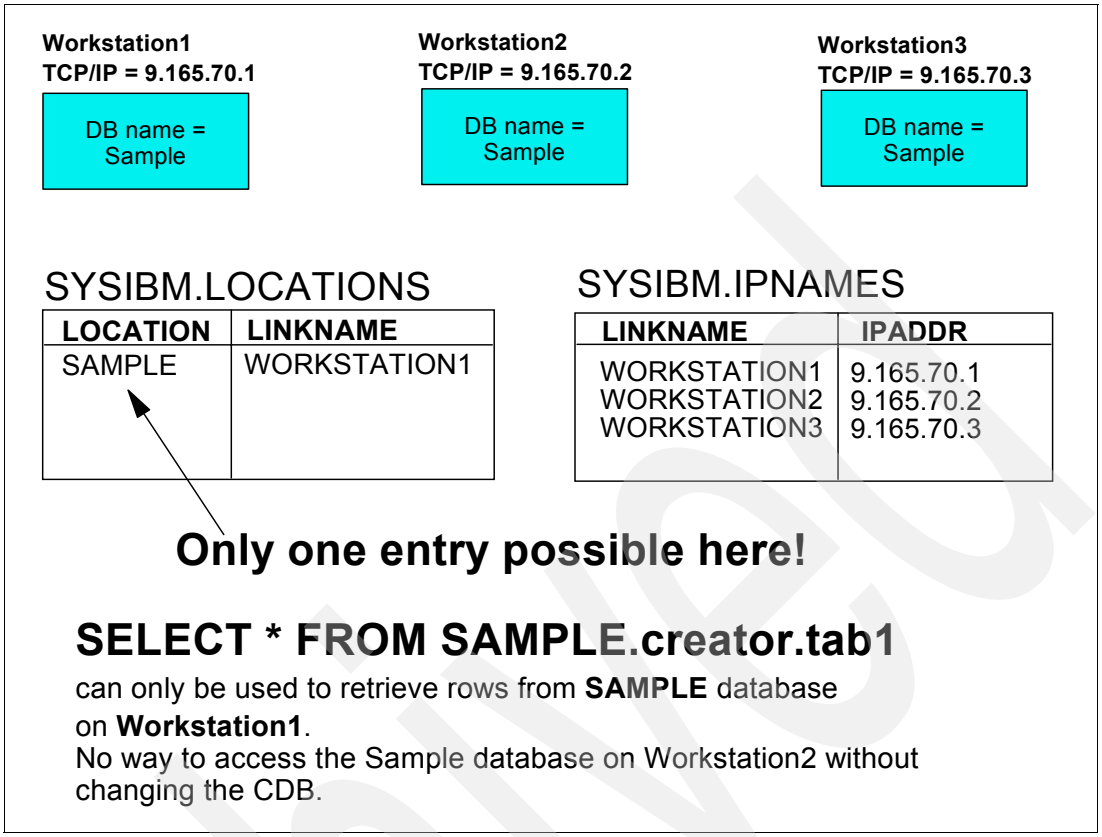


Figure 22-3 Access LUW database without DBALIAS

This restriction is removed in DB2 V8. A new column DBALIAS is added to the SYSIBM.LOCATIONS table. As reported in Figure 22-4, you continue to specify the LOCATION name as the first qualifier of your three part table name in your SELECT statement [1]. The mapped LINKNAME links you to the corresponding entry in SYSIBM.IPNAMES [2], which provides the correct TCP/IP address for the workstation you want to access [3]. The entry in column DBALIAS of SYSIBM.LOCATIONS points your SELECT statement to the real database name on the DB2 UDB for LUW that you want to connect to. You can now access the sample database on every LINUX, UNIX, and Windows system, even if thousands of them exist with the same database name.

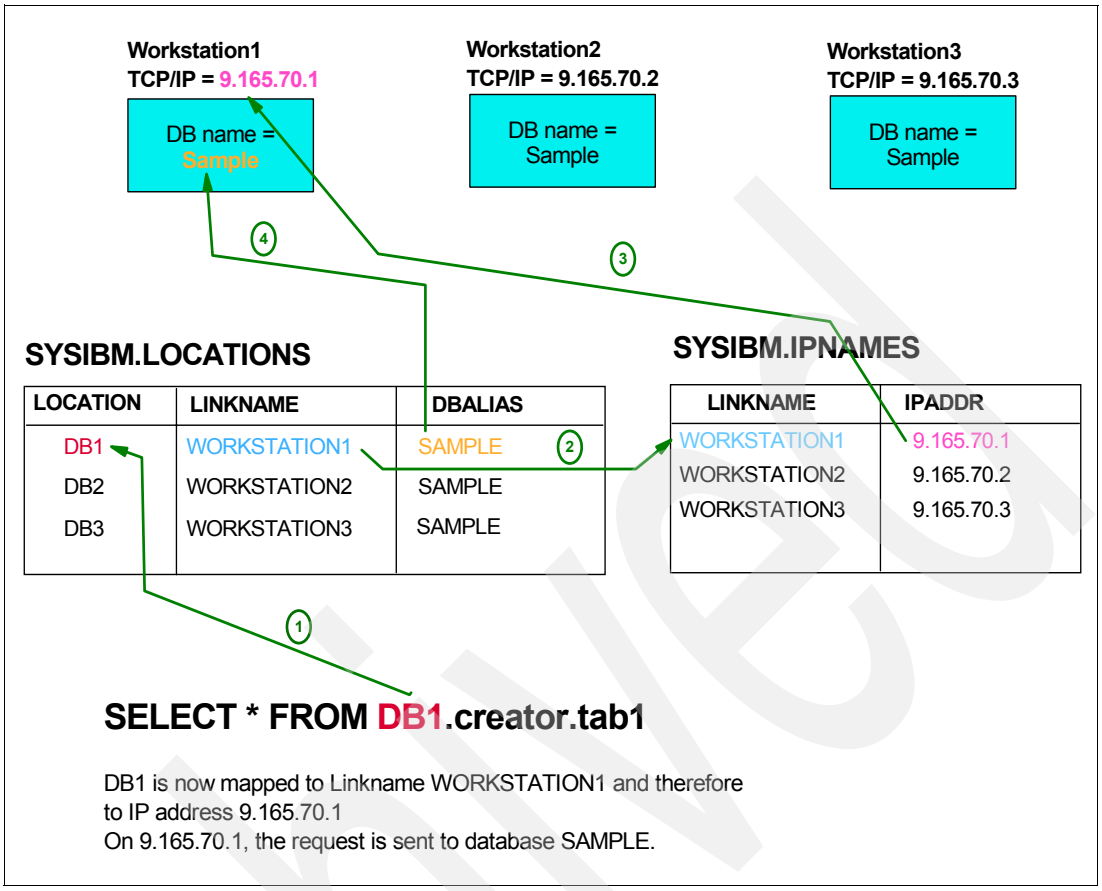


Figure 22-4 Access LUW database with DBALIAS

22.2.2 Server location alias

As mentioned before, a DB2 UDB for z/OS server is known in the network by its location name. This name is used by applications to identify a DB2 subsystem or an entire DB2 data sharing group. When two or more DB2 subsystems are consolidated to a single DB2 data sharing group, multiple locations must be consolidated into a single location (because the entire data sharing group uses the same location name). This means that all applications that use the old location name (used when the DB2 was still a stand-alone subsystem) need to be changed to access the location name of the data sharing group.

To ease this type of migration, DB2 V8 allows you to define up to eight alias names in addition to the location name for a DB2 data sharing group. A *location alias* is an alternative name that a requester can use to access a DB2 subsystem. You can define those location alias names with the CHANGE LOG INVENTORY utility (DSNJU003). As shown in Figure 22-5, you do not have to change the location names in your applications programs to be able to access your data after migrating to a new data sharing group. The only thing you must do is to add additional location alias names to your BSDS data sets on each member of the data sharing group.

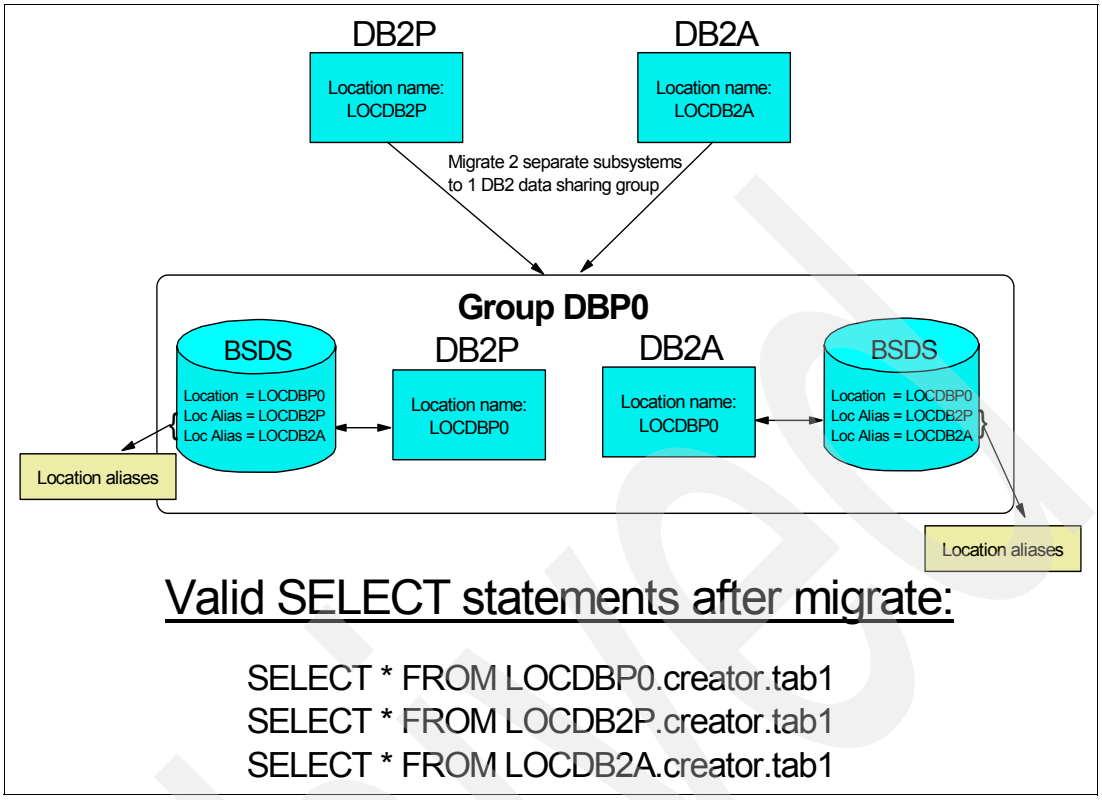


Figure 22-5 Location alias name

The syntax for the CHANGE LOG INVENTORY utility is:

```
DDF ALIAS = aliasname
```

The distributed data facility communication record in the BSDS data sets has been changed to store the location alias names you have specified for your subsystem. Figure 22-6 shows the output of the PRINT LOG MAP utility (DSNJU004). You can see that the location alias name DB7OGRP was added for our subsystem.

```
**** DISTRIBUTED DATA FACILITY ****
                      COMMUNICATION RECORD
                    14:26:17 SEPTEMBER 30, 2003
LOCATION=DB7O ALIAS=DB7OGRP
LUNAME=LU1 PASSWORD=(NULL) GENERICLU=(NULL) PORT=33744 RPORT=33745
```

Figure 22-6 DDF communication record

22.2.3 Member routing in a TCP/IP network

Currently, in a data sharing environment remote TCP/IP connections are normally set up to automatically balance connections across all members of a data sharing group. Sometimes the default TCP/IP workload balancing does not meet your needs, and you might therefore want to be able to route requests from certain DB2 UDB for z/OS requesters to specific members of your data sharing group, similar to the support for SNA connections that use the SYSIBM.LULIST table.

To achieve this, we combine the server location alias feature described in 22.2.2, “Server location alias” on page 346 with the use of a new table in the CDB, namely SYSIBM.IPLIST. Refer to Figure 22-7 to understand the process.

A location alias, LOCDBP1, has been defined for data sharing members DBP1 and DBP2. Note however that this alias does not exist in the BSDS for DBP3. In SYSIBM.LOCATIONS, you can find two entries. One for the “normal” location name of the data sharing group (LOCDBP0), and one for the location alias (LOCDBP1). If you now enter rows into the new communication database table SYSIBM.IPLIST, any requests for location name LOCDBP1 are routed to either of the mapped TCP/IP addresses in SYSIBM.IPLIST (for instance, 9.165.70.1 or 9.165.70.2 in Figure 22-7) [1].

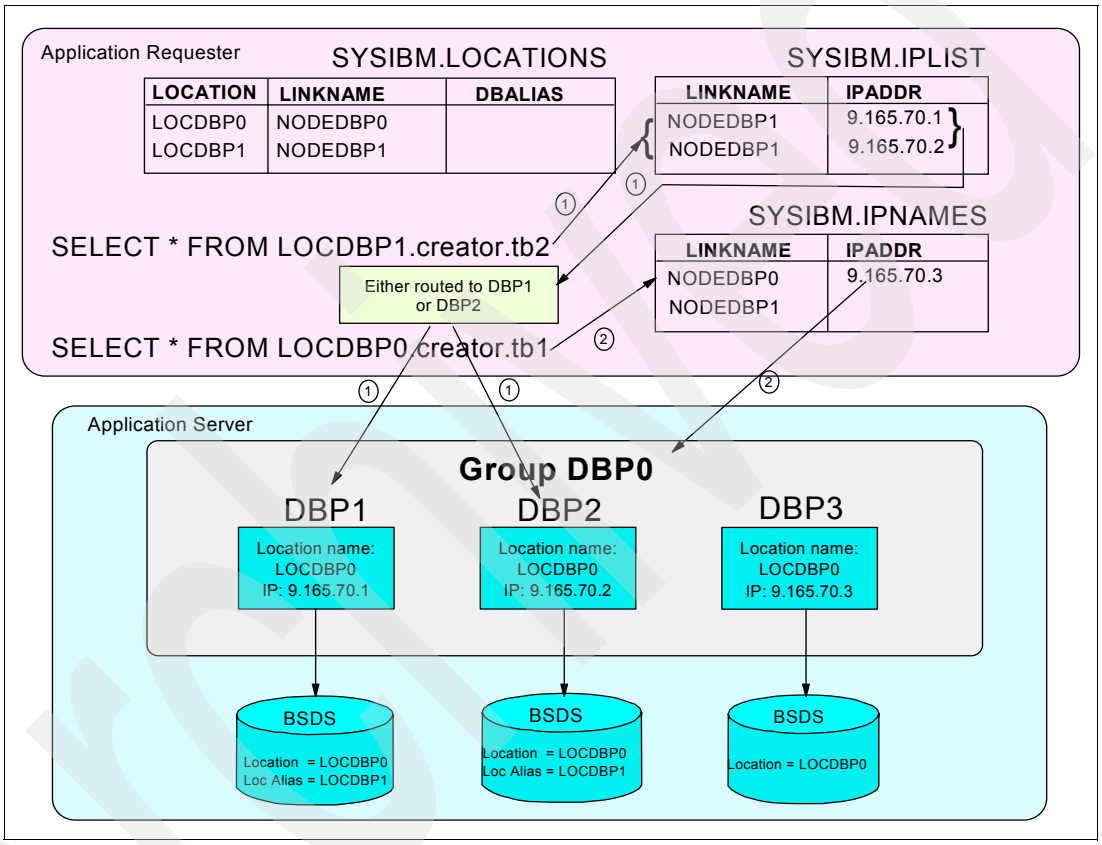


Figure 22-7 Member routing in a TCP/IP network

If a request comes in for location name LOCDBP0 [2], the entry in SYSIBM.SYSIPNAMES routes it to all available members in group DBP0.

When using WLM domain name server to perform workload balancing, the SYSIBM.IPLIST must contain the member specific domain name for each DB2 subsystem that requests are to be routed. When using dynamic VIPA to perform workload balancing, the IPLIST must contain the member specific dynamic VIPA for each DB2 subsystem whose requests are to be routed. Domain names and DVIPA cannot be defined in SYSIBM.IPLIST for the same DB2 data sharing group.

DB2 first checks the IPLIST table, and then the IPNAMES table. For member specific routing to NODEDBP1, if there is no entry in IPNAMES, or the entry for NODEDBP1 in IPNAMES is specified with an IPADDRESS, then SQLCODE -904 is issued.

22.2.4 RRSAF compatibility for CAF applications

The DB2 Call Attach Facility (CAF) is used by many applications today. Recoverable Resources Manager Services Attachment Facility (RRSAF) provides roughly the same functionality as CAF. However, RRSAF has a major advantage over CAF, and that is its ability to support two phase commit processing. Therefore, if your current CAF applications need two phase commit support, you have to migrate them to RRSAF.

To use RRSAF, you must link your programs with the RRSAF language interface module, DSNRLI instead of the CAF language interface DSNALI. For information on loading or linkediting this module, you can refer to *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

Explicit DB2 connections are established in almost the same way by CAF and RRSAF. For CAF applications, you must issue a CONNECT and OPEN. For RRSAF, an IDENTIFY and CREATE THREAD is necessary. You can then run your SQL statements. To disconnect, CAF uses a CLOSE and DISCONNECT, whereas RRSAF invokes a TERMINATE THREAD and TERMINATE IDENTIFY.

In contrast to this, CAF applications also have the possibility to connect implicitly to a DB2 subsystem, that is just by issuing SQL statements or IFI calls (without first doing a CONNECT and an OPEN). An implicit CAF connection derives the DB2 subsystem name it will connect to from the DSNHDECP module, and will allocate a plan that has the same name as the program (DBRM) that issues the first SQL call. With DB2 V7, implicit connections are not supported when you use RRSAF.

However, with DB2 V8 you can also connect to DB2 through RRSAF by just issuing SQL statements or IFI calls. When you request an implicit connection, RRSAF issues an IDENTIFY and CREATE THREAD under the covers, using the following values:

- **Subsystem name:** The default DB2 subsystem name specified in module DSNHDECP is used. The usual search order is used to find the DSNHDECP module; that is STEPLIB, JOBLIB, linklist. In a data sharing group, the default subsystem name is the group attachment name.
- **Plan name:** The member name of the database request module (DBRM) that DB2 produced when you precompiled the source program that contains the first SQL call. If your program can make its first SQL call from different modules with different DBRMs, then you cannot use a default plan name. You must use an explicit call using the CREATE THREAD function.
- **Authorization ID:** The authorization ID is set from the seven byte user ID associated with the address space, unless an authorized function has built an Accessor Environment Element (ACEE), which is a RACF control block for the address space. If an authorized function has built an ACEE, DB2 passes the 8 byte user ID from the ACEE.

22.2.5 Roll up accounting data for DDF and RRSAF threads

If you establish a connection to DB2 V7 through DDF, you normally want to use DB2's *type 2 inactive threads.* This function is also known as *thread pooling*. This feature is enabled by specifying CMSTAT=INACTIVE in DSNZPARM. Using the inactive thread support allows you to connect up to 150.000 users to a single DB2 subsystem. However, if you are running high volume OLTP workloads in this environment, you might encounter a performance bottleneck, because DB2 cuts an accounting record on every COMMIT or ROLLBACK when using thread pooling, and SMF might have a hard time to keep up with the massive number of DB2 accounting records that are produced.

You may encounter a similar problem when using the RRS attach, in combination with WebSphere. WebSphere drives the RRS signon interface on each new transaction, and DB2 cuts an accounting record when this happens. An accounting record is cut, even though some of these transactions contain just one SELECT statement followed by a COMMIT.

DB2 V8 adds a new installation option to activate the rollup of accounting information for DDF threads that become inactive, and RRS threads. The new option *DDF/RRSAF ACCUM* has been added to installation panel DSNTIPN The *default* is *NO*. The values accepted for this option range from 2 to 65535. The corresponding DSNZPARM is ACCUMACC.

When NO is specified, DB2 writes an accounting record when a DDF thread is made inactive, or when signon occurs for an RRSAF thread. If any number between 2 and 65535 is specified, DB2 writes an accounting record after every *n* occurrences of end user on any RRS or DDF thread, where *n* is the number specified for this parameter. An *end user* is identified by a combination of the following three values:

- End user user ID
- End user transaction name
- End user workstation name

Even when you specify a value between two and 65535, DB2 may choose to write an accounting record prior to the *nth* occurrence of the end user in the following cases:

- A storage threshold is reached for the accounting *rollup* blocks.
- You have specified accounting interval = COMMIT on the RRSAF signon call.
- When no updates have been made to the rollup block for 10 minutes, which is the user has not performed any activity for over 10 minutes that can be detected in accounting.

22.2.6 Improved query and result set processing

DB2 provides more flexibility for requestors such as DB2 Connect to specify larger query block sizes. This helps requestors to optimize their use of network resources. The DRDA protocol has been enhanced to allow for a maximum query block size of 10 MB, instead of the current 32 KB.

22.2.7 Time out for SNA allocate conversation requests

With DB2 V8, DDF searches every three minutes for threads waiting for a VTAM allocate conversation request to complete. When an allocate request has waited for a session for more than three minutes, DDF issues a deallocate abend conversation to VTAM to force VTAM to abnormally terminate the request. The remote SQL statement fails with a SQLCODE of -904 with reason code of 00D31033. This is an indicator that the network is down, and the network administrator should be notified of the communication failure.

22.2.8 Data stream encryption

A new connection encryption security mechanism is introduced in The Open Group Version 3 DRDA Technical Standard. In DB2 V, if connection encryption security mechanism is selected, then the userid, password and user data will be encrypted. This function:

- Provides the ability to encrypt and decrypt data as it is sent and received on the remote connection
- Is based on the technology used for password encryption
- Uses z/OS ICSF facilities with hardware assist

During connect processing, requester and server connection keys are exchanged and a shared connection key is generated. The connection keys are generated using the standard Diffie-Hellman distribution algorithm. Diffie-Hellman is the first standard public key algorithm ever invented. It gets its security from the difficulty of calculating discrete logarithms in a finite field. Diffie-Hellman requires three agreed upon values *n, g* such that *g* is a primitive of large prime *n* and the size of the exponent. The values for *n* and *g* are fixed. First, the application requester chooses a random large integer *x* and generates the value X where $X=g^x \bmod n$. *X* is the requester connection key. Second, the application server chooses another random large integer *y* and generates the value *Y* where $Y=g^y \bmod n$. *Y* is the server connection key. The application requester computes the shared private key, $k=Y^x \bmod n$. The application server computes the shared private key, $k1=X^y \bmod n$.

The 56-bit DES encryption key is generated from the shared private key. The Data Encryption Standard (DES), known as the Data Encryption Algorithm (DEA) by ANSI, and the DEA-1 by ISO is the worldwide standard for encrypting data. DES is a block cipher; it encrypts data in 64-bit blocks. DRDA encryption uses DES CBC mode as defined by the FIPS standard (FIPS PUB 81). DES CBC requires a 56-bit encryption key and an 8 byte token to encrypt the data. The Diffie-Hellman shared private key is 256 bits. To reduce the number of bits, 64 bits are selected from the connection key by selecting the middle 8 bytes, and parity is added to the lower order bit of each byte producing a 56-bit key with 8 bits of parity. The middle 8 bytes of the server's connection key is used as the token.

The user ID and password are encrypted at the requester. Once the user ID and password are decrypted, and the connection is authenticated, data streams exchanged between the requester and the server are encrypted using the same DES encryption key. The connection key is uniquely generated for each connection preventing the replay of data streams on different connection.

Connection encryption to a remote location

The communications database needs to be configured to enable connection encryption to a remote location. The SECURITY_OUT column in the IPNAMES that relates to the remote location must be updated with an "E" for connection encryption required. This column defines the security option that is used when local SQL applications connect to any remote server associated with this TCP/IP host.

Connection encryption from a remote location

No configuration is required to enable connection encryption from a remote location. During connect processing, the client negotiates the security mechanism for the connection. DB2 as a server will accept any connections requesting connection encryption.

Open Cryptographic Services Facility

Prior to V8, DB2 provided software support for the Diffie-Hellman algorithm and the DES decryption by loading in the required BSAFE services into the distributed address space. The encryption, decryption and D-H services were invoked directly by DDF. In V8, connection encryption will use The Open Cryptographic Services Facility (OCSF) to exploit any crytographic hardware installed on the processor. If properly configured, DB2 will utilize the IBM CCA Cryptographic Module and the Cryptographic Hardware feature instead of invoking the BSAFE services directly using the software-only support. Additionally, the OS/390 Integrated Cryptographic Service Facility (ICSF) must be installed, configured to run with the Cryptographic Hardware feature, and must be active. See the *OS/390 ICSF Administrator's Guide,* SC23- 3975, for more information.

22.2.9 DISPLAY LOCATION command

Prior to DB2 V3, the **DISPLAY LOCATION** command allowed no PARMs, and displayed all locations. Beginning with DB2 V3 and with SNA two-phase-commit support, the DISPLAY LOCATION command was enhanced to accept PARMs and also a *DETAIL* keyword. In order to provide simplified transition to the new option of this command, DB2 allowed a blank PARM to provide the same output as it did before the PARM was introduced. That is, **DISPLAY LOCATION()** would behave as if DISPLAY LOCATION was used and displayed all locations. Adding a PARM displayed only matching locations.

This behavior of the DISPLAY LOCATION with an empty PARM is different from the behavior of DISPLAY DATABASE with an empty PARM. DISPLAY DATABASE command parsing requires a value in the PARM. If you issue a **DISPLAY DB() SPACENAME()**, the command parser will fail the command with message DSN9010I.

In an effort to make all commands behave in a more predicable and consistent manner, beginning with DB2 V8, the semantics of DISPLAY LOCATION command with an empty PARM are changed. **DISPLAY LOCATION()** will now behave the same way as a **DISPLAY DATABASE() SPACE()** command, the command will fail with the message DSN9010I.

22.3 Enhancements for stored procedures and UDFs

DB2 V8 offers some improvements for stored procedures and user-defined functions (UDF). In this section we discuss these changes and how they might affect your daily work.

22.3.1 Maximum failures

With DB2 V7, you can specify a value for the maximum abend count on installation panel DSNTIPX (DSNZPARM parameter STORMXAB). With this value, you can specify the number of times a stored procedure or UDF is allowed to terminate abnormally before it is stopped. This parameter is subsystem wide, which means that you have to treat all stored procedures and UDF equally.

With DB2 V8, you can specify a value for each stored procedure or UDF. This means that you can now specify an appropriate value at the procedure level (instead of the subsystem level), for example, depending upon whether it is already an established application, or a new procedure being tested.

How to invoke this new functionality

The new functionality (syntax shown in Figure 22-8) is valid for external scalar and table functions, as well as external and SQL stored procedures.

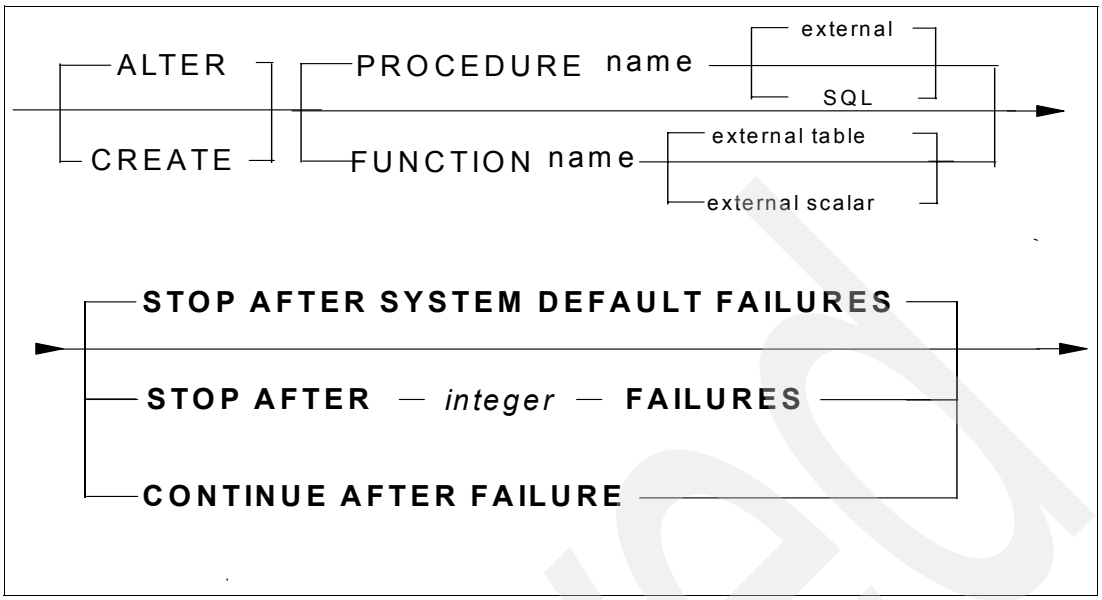


Figure 22-8 Stored procedure and UDF enhanced failure handling syntax

These parameters cannot be used for sourced functions and SQL scalar functions. These functions does not really have a “program” associated with them that runs in another address space. Therefore, the options have no relevance for them.

We now describe the new options in more detail:

- STOP AFTER *nn* FAILURES:

  This option puts the routine in a stopped state after *nn* failures. *nn* can be any value between 1 and 32767. That means it can either be higher or lower than the value specified for the DB2 subsystem (STORMXAB).

- STOP AFTER SYSTEM DEFAULT FAILURES:

  This option puts the routine in a stopped state when it reaches the number of abnormal terminations specified in the zparm value STORMXAB. This option is the default. Therefore, if you accept the default for STORMXAB (zero), and if you do not specify anything specific in your CREATE and ALTER PROCEDURE SQL statement, your stored procedure or UDF is stopped after every abnormal termination.

- CONTINUE AFTER FAILURE:

  If you decide to use this option, your stored procedures or UDFs are never put in stopped state, unless you explicitly use the -STOP PROCEDURE command.

To store this information, a new column MAX_FAILURE has been added to the SYSIBM.SYSROUTINES catalog table.

Attention: To activate these new settings, you must stop and start the corresponding stored procedure or UDF.

Monitoring the current status of your stored procedure or UDF

If you either specify STOP AFTER *nn* FAILURES or STOP AFTER SYSTEM DEFAULT FAILURES, it might be interesting to monitor the number of abnormal terminations that have already occurred. To satisfy your needs, the output of the DISPLAY PROCEDURE and

DISPLAY FUNCTION SPECIFIC command has been enhanced to show you the failure count for the procedure or function.

22.3.2 Exploit WLM server task thread management

The stored procedure management has been changed to exploit z/OS workload manager functions that allow z/OS's 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. Up to DB2 V7, specifying NUMTCB meant that a new WLM address space is started, if the number of TCBs running in one WLM managed stored procedure address space exceeded the value of NUMTCB. You can find more information regarding WLM application environments in conjunction with stored procedures in the redbook *Cross-Platform DB2 Stored Procedures: Building and Debugging,* SG24-5485.

This behavior is changed in DB2 V8. With V8, the value specified in NUMTCB is sent to WLM as a maximum task limit. WLM determines the actual number of TCBs that will run inside a WLM managed stored procedure address space. With memory available, we recommend that you use higher numbers in NUMTCB than you use in V7. This way WLM has more flexibility to decide how many tasks should run in one address space, and when to start an additional started task. NUMTCB can easily be around 60 or 30 if using the Java stored procedures.

This recommendation only applies for stored procedures, which are able to share the resources in one address space. There are still some stored procedures, for example DSNUTILS, which requires a value of 1 in NUMTCB.

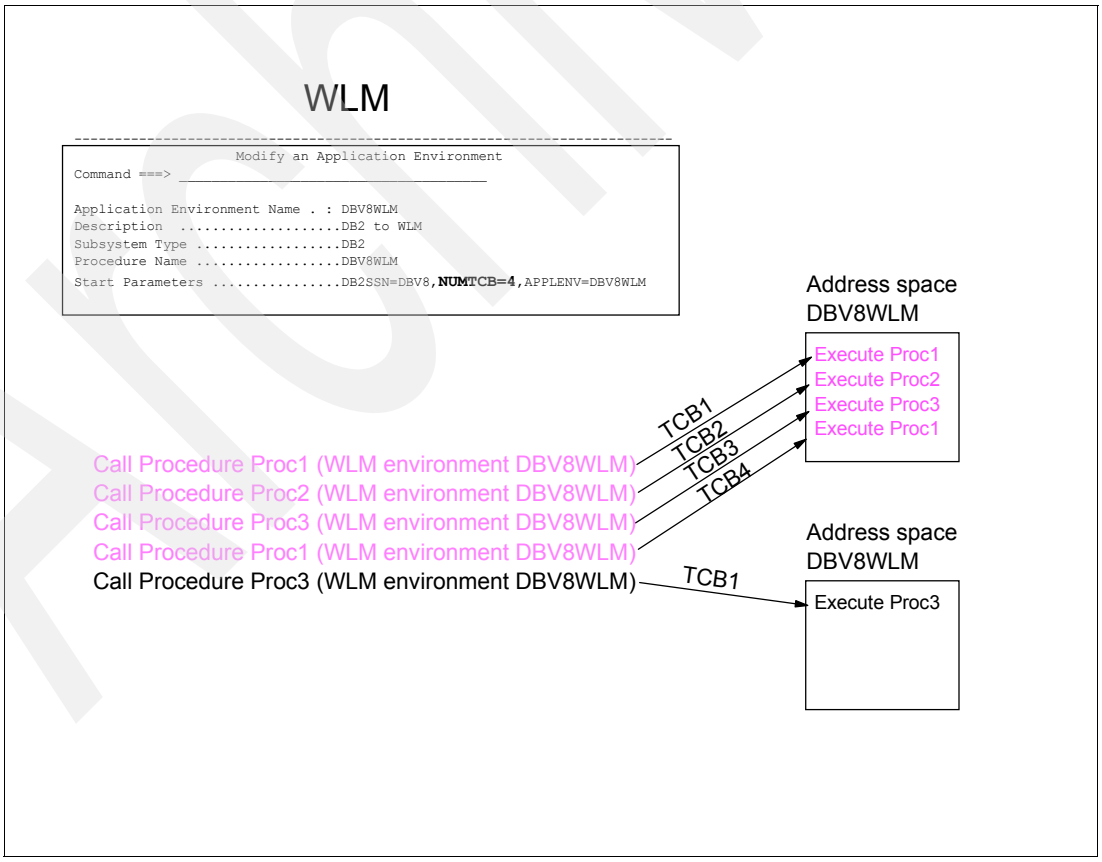


Figure 22-9 Address spaces with WLM

22.3.3 Nested stored procedure result sets for JDBC and ODBC applications

Up to V7 it is not possible for a JDBC or a CLI application to have more than one instance of an open result set cursor.

Figure 22-10 shows that an attempt to open a cursor that is already open from the previous call of the same procedure fails within DB2 UDB for OS/390 and z/OS. DB2 returns an error because the cursor is already open from the previous call.

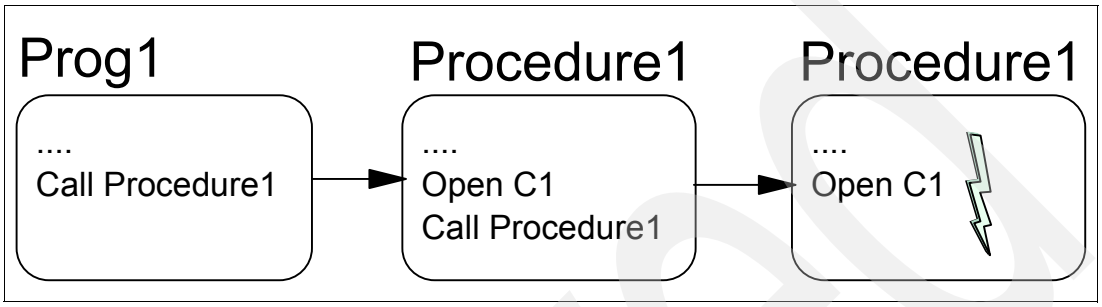


Figure 22-10 Nested stored procedure result sets

With the new Universal Java Client, as well as using ODBC/CLI applications on Linux, UNIX Windows, and z/OS, you can now have multiple instances of the same result set cursor open concurrently. This nesting of instances is possible for up to 16 instances. The DB2 server now provides a unique identifier to the requester for each open cursor or result set. The request can then manage the multiple instances using the unique cursor identifier.

22.3.4 Enhancements to SQL stored procedure language

The SQL stored procedure language and SQL stored procedures were introduced in DB2 UDB for OS/390 V5. The SQL Procedures language is a procedural language that is designed only for writing stored procedures. It is available across the entire DB2 family. The SQL Procedures language is based on SQL extensions, as defined by the Persistent Stored Modules (SQL/PSM) standard. The current implementation of the SQL Procedures language on DB2 UDB for OS/390 and z/OS does not cover everything that is described in this standard. In addition to that, there are also some differences regarding the available language elements within the entire DB2 family. In order to achieve a better compatibility between the different DB2 platforms, some new language elements have been added to DB2 V8.

RETURN

Currently, (up to V7) there are only two methods available for returning status information from an SQL procedure:

- Leave a condition unhandled

  Each condition that is not handled in a SQL procedure is returned to the caller in the SQLCA.

  Note: This behavior came in with APAR PQ56323. Prior to this change, SQL procedures for DB2 UDB for OS/390 did not return unhandled conditions to the calling application. This behavior is consistent with the rest of the DB2 family, and with the SQL standard.

- Define an extra parameter for status information

In this case, you would have to include an additional parameter (OUT or INOUT) for the status information as part of the parameter list.

Because both methods are not very convenient, support for the RETURN statement was added to SQL procedures. The RETURN statement can be used to return an integer value to the invoking application. Refer to Example 22-1 to see how this can be implemented.

Example 22-1 Using the RETURN statement

```
BEGIN
...
IF <failing condition> THEN
GOTO FAIL;
END IF;
...
SUCCESS: RETURN 0;
FAIL: RETURN -200;
END
```

SIGNAL

As described above, the RETURN statement allows you to pass back status information to the calling program of an SQL stored procedure. However, without the additional functionality of the SIGNAL SQL statement, the stored procedure cannot dictate what information is to be returned to the caller. To remove this restriction, the SIGNAL SQL statement can now be used for SQL stored procedures. The SIGNAL statement was already available for triggered actions of a trigger in DB2 V7.

The SIGNAL statement can be used to:

- Allow a procedure to set the SQLSTATE to a specific value
  - SQLCODE is set to +438 for an SQLSTATE class of '01' or '02'.
  - SQLCODE is set to -438 otherwise.
- Specify an optional MESSAGE_TEXT (1 KB maximum size)
  - The first 70 bytes of the message text is stored in the SQLERRMC field of the SQLCA.
  - The full message text can be obtained from the MESSAGE_TEXT field of GET DIAGNOSTICS.

The syntax for the SIGNAL SQL statement has changed from V7 to V8, as shown in Figure 22-11.

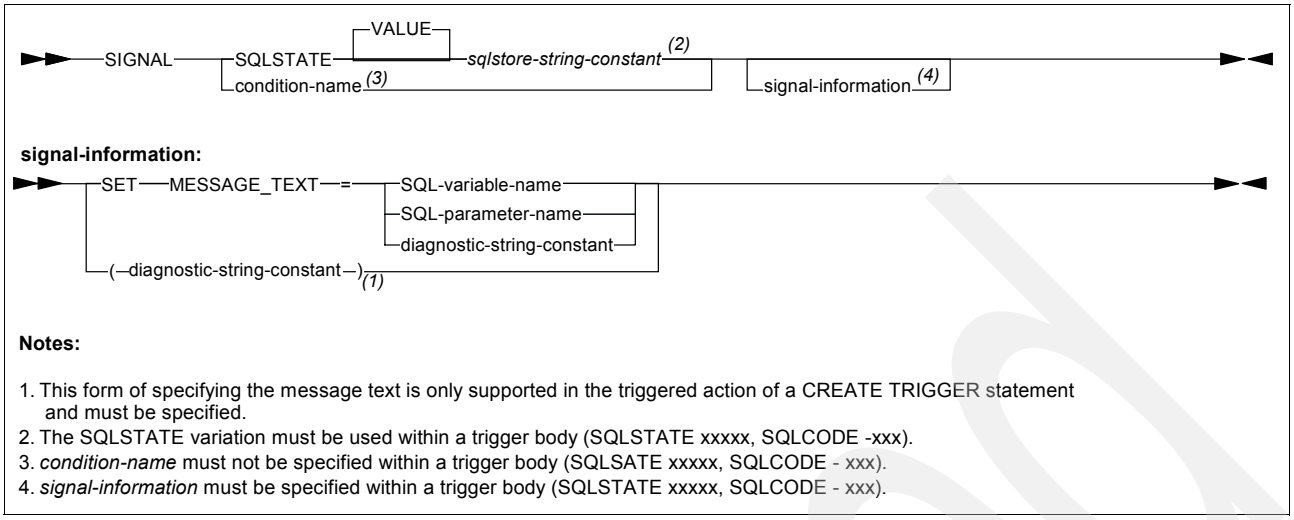


Figure 22-11 SIGNAL statement syntax diagram

You can use the SIGNAL keyword within a condition handler or anywhere else in the stored procedure body. Refer to Figure 22-12 to find out the differences in behavior, depending upon where the SIGNAL statement is specified.

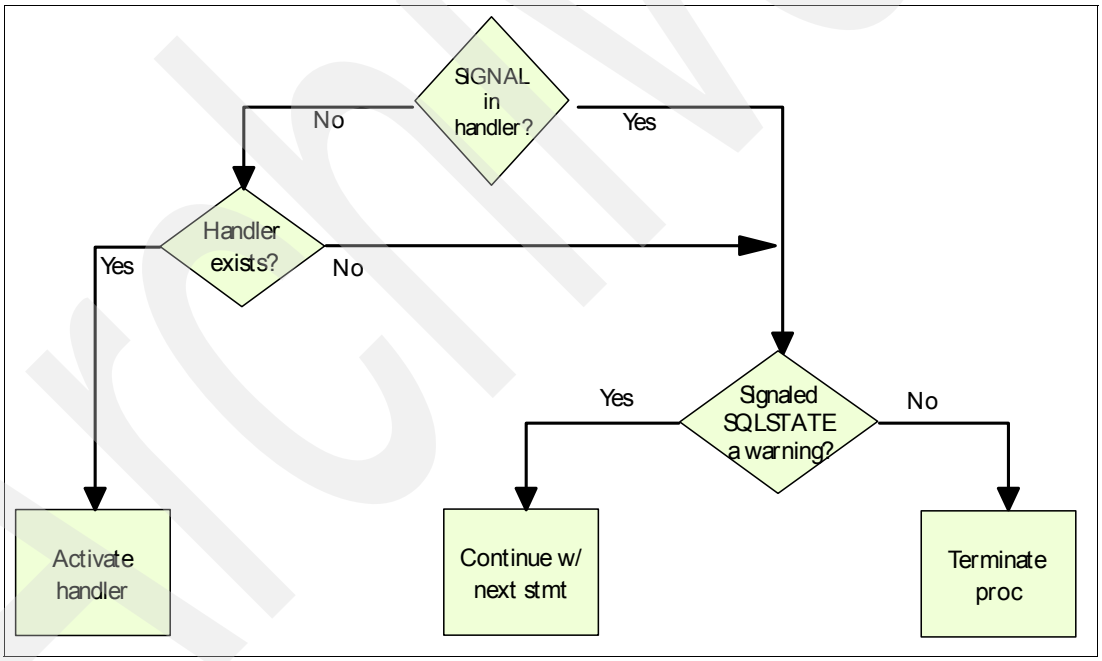


Figure 22-12 SIGNAL used in condition handler?

- If the SIGNAL is in the procedure body, but not part of a handler and a handler exists, the handler is activated.
- If the SIGNAL is in the procedure body, and there is no handler defined for this condition, then:
  - Continue if warning is signaled.
  - Exit if exception is signaled.
- If SIGNAL is part of a handler, then:
  - Continue if a warning is signaled.

– Exit if exception is signaled.

The CREATE PROCEDURE statements Example 22-2 shows the use of a SIGNAL statement which is part of a handler.

Example 22-2 Using the SIGNAL statement

```
CREATE PROCEDURE SUBMIT_ORDER
         (IN ONUM INTEGER, IN CNUM INTEGER,
          IN PNUM INTEGER, IN QNUM INTEGER)
         LANGUAGE SQL
         MODIFIES SQL DATA
         BEGIN
           DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
              SIGNAL SQLSTATE '75002'
                 SET MESSAGE_TEXT = 'Customer number is not known';
           INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
                       VALUES (ONUM, CNUM, PNUM, QNUM);
         END
```

If the stored procedure, during execution, encounters an SQLSTATE 23503 (which indicates that the insert or update value of a foreign key is invalid), the EXIT handler is invoked. As part of the EXIT handler, the SIGNAL statement signals SQLSTATE 75002 with message text Customer number is not known' instead. SQLSTATE 75002 is no predefined SQLSTATE used by DB2.

RESIGNAL

In contrast to the SIGNAL statement, the RESIGNAL statement is only used within a handler to return an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE along with optional message text. As for SIGNAL, the RESIGNAL statement sets SQLCODE to:

- +438 if SQLSTATE class is 01 or 02
- -438 otherwise

The syntax diagram for the RESIGNAL statement is provided for your reference in Figure 22-13.

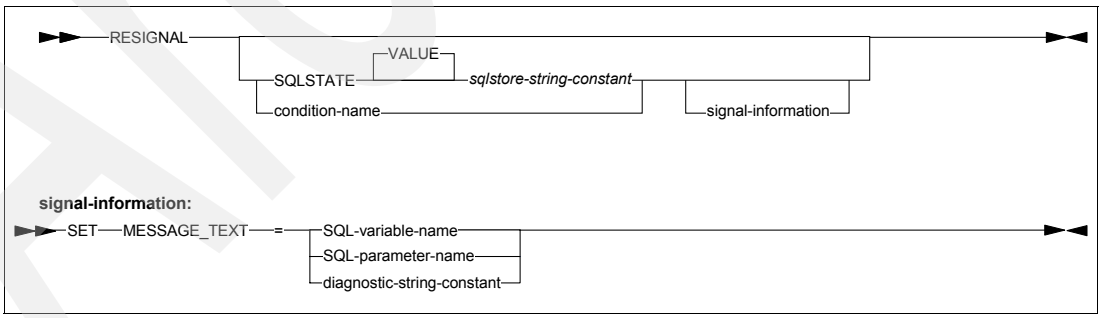


Figure 22-13 RESIGNAL statement syntax diagram

Example 22-3 shows how the RESIGNAL statement can be used.

Example 22-3 Using the RESIGNAL statement

```
CREATE PROCEDURE divide (  IN numerator INTEGER,
                           IN denominator INTEGER,
                           OUT divide_result INTEGER)
LANGUAGE SQL CONTAINS SQL
```

```
BEGIN
   DECLARE overflow CONDITION FOR SQLSTATE '22003';
   DECLARE EXIT HANDLER FOR overflow
      RESIGNAL SQLSTATE '22375';
   IF denominator = 0 THEN
      SIGNAL overflow;
   ELSE
      SET divide_result = numerator / denominator;
   END IF;
END
```

Referring to the example above, you can see that if the denominator equals to 0, the SIGNAL statement is used to signal a condition name, that is overflow in this example. Since we defined an EXIT handler for this overflow condition, the handler is fired now, that is the RESIGNAL statement assigns '22375' to SQLSTATE. SQLSTATE '22375' is not predefined and used by DB2. As you can see in Figure 22-13, you could have also added a message text as additional information.

Difference between the SIGNAL and RESIGNAL

Currently, the SIGNAL and RESIGNAL statements have only the following two superficial differences:

- RESIGNAL must be included in a handler, whereas SIGNAL can be included anywhere in an SQL stored procedure.
- You can specify RESIGNAL; by itself with no other syntax. You do not have to specify an SQLSTATE. (When you specify RESIGNAL without an SQLSTATE, DB2 uses the SQLSTATE that invoked the handler.) For SIGNAL, you must specify an SQLSTATE.

RESIGNAL as defined in the SQL standard has more functionality than is in DB2 Version 8.

SIGNAL/RESIGNAL versus RETURN

The RETURN statement enables a stored procedure to return only a single integer value to the caller, whereas the SIGNAL and RESIGNAL statements enable the stored procedure to pass more complex output. With SIGNAL and RESIGNAL, the stored procedure can return an SQLSTATE that you choose, along with optional message text.

Also, a RETURN statement always causes the procedure to terminate, whereas the SIGNAL and RESIGNAL statements do not cause the procedure to terminate when the SQLSTATE that is being signalled or re-signalled is a warning instead of an error.

GET DIAGNOSTICS

The GET DIAGNOSTICS statement has also been enhanced. You can now retrieve much more information than just the ROW_COUNT. The GET DIAGNOSTICS statement can also be used by any other programming language, not just by an SQL procedure as is the case in DB2 V7.

22.3.5 COMPJAVA stored procedures no longer supported

Visual Age for Java does no longer support compiled Java link library files. For this reason, DB2 V8 no longer supports stored procedures written in compiled Java. Therefore, you cannot use the keyword LANGUAGE COMPJAVA any more. Take the following steps to migrate a LANGUAGE COMPJAVA stored procedure to LANGUAGE JAVA:

1. Use ALTER PROCEDURE to change the LANGUAGE and the WLM ENVIRONMENT. The EXTERNAL NAME clause must also be specified, even if it is not changed. DB2 needs to verify it.

2. Make sure the WLM environment has been set up and the required JVM installed.
3. Make sure the .class file identified in the EXTERNAL NAME clause of the ALTER PROCEDURE is either contained in a JAR char has been installed to DB2 with an invocation of the INSTALL_JAR stored procedure, or that the .class file is in a directory name in the CLASSPATH ENVAR of the data set named on the JAVAENV DD card of the WLM stored procedures address space JCL.

22.3.6 DB2 established stored procedures

In DB2 V7 you have the choice between the DB2 managed, and WLM managed stored procedures. All DB2 managed stored procedures are executed in the same address space called *ssidSPAS*. The way to create a DB2 managed stored procedure is to specify NO WLM ENVIRONMENT in your CREATE or ALTER PROCEDURE statement.

WLM managed stored procedures provide many advantages, and because running WLM in goal mode is required in z/OS 1.3, and z/OS 1.3 is a prerequisite for DB2 V8, there are no reasons to continue using DB2 managed stored procedures. Therefore, DB2 V8 no longer allows the creation of DB2 managed stored procedures. This means that the keyword NO WLM ENVIRONMENT is no longer valid in the CREATE or ALTER PROCEDURE SQL statement.

For compatibility reasons, you can continue to use your existing DB2 managed stored procedures, but you should consider to convert them to WLM managed stored procedures to take advantage of the enhanced functionality described above.

22.4 CURRENT PACKAGE PATH special register

You cannot identify package resolution schemes for stored procedures and UDFs independently from the rules established by the caller's plan. Although the SET CURRENT PACKAGESET statement, or the COLLID option for a routine can be used within a procedure or function to change the package resolution rule from the invokers, these can only specify one collection. The new CURRENT PACKAGE PATH special register allows a stored procedure or user-defined function to be implemented without concern for the invoker's run-time environment, and also allows multiple collections to be specified.

CURRENT PACKAGE PATH specifies a VARCHAR(4096) value that identifies the path to be used to resolved references to packages that are used to execute SQL statements in the stored procedure as well as in any subprograms called by the stored procedure. The value can be an empty or blank string, or a list of one more collection IDs:

```
SET CURRENT PACKAGE PATH = DEVL7083, TEST7083, BETA7083, PROD7083;
```

The initial value of CURRENT PACKAGE PATH is an empty string.

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, for example:

```
COLLID DEVL7083
```

or, you may let it default by specifying:

```
NO COLLID
```

If you do specify NO COLLID, DB2 uses the following method to determine the collection ID *in this order:*

1. If the calling application has a package associated with it, DB2 uses its collection ID for the stored procedure.
2. For DB2 V8 onwards, DB2 examines the CURRENT PACKAGE PATH special register. If it contains a value, DB2 uses this as the collection ID for the stored procedure.
3. DB2 examines the CURRENT PACKAGESET special register. If it contains a value, DB2 uses this as the collection ID for the stored procedure.
4. DB2 examines the plan of the calling application and DB2 uses the list of collection IDs specified in the PKLIST in the specified order. This process is specially 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 till the package is found. SET CURRENT PACKAGESET eliminates this search.

22.5 New LOB parameters

In this section we discuss the new DB2_INSTALL_JAR and DB2_REPLACE_JAR stored procedures in V8 with LOB parms. These stored procedures were primarily for use by Development Center, and have been documented for general use.

The new stored procedures are:

- SQLJ.DB2_INSTALL_JAR

  Install the JAR in the AJAR BLOB parameter to DB2, so it can be used in the EXTERNAL NAME clause when creating a routine with LANGUAGE JAVA.

  The maximum size JAR that will be installed is 10 MB.

  Parameters are:

  - IN AJAR BLOB AS LOCATOR
  - IN JARNAME VARCHAR(257) CCSID EBCDIC
  - IN DEPLOY INTEGER

- SQLJ.DB2_REPLACE_JAR

  Replace the JAR stored in the DB2 catalog.

  The maximum size JAR that will be stored is 10 MB.

  Parameters are:

  - IN AJAR BLOB AS LOCATOR
  - IN JARNAME VARCHAR(257) CCSID EBCDIC

The key difference for these from the existing SQLJ.INSTALL_JAR and SQLJ.REPLACE_JAR is that they take a BLOB as a parameter instead of a file URL.

This makes it feasible to create a Java routine on a remote workstation, and invoke USS to install it with a JAR file as a parameter. The current stored procedures require that the JAR file reside in the local HFS.

23

Developing multi-threaded stored procedures in C language

In this chapter we explore the possibility of multi-threading within 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 that have 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
- Improvements
- Common design problems using multiple threads

23.1 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 of 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. This is particularly a benefit for J2EE™ applications that do not allow the use of threads in Enterprise Java Beans.
- 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. You have to explicitly establish a database connection using RRSAF from every thread that has to execute SQL statements.

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 Applications” of *z/OS C/C++ Programming Guide*, SC09-4765-03.

23.2 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.

23.3 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 23-1 shows the definition of the created global temporary table in which the calling application inserts the names of the table spaces.

Example 23-1 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 the Example 23-2.

Example 23-2 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 run-time 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 23-3 shows the definition of that table.

Example 23-3 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 23-4) 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 spaces names is inserted into the table. After that, RUNSTATP is called to run RUNSTATS on them in parallel.

Example 23-4 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 23-5 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 if the highest DSNUTILS return code was greater than 4.

Example 23-5 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 127 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 23-6.

Example 23-6 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 23-7.

Example 23-7 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.DSNACCSS..."
 #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 23-8 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 23-8 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 23-9.

Example 23-9 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 DSNACCSS to find out the SSID, we need to declare variables for it as shown in Example 23-10.

Example 23-10 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;
   /* DSNACCSS 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 a 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 23-11 for the cursor declarations.

Example 23-11 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 23-12. 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 23-12 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 properly the required THREAD_ PARM variables. See Example 23-13.

Example 23-13 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 DSNACCSS to determine the SSID of the subsystem we are connected to with the code shown in Example 23-14. 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 DSNACCMD.

Example 23-14 Determining the subsystem ID

```
/*******************************************************************/
   /* Determine the current subsystem ID for RRSAF connection.        */
   /*******************************************************************/
   if (rc < RETSEV)
   {
     EXEC SQL CALL SYSPROC.DSNACCSS(: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 23-15. We need that thread ID to later synchronize with the thread.

Example 23-15 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 23-16 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 23-16 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 23-17.

Example 23-17 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 127. You can download the complete source code from the Web as additional material. Download instructions can be found in Appendix D, “Additional material” on page 651.

Next, we look at the function that runs DSNUTILS in a secondary thread. This is shown in Example 23-18. Like the main thread, it has its own sqlca SQL parameters it requires to call DSNUTILS declared.

Example 23-18 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 23-19. 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 23-19 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 23-20) to establish the task as a user of the DB2 subsystem.

Example 23-20 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 23-21 RRS SIGNON

```
/* wait for termination ECB be posted */
unsigned int termecb;            /* termination Event Control Block */
selectex(0, NULL, NULL, NULL, NULL, (int *)&termecb);

  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 23-22.

Example 23-22 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 23-23.

Example 23-23 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 23-24.

Example 23-24 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 orderly disconnect from the subsystem and leave the utility thread as shown in Example 24-25.

Example 23-25 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);
}
```

23.4 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 23-26.

Example 23-26 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
/*
```

23.5 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 not incur the costs of the RRSAF connect and disconnect every time. 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 on how to use mutex objects and condition variables can be found in the *z/OS C/C++ Programming Guide*, SC09-4765-03 and *z/OS C/C++ Run-Time Library Reference.*, SA22-7821.

23.6 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 23-27.

Example 23-27 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 the 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.

24

Accessing CICS and IMS

You may have legacy applications that access data, which 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, instead 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 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 14, “Debugging” on page 173, and Chapter 28, “Tools for debugging DB2 stored procedures” on page 463 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.

Accessing CICS and IMS from DB2 stored procedures was previously described in the redbook *Cross-Platform DB2 Stored Procedures: Building and Debugging*, SG24-5485-01. There have been some enhancements in this area since the publication of that redbook, including the introduction of some DB2-supplied stored procedures to access non-DB2 data. This chapter provides an update to the topics discussed in SG24-5485-01. Refer to that redbook for the prerequisite information and more details on these topics.

Note: Complete sample programs can be downloaded from the ITSO Web site as additional material. Download instructions can be found in Appendix D, "Additional material" on page 651.

Before downloading, we strongly suggest that you first read 3.3, "Sample application components" on page 22 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

24.1 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.

24.1.1 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 or OS/390 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 External Interfaces Guide*, SC34-6006-05.

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 24-11 on page 395. 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:

- Prepare 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. Detailed screen prints of sample CEDA entries for an EXCI application are provided in *Cross-Platform DB2 Stored Procedures: Building and Debugging*, SG24-5485-01. The examples in SG24-5485-01

do not include accessing a VSAM file, so we modified our case study to include accessing a VSAM file. The JCL in Example 24-11 on page 395 defines the VSAM data set for both CICS and IMS examples. 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.

The RDO definition H:

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: I)
   – 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**

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

Once the resource definitions have been made and the CICS program has been prepared, you need to refresh the load module for the CICS program by issuing a CEMT command. Issue the following command in CICS to pull in the latest version of program EMPEXC2C:

CEMT INQ PROG(EMPEXC2C)

The results of the CEMT command are shown in Example 24-1.

Example 24-1 CEMT command used to refresh a CICS program

```
INQ PROG(EMPEXC2C)
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Prog(EMPEXC2C) Leng(0000005328) Cob Pro Ena Pri     Ced
    Res(000) Use(0000000009) Bel Uex Ful Qua
```

To pull in the new copy of the module in CICS enter NEW in the spaces between **Pri** and **Ced**.

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 DB2GDEC2, which represents a second COBOL environment. The load library that contains the EXCI stub program DFHMIRS needs to be included in the startup JCL for the WLM proc. The name of the load library typically ends in SDFHEXCI. A sample load module name is provided in your SDSNSAMP library in member DSNTIJCI.

Once the WLM environment has been defined you need to refresh the WLM environment by issuing a vary WLM refresh command. For our EXCI environment, DB2GDEC2, we issued the following command:

/V WLM,APPLENV=DB2GDEC2,REFRESH

Coding a stored procedure to use EXCI

Stored procedure EMPEXC1C includes an EXEC CICS LINK statement to invoke the new transid defined in the RDO entry as described in the section above. Example 24-2 shows a sample LINK statement.

Example 24-2 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 will load 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 are 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 24-3.

Example 24-3 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 CICSTS22.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. A sample CICS translation step is shown in Chapter 7 of *Cross-Platform DB2 Stored Procedures: Building and Debugging*, SG24-5485-01. The CICS program needs to be prepared using standard CICS preparation JCL.

24.1.2 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 389), 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. The DDL to create the procedure is located in member DSNTIJCI of the SDSNSAMP library. A sample CREATE PROCEDURE statement is shown in Example 24-4.

Example 24-4 DDL to create sample stored procedure DSNACICS

```
CREATE PROCEDURE SYSPROC.DSNACICS
  ( IN PARM_LEVEL         INTEGER
  , IN PGM_NAME           CHAR(8)        CCSID EBCDIC
  , IN CICS_APPLID        CHAR(8)        CCSID EBCDIC
  , IN CICS_LEVEL         INTEGER
  , IN CONNECT_TYPE       CHAR(8)        CCSID EBCDIC
  , IN NETNAME            CHAR(8)        CCSID EBCDIC
  , IN MIRROR_TRANS       CHAR(4)        CCSID EBCDIC
  , INOUT COMMAREA        VARCHAR(32704) CCSID EBCDIC
  , IN COMMAREA_TOTAL_LEN INTEGER
  , IN SYNC_OPTS          INTEGER
  , OUT RETURN_CODE       INTEGER
  , OUT MSG_AREA          VARCHAR(500)   CCSID EBCDIC )
```

```
  EXTERNAL NAME DSNACICS
  LANGUAGE ASSEMBLE
  WLM ENVIRONMENT !WLMENV!
  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. See member DSNTIJCI in the SDSNSAMP library for details on installing DSNACICS.

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 389). 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 24-5. The parameters for the CALL statement are described in detail in Appendix H of *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413.

Example 24-5 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 describe the parameters that are pertinent for our test case below. 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 SCSCPAPB.
- 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.
- 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.

24.2 Accessing IMS databases from DB2 stored procedures

There are two alternative methods for accessing IMS databases from a DB2 stored procedure: the IMS Open Database Access interface (ODBA) and stored procedure DSNAIMS. The ODBA interface provides the capability to include IMS database calls within your DB2 stored procedure. The DB2-supplied stored procedure DSNAIMS provides 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 from DB2.

24.2.1 Accessing IMS databases through ODBA interface

The IMS Open Database Access interface (ODBA) provides access to IMS databases from a z/OS or OS/390 application, such as a DB2 stored procedure. Your stored procedure can issue a call using the AERTDLI API and pass a DL/I status 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 status 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 of 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

The following setup steps were made 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 proc 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 24-6 lists the macros we defined in our IMS.

Example 24-6 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 24-7 provides the DBDGEN source and execution proc to create our DEPT database.

Example 24-7 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 24-8 provides the source for the Load PSB, DEPTPSB.

Example 24-8 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 24-9 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 24-9 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 24-10 defines our ACBs to IMS for our DEPT example.

Example 24-10 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 24-11 defines the VSAM data set for our IMS DEPT database.

Example 24-11 IDCAMS defines for DEPT VSAM data set

```
//ALLOCATE EXEC PGM=IDCAMS,DYNAMNBR=200
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
   DEFINE CLUSTER(                          -
                  NAME(IMS710G.DEPTDB1)     -
                  NONINDEXED                -
                  FREESPACE(10 10)          -
                  RECORDSIZE(2041 2041)     -
                  SHAREOPTIONS(3 3)         -
                  UNIQUE                    -
                  VOLUMES(TOTIM1)           -
                  CYLINDERS(02)             -
                  CONTROLINTERVALSIZE(2048) -
                 )                          -
              DATA(                         -
                  NAME(IMS710G.DEPTDB1.DATA) -
                 )
```

Define the source and run the dynamic allocation job for the database

The JCL and source in Example 24-12 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 24-12 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=IMS710G.DEPTDB1,                                 X
               DISP=SHR
  DFSMDA TYPE=FINAL
         END
```

Define and run the DBRC registration for the DEPT database

The JCL and source in Example 24-13 register the DEPTDB database in the DBRC recon data sets.

Example 24-13 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(IMS710G.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 24-14.

Example 24-14 Load JCL and data for DEPT database

```
//DLT0   PROC MBR=DFSDDLT0,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=IMS710G.SDFSRESL,DISP=SHR
//         DD DSN=IMS710G.PGMLIB,DISP=SHR
//IMS      DD DSN=IMS710G.PSBLIB,DISP=(SHR,PASS)
//         DD DSN=IMS710G.DBDLIB,DISP=(SHR,PASS)
//DFSRESLB DD DSN=IMS710G.SDFSRESL,DISP=SHR
//IEFRDER  DD DUMMY
// PEND
//DLT0 EXEC DLT0
//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 proc that executes our IMS ODBA stored procedure. See Example 24-15.

Example 24-15 DFSPRP macro that creates the DRA

```
DFSIMSG0 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 24-16 assembled our DFSPRP macro.

Example 24-16 Assembly JCL for the DFSPRP macro

```
//ASSEM EXEC HLASMCL
//C.SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//           DD DSN=IMS710G.SDFSMAC,DISP=SHR
//C.SYSIN    DD DSN=SG247083.ODBA.CNTL(DFSIMSG0),DISP=SHR
//L.SYSLMOD  DD DSN=IMS710G.SDFSRESL(DFSIMSG0),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 online change for MODBLKS and ACBLIB. The online change JCL in Example 24-17 was used.

Example 24-17 IMS online change input

```
// JCLLIB ORDER=IMS710G.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 24-18.

Example 24-18 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 24-19 for executing our DB2 COBOL ODBA stored procedure.

Example 24-19 WLM environment for our DB2 COBOL ODBA case study

```
Appl Environment Name . . DB2GODBA
Description . . . . . . . DB2G IMS-ODBA- SG247083 - devl
```

```
Subsystem type  . . . . . DB2
Procedure name  . . . . . DB2GODBA
Start parameters  . . . . DB2SSN=&IWMSSNM,APPLENV=DB2GODBA
```

Define the associated WLM proc for the WLM environment

The WLM proc needs to include the STEPLIB data set we assembled our DFSPRP macro into, which in our case was IMS710G.SDFSRESL. The proc also needs to include the //DFSRESLB DD statement. Additionally, we included data sets for the Distributed Debugger. See Example 24-20.

Example 24-20 WLM proc for executing our DB2 COBOL stored procedure.

```
//STEPLIB  DD    DSN=SG247083.DEVL.LOAD,DISP=SHR
//         DD    DISP=SHR,DSN=DB2G7.SDSNEXIT
//         DD    DISP=SHR,DSN=DB2G7.SDSNLOAD
//         DD    DISP=SHR,DSN=IMS710G.SDFSRESL
//         DD    DISP=SHR,DSN=IMS710G.PGMLIB
//         DD    DISP=SHR,DSN=EQAW.SEQAMOD
//SYSTCPD  DD    DSN=TCP.SC63.TCPPARMS(TCPDATA),DISP=SHR
//CEEDUMP  DD    SYSOUT=*
//DFSRESLB DD  DISP=SHR,DSN=IMS710G.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 24-21. The parameter APSB contains the value APSB, which is what is required to request allocation of a PSB.

Example 24-21 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 24-22. 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 24-22 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 24-23. 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 24-23 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 D, “Additional material” on page 651.

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 24-24 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 24-24 Sample link edit step for stored procedure with ODBA call

```
//LKED.SYSLMOD DD DSN=SG247083.DEVL.LOAD(EMPODB1C),
//              DISP=SHR
//LKED.LOAD DD DSN=IMS710G.SDFSRESL,
//              DISP=SHR
//LKED.SYSIN   DD *
     INCLUDE SYSLIB(DSNRLI)
     INCLUDE LOAD(DFSCDLI0)
//*-----------------------------------------------
```

24.2.2 Accessing IMS databases through stored procedure DSNAIMS

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), currently under development as of the writing of this redbook, will allow 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 beyond. PTFs UQ96684 (DB2 V7) and UQ96685 (DB2 V8) for APAR PQ77702 deliver DSNAIMS. This routine, written in C language, provides comparable functionality for an IMS transaction environment to what DSNACICS provides for a CICS transaction environment. For more details see Appendix C, "DSNAIMS stored procedure" on page 643. The DSNAIMS parameters will be described in the DB2 documentation, specifically the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02.

A sample call statement is shown in Example 24-25.

Example 24-25 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-CODEX
                              )
      END-EXEC.
```

DSNAIMS prerequisites

The following functions are required before installing and executing the DSNAIMS stored procedure:

- DB2 Version 7 or above with RRSAF enabled
- IMS Version 7 or above with OTMA and Callable Interface enabled
- DSNAIMS must run in a WLM-Managed stored procedure address space
- DB2 Version 7 requires PTFs UQ66553 and UQ94695 respectively for PQ44819 and PQ89544
- DB2 Version 8 requires PTF UQ94696 for PQ89544
- APAR PK04339, currently open, delivers a solution for message truncation caused by X'00' in the input variable string.

24.3 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 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 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 24-26 shows what a call to stored procedure EMPRSETC from a COBOL CICS application might look like.

Example 24-26 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.

24.4 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 24-27 shows what a call to stored procedure EMPRSETC from a COBOL IMS application might look like.

Example 24-27 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.

25

DB2-supplied stored procedures

DB2 provides several stored procedures that you can call in your application programs to perform database and system administration functions. The stored procedures that are installed with the DB2 server including DSNUTILS, DSNWZP, and WLM_REFRESH are well known and documented. In this chapter, we provide an overview of all the stored procedures supplied by DB2, and document the ones that you cannot find in the DB2 for z/OS books. We show how to install and activate them, and how to use them with comprehensive sample programs written in Java that show how to process results and handle errors correctly.

All of this will enable 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
- Installing and activating the DB2-supplied stored procedures
- z/OS Enablement stored procedures
- Using the DB2 provided stored procedures
- Summary

25.1 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. These stored procedures include DSNWZP, DSNUTILS, and WLM_REFRESH. They are well documented in the DB2 for z/OS books.

And there are numbers of very useful stored procedures shipped with the optional and SMP/E installable DB2 Management Clients package. These procedures are not documented because they were developed and are used to provide server-side functionality to the DB2 UDB Control Center. The trend of moving DB2 subsystem management functions to stored procedures appears to be general and strategic within the entire DB2 family.

Even though currently most of the DB2 *administration* functionality is available through documented C APIs on DB2 for Linux, UNIX and Windows, this hinders the development of portable and plug-in Java application layers on top of DB2 administration capabilities especially for DB2 for z/OS where these C APIs are not available. It is natural to have common SQL interfaces to administration functions and ease Java application integration through JDBC or SQLJ.

Many application environments use the context of a single connection to drive all access to their databases. It is much easier and seamless to use SQL routines to perform the administration functions. SQL APIs are already used by most DB vendors.

A winning strategy is to provide standardized SQL access to administration functions in the form of procedures and table functions. Such an access ensures a base standard infrastructure that satisfies customer requirements and enables IBM to build on other application layers such as Java administration APIs, Web Services, etc.

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 for application vendors, such as SAP, who increasingly leverage the advantages of Web applications running in application servers like WebSphere Application Server (WAS). The DB2-supplied stored procedures provide that infrastructure.

The intent of this chapter is to show that most of the DB2 administration functions can easily be executed through DB2-supplied stored procedures.

25.1.1 DB2 Control Center

The DB2 Control Center is part of the DB2 Management Clients package, and it is a DB2 database and system administration tool with a sophisticated graphical user interface (GUI).

In addition it provides utility management functionality such as object maintenance automation and advanced system administration functions such as subsystem cloning (Homogenous System Copy). It is targeted for application developers, and database and system administrators who prefer using a Windows GUI over an ISPF GUI.

With DB2 for OS/390 and z/OS V7, the DB2 Management Clients Package includes DB2 Connect Personal Edition (which includes the Control Center), DB2 Installer, Visual Explain, and DB2 Estimator on CD-ROM, which can be installed on a workstation. The SMP/E installable package with FMID JDB771D is called and referred to as 390 Enablement.

With DB2 for z/OS V8, the DB2 Management Clients Package includes the DB2 Administration Server for z/OS (SMP/E installable FMID HDAS810), z/OS Enablement

(SMP/E installable package with FMID JDB881D), DB2 for z/OS Visual Explain, DB2 Connect Personal Edition.

The z/OS Enablement was initially developed to provide server-side functionality to the DB2 UDB Control Center, which supported DB2 for OS/390 Version 5 servers starting with version 5 of DB2 Connect. Today, the stored procedures and user defined functions of the z/OS Enablement package are also used by other DB2 Administration Tools including the DB2 Replication Center and SAP's Computing Center Management System (CCMS). With CCMS, a SAP administrator can monitor all servers, components, and resources in his SAP landscape from one single centralized server, facilitating problem discovery and problem diagnosis.

25.1.2 Summary information on DB2-supplied stored procedures

In this section we list stored procedures including their name, function, FMID, or required service and activation job, grouped by the function they perform on the connected DB2 subsystem or LPAR in which the subsystem runs. When maintenance (a PTF number) is specified, it means that this maintenance level is the minimum level required or recommended to use these stored procedures.

Table 25-1 lists the current FMIDs referred to in this chapter.

Table 25-1 FMIDs

DB2 Version	DB2 Server Base FMID	z/OS Enablement FMID
6	HDB6610	JDB661D
7	HDB7710	JDB771D
8	HDB8810	JDB881D

DB2 versions prior to Version 6 have been withdrawn from service. We only discuss versions 6 and higher.

Table 25-2 lists the stored procedures available for DB2 system administration, with a short description, and the job that activates them.

Table 25-2 Stored procedures for DB2 system administration

Name	Function	FMID and maintenance	Activation job
DSNWZP	Displays the current settings of system parameters	HDB6610 HDB7710 HDB8810	DSNTIJSG
WLM_REFRESH	Refreshes a WLM environment	HDB7710 HDB8810	DSNTIJSG
DSNACCSS	Queries the SSID	JDB661D UQ71136 JDB771D UQ71135 HDB7710 UQ71134 JDB881D	DSNTIJCC
DSNACCSI	Queries the fully qualified TCP/IP domain name	JDB661D UQ71136 JDB771D UQ71135 HDB7710 UQ71134 JDB881D	DSNTIJCC

Name	Function	FMID and maintenance	Activation job
DSNAICUG	Gets list of users and groups defined in USS	JDB661D UQ71136 JDB771D UQ71135 HDB7710 UQ71134 JDB881D	DSNTIJCC

Table 25-3 lists the stored procedures available for DB2 database administration.

Table 25-3 Stored procedures for DB2 database administration

Name	Function	FMID and maintenance	Activation job
DSNACCOR	Makes recommendations for object maintenance	HDB7710 HDB8810	DSNTIJSG
DSNUTILS	Execute DB2 online utilities	HDB6610 UQ42711 HDB7710 HDB8810	DSNTIJSG
DSNUTILU	Executes DB2 online utilities and accepts parameters	HDB8810	DSNTIJSG
DSNACCMO	Executes DB2 online utilities in parallel using an optimized scheduler	JDB661D UQ71136 HDB7710 UQ71134 JDB771D PQ75973*) JDB881D PQ75973*) * PTF not available at time of publishing	DSNTIJCC
DSNACCMD	Executes DB2 commands	JDB661D UQ71136 HDB7710 UQ56630 JDB771D UQ56631 HDB8810 UQ80977	DSNTIJCC

Note that the table space and index information stored procedure DSNACCQC, and the partition information stored procedure DSNACCAV are still part of z/OS Enablement for compatibility with older versions of DB2 Control Center and CCMS. We do not recommend using them for new applications, as they will be removed from the z/OS Enablement package in the future. We recommend using DSNACCOR instead, which provides all of the functionality previously provided by DSNACCAV and DSNACCQC, and performs better. DSNACCOR uses real-time statistics to make recommendations. You can find detailed information on how to use DSNACCOR in the *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

Table 25-4 lists the stored procedures available for data set manipulation.

Table 25-4 Stored procedures for data set manipulation

Name	Function	FMID and maintenance	Activation job
DSNACCDS	Creates or writes a data set	JDB661D UQ76360 HDB7710 UQ76358 JDB771D UQ76359 JDB881D UQ76361	DSNTIJCC

Name	Function	FMID and maintenance	Activation job
DSNACCDR	Renames a data set	JDB661D UQ43116, UQ76360 HDB7710 UQ76358 JDB771D UQ76359 JDB881D UQ76361	DSNTIJCC
DSNACCDD	Deletes a data set	JDB661D UQ43116, UQ76360 HDB7710 UQ76358 JDB771D UQ76359 JDB881D UQ76361	DSNTIJCC
DSNACCDE	Checks if a data set exists	JDB661D UQ76360 HDB7710 UQ76358 JDB771D UQ76359 JDB881D UQ76361	DSNTIJCC
DSNACCDL	Queries data set properties	JDB661D UQ43116, UQ76360 HDB7710 UQ76358 JDB771D UQ76359 JDB881D UQ76361	DSNTIJCC

Table 25-5 lists the stored procedures available for submitting JCL and UNIX System Services commands.

Table 25-5 Stored procedures for submitting JCL and USS commands

Name	Function	FMID and maintenance	Activation job
DSNACCJS	Submits a JCL job	JDB881D UQ81110	DSNTIJCC
DSNACCJF	Fetches output of a JCL job	JDB881D UQ81110	DSNTIJCC
DSNACCJP	Purges a JCL job	JDB881D PQ84480	DSNTIJCC
DSNACCJQ	Retrieves JCL job status	JDB881D PQ84480	DSNTIJCC
DSNACCUC	Issues USS commands	JDB881D UQ81110	DSNTIJCC

When the required FMIDs are installed and activated, you can invoke all these stored procedures from a client application program. In the next chapter, we show how to activate these DB2 provided stored procedures.

25.2 Installing and activating the DB2-supplied stored procedures

The roadmap in Figure 25-1 guides you through installing, activating, and verifying the DB2-supplied stored procedures. We use DB2 Control Center to verify the activation.

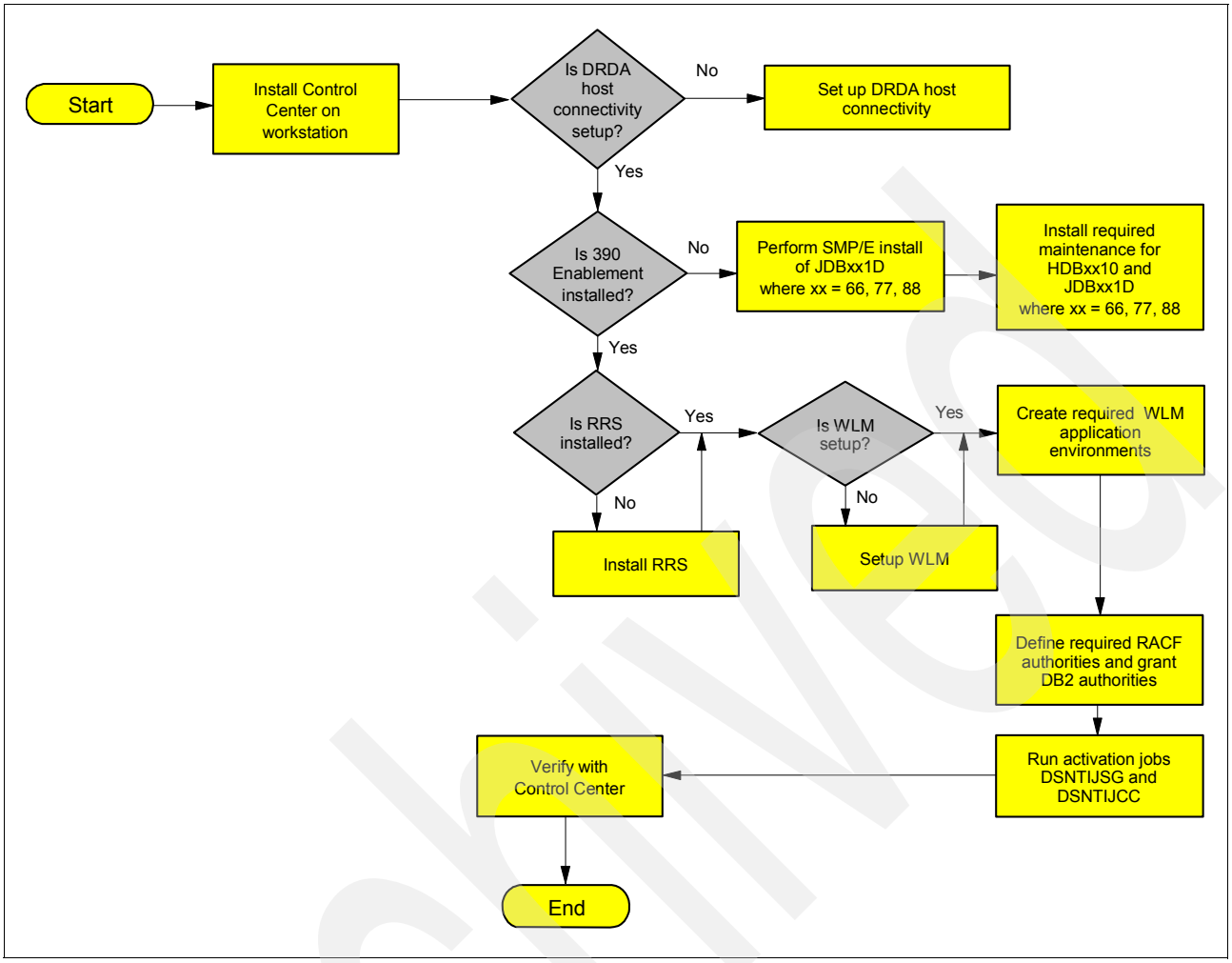


Figure 25-1 Roadmap to DB2-supplied stored procedures

Step1: Install the DB2 Control Center on the workstation

Install any DB2 UDB Version 8.1 (or higher) edition for your client platform (Linux or Windows) that includes the following features such as DB2 Connect Personal Edition:

Application Development Tools

- Client support
- Server support
- Administration tools

Make sure that all of these features are installed completely.

Step 2: 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. You can use the Configuration Assistant to catalog the DB2 subsystem, or you can enter the following commands from a DB2 command window:

```
db2 catalog tcpip node <node> remote <IP address> server <port> ostype mvs
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; or, for DB2 Version 6, look for the DSNL004I message that is printed to the log when DDF is started. In our tests we used the following commands to catalog the test system:

```
db2 catalog tcpip node TCPO001 remote wtsc63.itso.ibm.com server 12345 ostype mvs
db2 catalog dcs database DB8A as DB8A
db2 catalog database DB8A as DB8A at node TCPO001 authentication DCS
```

Verify that your DB2 subsystem is cataloged correctly by connecting to it. Enter the following commands from a DB2 command window:

```
db2 connect to <db alias> user <user> using <password>
```

You should see a message like the one below with the version, release, and modification number of your DB2 subsystem:

```
Database Connection Information

 Database server        = DB2 OS/390 8.1.5
 SQL authorization ID   = PAOLOR10
 Local database alias   = DB8A
```

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 grant public
db2 bind @db2cli.lst blocking all 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 the *DB2 Connect User's Guide*.

Step 3: Perform SMP/E install of JDBxx1D

Follow the instructions in the Program Directory that came when you ordered the DB2 Management Clients Package to install z/OS Enablement for your version (xx) of DB2.

Step 4: Install required maintenance for HDBxx10 and JDBxx1D

View the Preventive Service Planning information for the z/OS Enablement subset (JDB661D, JDB771D, JDB881D) you are about to install and make sure that you apply all available maintenance without exception. In DB2 Version 7, the activation sample job for z/OS Enablement DSNTIJCC was shipped with the DB2 Base Server FMID HDB7710. Hence, most PTFs for JDB771D also require you to install a corresponding PTF for HDB7710 first because it is defined as a prerequisite.

Step 5: Create required WLM application environments

With the exception of DSNWZP in DB2 Version 6 and 7, all DB2 provided stored procedures run in a WLM application environment. Table 25-6 lists considerations for creating WLM application environments for the DB2 provided stored procedures.

Table 25-6 WLM environment definitions for DB2 stored procedures

Stored procedure name	NUMTCB	WLM AE Name
DSNUTILS DSNUTILU	1	DSN1WLM1

Stored procedure name	NUMTCB	WLM AE Name
DSNTPSMP DSNTBIND	1	DSN1WLMREXX (requires special WLM procedure JCL DDDEFs)
DSNWZP*) WLM_REFRESH DSNACCSS DSNACCSI DSNAICUG DSNACCOR DSNACCMD DSNACCMO DSNACCDS DSNACCDR DSNACCDD DSNACCDE DSNACCDL INSTALL_JAR REPLACE_JAR REMOVE_JAR DSNTJSPP DSNACICS**) (all DSNTIJMS stored procedures)	>1	DSNWLM40
DSNACCJS DSNACCJF DSNACCJP DSNACCJQ DSNACCUC	> 1	DSN1WLMPC

We recommend creating at least five different WLM application environments per DB2 subsystem. It is required that DSNUTILS and DSNUTILU run in a WLM application environment where NUMTCB=1. DSNUTILS and DSNUTILU 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. The same requirement exists for LANGUAGE REXX stored procedures, which must also run in a NUMTCB=1 environment.

DSNACCMO behaves like the other 'regular' stored procedures and can be assigned to the WLM application environment that is called DSNWLM40. DSNACCMO is a complex stored procedure that creates up to 99 parallel threads to execute DB2 online utilities. For every parallel thread, a separate Task Control Block is required. Hence, a WLM application environment for DSNACCMO with NUMTCB=100 has to be created. No other stored procedure should be allowed to run in that application environment.

No special considerations exist for the remaining listed stored procedures except the stored procedures for submitting JCL and USS commands, which are required to be program-controlled. To minimize the overhead required to start new and manage WLM application environment address spaces, we recommend that you create a WLM application environment with NUMTCB=25 to 40, and assign all the listed stored procedures to it. However, in a very CPU-constrained environment, the recommendation is to use a lower number such as 6 through 10.

Notes:

- DSNWZP is run in DB2 SPAS in DB2 for OS/390 V7.
- DSNACICS NUMTCB must not exceed 25 if CICS 4.1 is to be called by DSNACICS. It must not exceed 100 for versions of CICS newer than CICS 4.1
  The recommended maximum value is 40.

Step 6: Define required RACF authorities and grant DB2 authorities

In addition to the required authorities to execute a stored procedure, the following RACF authorities and DB2 authorities need to be granted:

- Authorities needed for stored procedures for DB2 system administration

  The user ID that calls DSNWZP must have TRACE and MONITOR1 privileges.

  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.

  DSNACCSS, DSNACCSI, and DSNAICUG require no special authorization.

- Authorities needed for stored procedures for DB2 database administration

  The user ID that calls DSNACCOR must have SELECT authority on the real-time statistics tables and the DISPLAY system privilege.

  The user ID that runs a utility through DSNUTILS or DSNUTILU must have authorization to run the specified utility.

  The user ID that runs DSNACCMO must have the authorization to run the specified utility and the DISPLAY system privilege.

  DSNACCMD 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.

- Authorities needed for stored procedures for data set manipulation

  The data set manipulation stored procedures require no other authorities than the access authorities that are in place for the data set through RACF data set profiles.

- Authorities needed for stored procedures for submitting JCL and USS commands

  All these stored procedures use the __login() function to switch users. This requires that programs loaded into the WLM address space must be defined to RACF program control. Otherwise, the following error will be returned: `EDC5139I Operation not permitted`.

  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.

  To define programs from traditional libraries to program control, you need to:

  a. Activate the RACF program control (both access control to load modules and program access to data sets).

     ```
     SETROPTS WHEN(PROGRAM)
     ```

  b. Define one of the following profiles.

     - 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 require to be program controlled:

     ```
     SDSNLOAD(DSNX9WLM)
     ```

```
SDSNLOAD(DSNX9SPA)
SDSNLOAD(DSNARRS)
SDSNLOAD(DSN3IDOO)
SDSNLOAD(DSNX9WLS)
SDSNLOAD(DSNACCJS)
SDSNLOAD(DSNACCJP)
SDSNLOAD(DSNACCJQ)
SDSNLOAD(DSNACCJF)
SDSNLOAD(DSNACCUC)
```

- For all members in a data set:

```
RDEFINE PROGRAM * ADDMEM('datasetname'/volser/NOPADCHK) UACC(READ)
```

c. 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:

a. Define the resource profile:

```
RDEFINE FACILITY BPX.DAEMON.HFSCTL UACC(NONE)
```

b. Give READ access to users:

```
PERMIT BPX.DAEMON.HFSCTL CLASS(FACILITY) ID(uuuuuu) ACCESS(READ)
SETROPTS RACLIST(FACILITY) REFRESH
```

For more information on BPX.DAEMON and setting up program control, you can refer to *z/OS UNIX System Services Planning*, GA22-7800-03.

The stored procedures DSNACCJP and DSNACCJQ also use the Extended MCS console to issue JES commands to the console.

These two stored procedures use the TSO/E user ID (which is the user ID specified in the user ID parameter of the stored procedures) as the console name. So, one should consider ways to control what an authorized TSO/E user can do during a console session. The security administrator can define a RACF user profile to control the console attributes of the EMCS console.

For example:

```
ADDUSER USER001 OPERPARM(AUTH(SYS))
```

This example defines the user ID USER001 as an EMCS console with console attributes defined by the OPERPARM keyword. Note that the example includes only the information about console attributes for USER001. Additionally, with JES3, the privilege AUTH(CONS) is required to execute DSNACCJP. For complete information on the RACF ADDUSER command, refer to *z/OS Security Server RACF Command Language Reference*, SA22-7687-03.

Ensure that the user of the EMCS console (which is the user ID specified in the user-ID parameter of the stored procedures) has READ access to a profile in the RACF OPERCMDS class named:

```
MVS.MCSOPER.console-name
```

The following steps can be taken by the RACF security administrator to give users access to the RACF OPERCMDS class:

a. Issue the SETROPTS command to activate the OPERCMDS class:

```
SETROPTS CLASSACT(OPERCMDS)
```

b. Issue the SETROPTS command to activate generic profiles for the class:

```
SETROPTS GENERIC(OPERCMDS)
```

c. Issue RDEFINE to establish a profile for MVS.MCSOPER.*:

```
RDEFINE OPERCMDS MVS.MCSOPER.* UACC(NONE)
```

d. Give the TSO/E user ID access to the class:

```
PERMIT MVS.MCSOPER.* CLASS(OPERCMDS) ID(USER001) ACCESS(READ)
```

e. Issue the SETROPTS RACLIST command to refresh the OPERCMDS reserve class:

```
SETROPTS RACLIST(OPERCMDS) REFRESH
```

For more information on RACF commands, refer to *z/OS Security Server RACF Command Language Reference*, SA22-7687-03.

For more information on the EMCS console, refer to *z/OS MVS Planning: Operations,* SA22-7601-03.

The stored procedures DSNACCJP and DSNACCJQ issue the JES commands to cancel, purge, or display a job. To protect these JES commands, you need to:

a. Define the resource profile:

```
RDEFINE OPERCMDS jesname.CANCEL.* UACC(NONE)
RDEFINE OPERCMDS jesname.STOP.* UACC(NONE)
RDEFINE OPERCMDS jesname.DISPLAY.* UACC(NONE)
```

b. Give UPDATE access to users:

```
PERMIT jesname.CANCEL.* ID(uuuuuu) ACCESS(UPDATE)
PERMIT jesname.STOP.* ID(uuuuuu) ACCESS(UPDATE)
PERMIT jesname.DISPLAY.* ID(uuuuuu) ACCESS(READ)
SETROPTS RACLIST(OPERCMDS) REFRESH
```

To make sure that the related messages are always received by the EMCS console, provide the command:

```
ALTUSER userID OPERPARM(ROUTCODE(ALL) AUTH(INFO))
```

for the user under whose authority the stored procedures will be run.

The sample JCL in Example 25-1 shows all the RACF commands required for the DB2-supplied stored procedures that are not part of the DSNTIJMS, DSNTIJSG or DSNTIJCC activation JCL samples. It also shows how to restrict access to the extended MCS console to authorized users only.

Example 25-1 RACF commands

```
//RACFJOB  JOB (ACCOUNT),'NAME',MSGCLASS=H,MSGLEVEL=(1,1),CLASS=A,
// NOTIFY=&SYSUID
//STEP01   EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
  RDEFINE FACILITY BPX.DAEMON.HFSCTL UACC(READ)
  RALTER OPERCMDS MVS.MCSOPER.* UACC(READ)
  PE MVS.MCSOPER.* CLASS(OPERCMDS) ID(USRT003) ACCESS(NONE)
  SETROPTS RACLIST(OPERCMDS) REFRESH
  SETROPTS RACLIST(FACILITY) REFRESH
  ALU USRT001 OPERPARM(ROUTCODE(ALL))
  ALU USRT002 OPERPARM(ROUTCODE(ALL) AUTH(INFO))
```

In a production environment, it is recommended to define the required SDSNLOAD members to program control like this as an alternative to:

```
RDEFINE FACILITY BPX.DAEMON.HFSCTL UACC(READ)
```

as shown in Example 25-2.

Example 25-2 RACF commands in production environment

```
//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 DSNACCJS ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNACCJP ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNACCJQ ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   RDEFINE PROGRAM DSNACCJF ADDMEM('DSN.SDSNLOAD'/volser/NOPADCHK) UACC(READ)
   SETROPTS WHEN(PROGRAM) REFRESH
   RALTER OPERCMDS MVS.MCSOPER.* UACC(READ)
  PE MVS.MCSOPER.* CLASS(OPERCMDS) ID(USRT003) ACCESS(NONE)
  SETROPTS RACLIST(OPERCMDS) REFRESH
  SETROPTS RACLIST(FACILITY) REFRESH
  ALU USRT001 OPERPARM(ROUTCODE(ALL))
  ALU USRT002 OPERPARM(ROUTCODE(ALL) AUTH(INFO))
```

Step 7: Run activation jobs DSNTIJSG and DSNTIJCC or migration job DSNTIJCM

Edit and run the activation job DSNTIJSG first, and then DSNTIJCC following the instructions in their headers. If you are using 390 Enablement Version 7, and you are migrating to z/OS Enablement Version 8, edit and run the migration job DSNTIJCM instead of editing and running the activation job DSNTIJCC. Follow the instructions in the DSNTIJCM header before running the job.

Step 8: Verify with DB2 Control Center

Perform the following DB2 Control Center tasks to verify the installation and activation of the DB2-supplied stored procedures in DB2 Control Center:

1. Select a DB2 subsystem and when prompted, and enter a valid user ID and password to connect. The node expands and shows a number of folders under the subsystem icon including Databases, Table Spaces, Tables, and Indexes. If the node does not expand, DB2 Control Center has determined that the z/OS Enablement stored procedures are not known to DB2. In this case, z/OS Enablement is not installed and activated. This verification step tests DSNUTILS when connecting to a DB2 Version 7 or higher, because it executes the DIAGNOSE DISPLAY AVAILABLE utility through DSNUTILS to determine which online utilities are installed.
2. Select the **Buffer Pools** folder. A list of defined buffer pools is displayed. This step tests DSNACCMD.
3. Select the DB2 subsystem. Right-click **Display Subsystem Parameters**. You should get back a list of install panels and corresponding parameters and values. This step tests DSNWZP.
4. Select the **Table Spaces** folder. Choose any table space and right-click **Report...**. A valid report summary should be returned. This step tests DSNACCMO, DSNACCSS, and DSNUTILS.

5. Start a terminal emulator on your workstation and log on to a TSO/E session. Create and catalog a DUMMY sequential data set or member of a partitioned data set. You will use this data set to test the data set functions of the z/OS Enablement stored procedures using DB2 Control Center. You can copy and rename an existing data set to create the DUMMY data set.
6. From DB2 Control Center, select the **Data Sets** folder. When prompted, enter a filter that will cause the DUMMY data set or the PDS that contains the DUMMY member to be displayed. This step tests DSNACCDL.
7. Select the **DUMMY** data set or member in the details view, and right-click the **Rename** action. Rename the DUMMY data set to DUMMY2. Refresh the details view and verify that DUMMY2 now appears in the list. This step tests DSNACCDR.
8. Expand the **Utility Objects** folder, and then select the **Data Set Templates** folder. Select a template in the details view, and right-click the **Show Statements** action. From the Show Utility Statements dialog select the **Export** button.
9. In the Export to Data Set dialog, enter Name and Member values to point to the DUMMY2 data set. Select the **OK** button. You should be presented with a dialog that asks whether you wish to append to or replace the data set. Select **APPEND** or **REPLACE** to continue. This step tests DSNACCDS.
10. Select the **Data Set** folder once again, and enter filter criteria to display the DUMMY2 data set. Right-click to select the delete action for the DUMMY2 data set. Refresh the details view to confirm that the data set has been deleted. This step tests DSNACCDD.
11. Start a DB2 Command Window and enter the following commands:

```
db2 connect to  user  using
db2 SELECT USERS FROM TABLE (ICM.USER_GROUPS(1,'')) AS T(USERS)
```

You should retrieve a number of user IDs like this:

```
USERS
--------
BPXROOT
WEBADM
BPXROOT
WEBADM
WEBSRV
```

You will only see users that have an OMVS segment defined. This step tests DSNAICUG. To verify the JCL stored procedures, run the sample program DB2JCLUtilities. You can find the source code for it in Apendix B and download it as additional material.

25.3 z/OS Enablement stored procedures

The z/OS Enablement stored procedures are not documented in the DB2 for z/OS manuals. The following section contains the syntax diagram, and a description of the options and results of each stored procedure. This information is a Product-sensitive Programming Interface.

25.3.1 DSNACCJF

This DB2-supplied stored procedure can be used to fetch the output of a submitted JCL job.

Figure 25-2 shows the DSNACCJF syntax diagram.

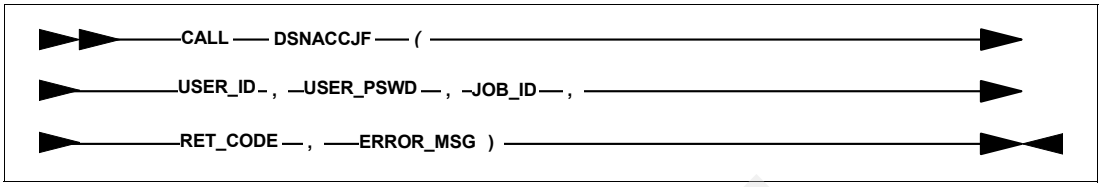


Figure 25-2 DSNACCJF syntax diagram

The possible options are:

- **USER_ID**: User ID that the stored procedure runs under. This is an input parameter of type VARCHAR(8).
- **USER_PSWD**: Password for the user ID specified in the USER_ID parameter. This is an input parameter of type VARCHAR(8).
- **JOB_ID**: JES2 or JES3 job ID whose SYSOUT data sets are to be processed. This is an input parameter of type VARCHAR(8).
- **RET_CODE**: Return code from the stored procedure. Possible values are:
  - 0 - The call completed successfully.
  - 12 - The call did not complete successfully.

  The ERROR_MSG parameter contains messages that describe the error.

  This is an output parameter of type INTEGER.
- **ERROR_MSG**: 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 DB2 Universal Database™ might follow the first messages.

  This is an output parameter of type VARCHAR(1331).

DSNACCJF output

If DSNACCJF executes successfully, in addition to the output parameters, DSNACCJF returns a result set that contains the data from the SYSOUT data sets of the job ID specified in the input parameter JOB_ID.

Table 25-7 shows the format of the result set.

Table 25-7 Result set format

Column name	Data type	Contents
JF_SEQUENCE	INTEGER	Sequence number of the table row (1,...n)
JF_TEXT	VARCHAR(4096)	Record in the SYSOUT data set

25.3.2 DSNACCJP

This DB2-supplied stored procedure can be used to purge a JCL job.

Figure 25-3 shows the DSNACCJP syntax diagram.

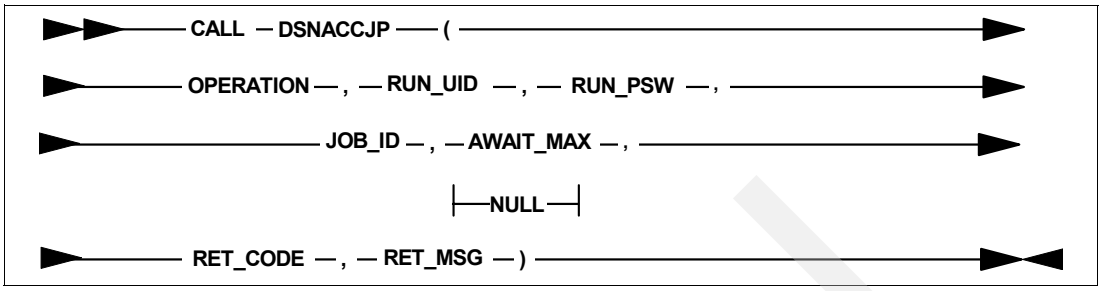


Figure 25-3 DSNACCJP syntax diagram

The possible options are:

- **OPERATION:** Specifies the type of command to invoke:
  - 1 - Cancel a job
  - 2 - Purge a job

  This is an input parameter of type INTEGER.
- **RUN_UID**: User ID that the stored procedure DSNACCJP runs under. This is an input parameter of type CHAR(8).
- **RUN_PSW**: Password for the user ID specified in the RUN_UID parameter. This is an input parameter of type CHAR(8).
- **JES_ID**: This is the ID that has been assigned by the system programmer to the JES. If only one JES2 is in use then usually the default name JES2 is used.

  This is an input parameter of type VARCHAR(8).
- **JOB_ID**: JES2 or JES3 job ID. It can have the format J####### or JOB#####, where the # symbol stands for any digit 0-9. This is an input parameter of type CHAR(8).
- **AWAIT_MAX**: Number of seconds that DSNACCJP will wait for the requested operation to be processed by JES.

  If this parameter is set to NULL, then the default value (1 second) will be used. If the time expires, then the stored procedure will finish with return code 4.

  This is an input parameter of type INTEGER.
- **RET_CODE**: Return code from the stored procedure. Possible values are:
  - 0 - The call completed successfully. The RET_MSG parameter contains the output from the cancel or purge operation.
  - 4 - The call completed successfully but the operation is not processed within the time specified in the wait-time parameter. The RET_MSG parameter contains messages that describe the error.
  - 8 - The call did not complete successfully. An error occurred. The RET_MSG parameter contains messages that describe the error.
  - 12 - The call did not complete successfully. A severe error occurred. The RET_MSG parameter contains messages that describe the error.

  This is an output parameter of type INTEGER.
- **RET_MSG**: 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 z/OS might follow the first messages. This is an output parameter of type VARCHAR(32677).

25.3.3 DSNACCJQ

This DB2-supplied stored procedure can be used to retrieve a JCL job status.

Figure 25-4 shows the DSNACCJQ syntax diagram.

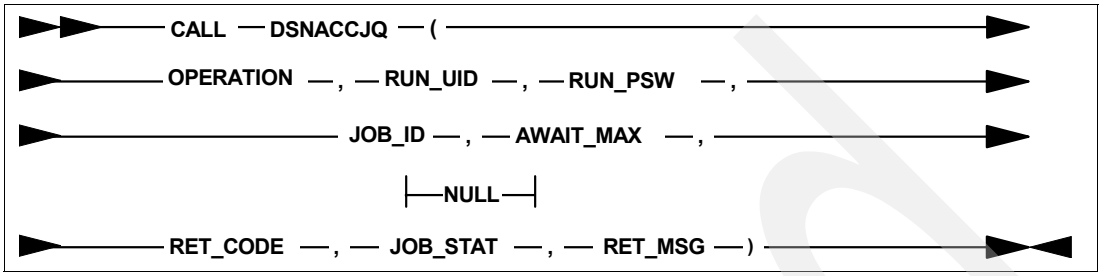


Figure 25-4 DSNACCJQ syntax diagram

The possible options are:

- **OPERATION**: Format of the job status information. Possible values are:
  - 1: Returns a job status indicating whether the job is in the input queue, is active, or is in the output queue.
  - 2: Returns the job status in a more detailed format.

  This is an input parameter of type INTEGER.
- **RUN_UID**: User ID that the stored procedure DSNACCJQ runs under.

  This is an input parameter of type CHAR(8).
- **RUN_PSWD**: Password for the user ID specified in the RUN_UID parameter.

  This is an input parameter of type CHAR(8).
- **JOB_ID**: Specifies a JES2 or JES3 job ID. It can have the format J####### or JOB#####, where the # symbol stands for any digit 0 to 9.

  This is an input parameter of type CHAR(8).
- **AWAIT_MAX:** Specifies the number of seconds that DSNACCJQ will wait for the requested operation to be processed by JES. This value is ignored when OPERATION 1 is specified.

  If this parameter is set as NULL, then the default value (1 second) will be used.

  If the time expires, then the stored procedure will finish with the return code 4.

  This is an input parameter of type INTEGER.
- **RET_CODE:** Return code from the stored procedure. Possible values are:
  - 0 - The call completed successfully.

    If OPERATION 2 is specified, the RET_MSG parameter contains the job status information in a more detailed format.
  - 4 - If OPERATION 2 is specified, the call completed successfully, but the operation is not processed within the time specified in the wait-time parameter.

    The RET_MSG parameter contains messages that describe the error.
  - 8 - The call did not complete successfully. An error occurred.

    The RET_MSG parameter contains messages that describe the error.
  - 12 - The call did not complete successfully. A severe error occurred.

The RET_MSG parameter contains messages that describe the error.

This is an output parameter of type INTEGER.

- **JOB_STAT**:

  If OPERATION 1 is specified, one of the following job status information conditions is returned:

  - 1 The job was received, but not yet run (INPUT)
  - 2 The job is running (ACTIVE)
  - 3 The job has finished and has output to be printed or retrieved (OUTPUT)
  - 4 The job is not found (return code is set to 4)
  - 5 The job is in an unknown phase (return code is set to 4)

  If OPERATION 2 is specified, zero is returned.

  This is an output parameter of type INTEGER.

- **RET_MSG**: 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 z/OS might follow the first messages.

  This is an output parameter of type VARCHAR(32677).

25.3.4 DSNACCJS

This DB2-supplied stored procedure can be used to submit a JCL job.

Figure 25-5 shows the DSNACCJS syntax diagram.

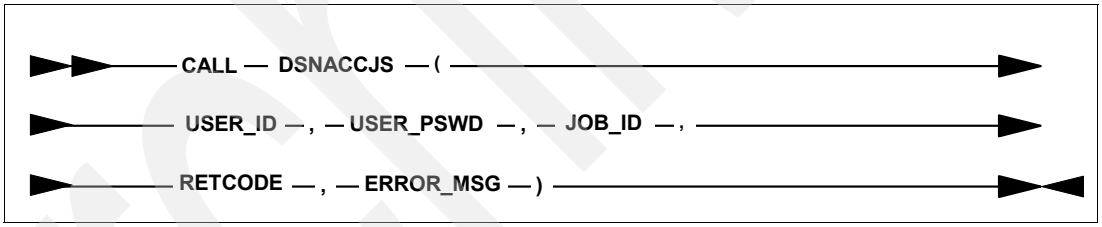


Figure 25-5 DSNACCJS syntax diagram

The possible options are:

- **USER_ID**: User ID that the stored procedure DSNACCJS runs under.

  This is an input parameter of type VARCHAR(8).

- **USER_PSWD**: Password for the user ID specified in the USER_ID parameter.

  This is an input parameter of type VARCHAR(8).

- **JOB_ID**: JES2 or JES3 job ID of the submitted job.

  This is an output parameter of type VARCHAR(8).

- **RETCODE**: Return code from the stored procedure. Possible values are:

  - 0 - The call completed successfully.
  - 12 - The call did not complete successfully.

  The msg-area parameter contains messages that describe the error.

  This is an output parameter of type INTEGER.

- **ERROR_MSG**: 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 DB2 Universal Database might follow the first messages. This is an output parameter of type VARCHAR(1331)

DSNACCJS input

We have seen the input parameters described in the list of DSNACCJS options, in addition DSNACCJS submits the job from the global temporary table DSNACC.JSRECORDS for execution.

The following table shows the format of the temporary table:

```
Column name   Data type     Contents
-----------   ---------     ---------------------------------
JS_SEQUENCE   INTEGER       Sequence number of the table row (1,...n)
JS_TEXT       VARCHAR(80)   A JCL statement
```

25.3.5 DSNACCUC

This DB2-supplied stored procedure can be used to issue USS commands.

Figure 25-6 shows the DSNACCUC syntax diagram.

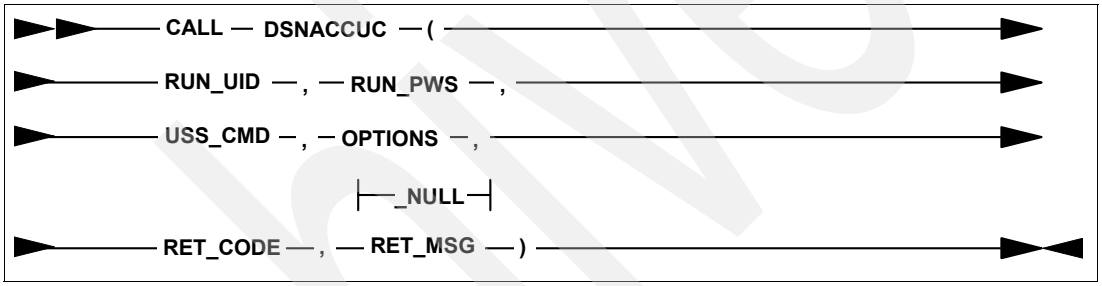


Figure 25-6 DSNACCUC syntax diagram

The possible options are:

- **RUN_UID**: User ID that the stored procedure DSNACCUC runs under.

  This is an input parameter of type CHAR(8).

- **RUN_PSW**: Password for the user ID specified in the RUN_UID parameter.

  This is an input parameter of type CHAR(8).

- **USS_CMD**: USS command to be executed.

  This is an input parameter of type VARCHAR(32677).

- **OPTIONS**: Specifies how the output from the USS command is returned. Possible values are:

  – OUTMODE=LINE - The output from the USS command is a multi-line message.

    Each line is returned as a row in the result set.

  – OUTMODE=BLK - The output from the USS command is a multi-line message.

    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 default option OUTMODE=BLK is used.

  This is an input parameter of type VARCHAR(1024).

- **RET_CODE**: Return code from the stored procedure. Possible values are:

– 0 - The call completed successfully.
– 12 - The call did not complete successfully. A severe error occurred. The RET_MSG parameter contains messages that describe the error.

This is an output parameter of type INTEGER.

- **RET_MSG**: 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 z/OS or DB2 Universal Database might follow the first messages.

  This is an output parameter of type VARCHAR(32677).

DSNACCUC output

If DSNACCUC executes successfully, in addition to the output parameters described above, DSNACCUC returns a result set that contains the output from the USS command.

Table 25-8 shows the format of the result set.

Table 25-8 Contents of DSNACCUC result set

Column name	Data type	Contents
UC_SEQUENCE	INTEGER	Sequence number of the table row (1,...n)
UC_TEXT	VARCHAR(32677)	A block of text or a line from the output of the USS command

25.3.6 DSNACCMO

This DB2-supplied stored procedure can be used for invoking parallel utility execution.

Figure 25-7 shows the DSNACCMO syntax diagram.

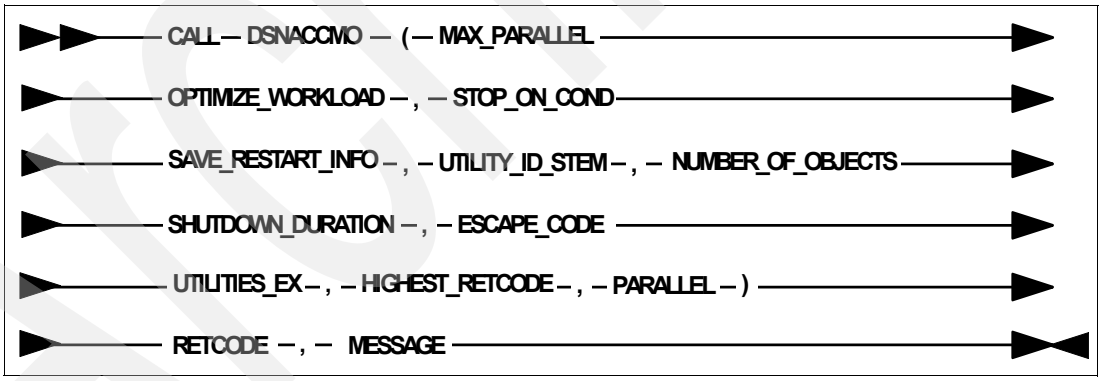


Figure 25-7 DSNACCMO syntax diagram

The possible options are:

- **MAX_PARALLEL**: 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. You may specific 1 to 99. This is an input parameter of type SMALLINT.
- **OPTIMIZE_WORKLOAD:** Specifies whether the parallel utility executions should be sorted to achieve shortest overall execution time. This is an input parameter of type VARCHAR(8). NO or null indicates that the workload is not to be sorted. The default is null. YES indicates that the workload is to be sorted.
- **STOP_ON_COND**: Utility execution condition after which DSNACCMO 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. This is an input parameter of type VARCHAR(8):

  - AUTHORIZ or null: No new utility executions will be started after one of the currently running utilities has encountered a return code from DSNUTILS of 12 or higher. The default is null.
  - WARNING: No new utility will be started after one of the currently running utilities has encountered a return code from DSNUTILS of 4 or higher.
  - ERROR: No new utility will be started after one of the currently running utilities has encountered a return code from DSNUTILS of 8 or higher.

- **SAVE_RESTART_INFO**: Specifies if restart info for stopped utilities should be inserted into the DB2 Control Center utility restart tables. This is an input parameter of type VARCHAR(8).
  - NO or null indicates that the restart info is not saved in the DB2 Control Center utility restart tables. The default is null.
  - YES indicates that the restart info is saved in the DB2 Control Center utility restart tables. The use of YES is reserved for DB2 Control Center use. In your client applications, always use NO.
- **UTILITY_ID_STEM**: Utility-id stem. This is an input parameter of type VARCHAR(8).

  The actual utility-id of a utility-execution in a parallel thread will be dynamically created as utilityidstemTTNNNNNN where TT is the zero-padded number of the subtask executing the utility, and NNNNNN is a consecutive number of utilities executed in a subtask.

  utilityidstem02000005 is the fifth utility execution that has been processed by the second subtask.
- **NUMBER_OF_OBJECTS**: Number of utility executions and their sorting objects that were passed in the DSNACC.MO_TBL.

  This is an input parameter of type INTEGER from 1 to 999999.
- **SHUTDOWN_DURATION**: Number of seconds from 1 to 999999999999999 DSNACCMO 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:
  - This is an input parameter of type FLOAT(8).
  - 999 or null indicates that a shutdown will not be performed.
  - The default is null.
- **ESCAPE_CODE**: Escape character code of your DB2 server.

  This is an input parameter of type CHAR(1):
  - D indicates double-quote " which is the default if the SQLDELI parameter is DEFAULT. (See the *Installation Guide* for details.)
  - S indicates single-quote '
  - Always specify D since the utilities are not sensitive to values set in the SQLDELI parameter (see the *Installation Guide* for details) so delimited identifiers are always in double quote ".
- **UTILITIES_EX**: Number of actual utility executions. This is an output parameter of type INTEGER.
- **HIGHEST_RETCODE**: Highest returncode from DSNUTILS for all utility executions. This is an output parameter of the type INTEGER.

- **PARALLEL:** Actual number of parallel tasks that were started to execute the utility in parallel. This is an output parameter of the type SMALLINT.
- **RETCODE**: Retcode of DSNACCMO.

  This is an output parameter of the type INTEGER:

  – 0 - All parallel utility executions ran successfully.
  – 4 - The statistics for one or more sorting objects have not been gathered in the catalog.
  – 12 - A DSNACCMO error occurred. The message will contain details.

  Note: The DSNACCMO return code is different from the DSNUTILS highest_retcode. If for instance stoponcond AUTHORIZ is selected, and all passed utility executions end in a DSNUTILS return code of 8, DSNACCMO will still return retcode 0, but the dsnutils highest return code of 8. Only if a DSNACCMO internal error occurred, such subtasks cannot be started due to lack of memory in the address space; DSNACCMO will return with a return code of higher than 4.

- **MESSAGE**-text is an output parameter of type VARCHAR(1331).

Passing parameters to DSNACCMO

The calling application has to insert a row for every utility execution into DSNACC.MO_TBL. If a utility is executed against multiple objects in a single utility execution, only a single insert has to be made with the object name of the most significant object for the utility execution (the sorting object). The sorting object that is specified there is used for sorting the utility executions, and for replacing the placeholder in the associated utility statement.

DSNACC.MO_TBL is created as shown in Example 25-3.

Example 25-3 DDL for DSNACC.MO_TBL

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.MO_TBL
  ( MO_OBJECTID INTEGER NOT NULL,
    MO_STMTID INTEGER NOT NULL,
    MO_TYPE VARCHAR(261) NOT NULL,
    MO_NAME VARCHAR(31) NOT NULL,
    MO_PART SMALLINT,
    MO_RESTART VARCHAR(8) NOT NULL,
    MO_UTILITY_NAME VARCHAR(20)  NOT NULL,
    MO_USE_TEMPLATE CHAR(1) NOT NULL,
    MO_RECDSN VARCHAR(54)  NOT NULL,
    MO_RECDEVT CHAR(8)  NOT NULL,
    MO_RECSPACE SMALLINT NOT NULL,
    MO_DISCDSN VARCHAR(54)  NOT NULL,
    MO_DISCDEVT CHAR(8)  NOT NULL,
    MO_DISCSPACE SMALLINT NOT NULL,
    MO_PNCHDSN VARCHAR(54)  NOT NULL,
    MO_PNCHDEVT CHAR(8)  NOT NULL,
    MO_PNCHSPACE SMALLINT NOT NULL,
    MO_COPYDSN1 VARCHAR(54)  NOT NULL,
    MO_COPYDEVT1 CHAR(8)  NOT NULL,
    MO_COPYSPACE1 SMALLINT NOT NULL,
    MO_COPYDSN2 VARCHAR(54)  NOT NULL,
    MO_COPYDEVT2 CHAR(8)  NOT NULL,
    MO_COPYSPACE2 SMALLINT NOT NULL,
    MO_RCPYDSN1 VARCHAR(54)  NOT NULL,
    MO_RCPYDEVT1 CHAR(8)  NOT NULL,
```

```
MO_RCPYSPACE1 SMALLINT NOT NULL,
MO_RCPYDSN2 VARCHAR(54)  NOT NULL,
MO_RCPYDEVT2 CHAR(8)  NOT NULL,
MO_RCPYSPACE2 SMALLINT NOT NULL,
MO_WORKDSN1 VARCHAR(54)  NOT NULL,
MO_WORKDEVT1 CHAR(8)  NOT NULL,
MO_WORKSPACE1 SMALLINT NOT NULL,
MO_WORKDSN2 VARCHAR(54)  NOT NULL,
MO_WORKDEVT2 CHAR(8)  NOT NULL,
MO_WORKSPACE2 SMALLINT NOT NULL,
MO_MAPDSN VARCHAR(54)  NOT NULL,
MO_MAPDEVT CHAR(8)  NOT NULL,
MO_MAPSPACE SMALLINT NOT NULL,
MO_ERRDSN VARCHAR(54)  NOT NULL,
MO_ERRDEVT CHAR(8)  NOT NULL,
MO_ERRSPACE SMALLINT NOT NULL,
MO_FILTRDSN VARCHAR(54)  NOT NULL,
MO_FILTRDEVT CHAR(8)  NOT NULL,
MO_FILTRSPACE SMALLINT NOT NULL)
CCSID EBCDIC;
```

The columns have the following meaning:

- MO_OBJECTID specifies a unique positive identifier for the object the utility execution is associated with. When you insert multiple rows, increment MO_OBJECTID by 1, starting at 0 for every insert.
- MO_STMTID points to a statement row in DSNACC.MO_TBL2.
- MO_TYPE is the input type of the sorting object and can be:
  - TABLESPACE
  - INDEXSPACE
  - TABLE
  - INDEX
  - STOGROUP
- MO_NAME contains the name of the sorting object. If the sorting object is not qualified and of the type TABLESPACE and INDEXSPACE, then the default database is DSNDB04. If the sorting object is of type table or index, the schema is the current SQL authorization ID. MO_NAME has a different length for DB2 for z/OS Version 8 as it allows longer database object names.
- MO_PART may be NULL or 0 if the sorting object does not specify a partition or the partition number otherwise.
- MO_RESTART is the restart parameter of DSNUTILS; see APPENDIX1.2.1.5 DSNUTILS option descriptions.
- MO_UTILITY_NAME is the utility_name parameter of DSNUTILS; see APPENDIX1.2.1.5 DSNUTILS option descriptions.

  All DSNUTILS utility_name parameters may be used except ANY. Template dynamic allocation is indicated in the next parameter. The utility name must be known to the optimizing scheduler also in the case of template dynamic allocation.
- MO_USE_TEMPLATE may be Y or N; use Y when template dynamic allocation is used in the utility statement.
- MO_RECDSN to MO_FILTRSPACE: See the corresponding parameters of DSNUTILS as described in “Appendix C. DB2-supplied stored procedures” of the *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

- MO_STMTID points to a utility statement in DSNACC.MO_TBL2.

DSNACC.MO_TBL2 is created as shown in Example 25-4.

Example 25-4 DDL for DSNACC.MO_TBL2

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.MO_TBL2
  ( MO_STMTID INTEGER NOT NULL,
    MO_SEQUENCE INTEGER NOT NULL,
    MO_UTSTMT VARCHAR(4000) NOT NULL)
    CCSID EBCDIC;
```

The columns have the following meaning:

- MO_STMTID is a unique positive identifier for a single utility execution statement.
- MO_SEQUENCE is a unique positive identifier for a part of the utility execution statement. If a utility statement exceeds 4000 characters, it can be split up and inserted into DSNACC.MO_TBL2 with the sequence starting at 0, and then being incremented with every insert. For the actual execution, the statement pieces are concatenated without any separation characters or blanks in between.
- MO_UTSTMT is a utility statement or part of a utility statement. A placeholder &OBJECT. can be used to be replaced by the sorting object name passed in DSNACC.MO_TBL. 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 for each thread of the utility execution.

DSNACCMO 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, DSNACCMO 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 DSNACCMO) 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 run time 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 information. 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 both 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. Although 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

DSNACCMO allows you to specify the maximum number of parallel subtasks that may be started between 1 to 99.

Every started subtask will require 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 DSNACCMO with higher parallelism.

25.3.7 DSNACCDS

DSNACCDS is a DB2-supplied stored procedure, which writes records passed as input parameters in a global temporary table to either a PS data set, PDS, PDSE member, or GDS. It can either append or replace an existing PS or GDS data set or member. It can create a new PS data set, PDS or PDSE data set or member, or a new GDS from an existing GDG as needed, and it will allocate a minimal amount of space.

The syntax diagram in Figure 25-8 shows the SQL CALL statement for DSNACCDS.

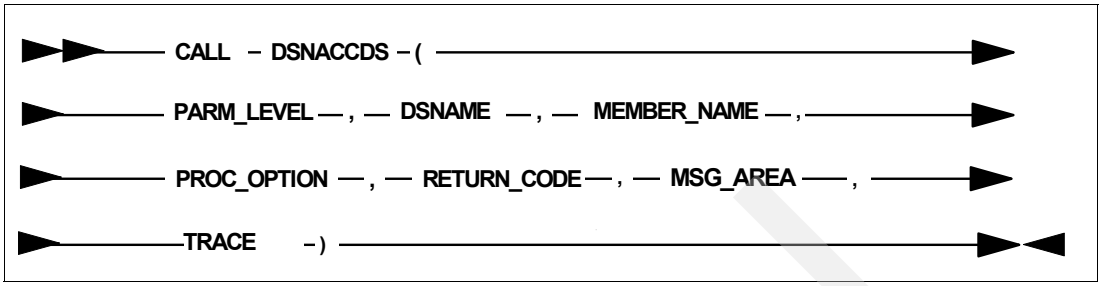


Figure 25-8 DSNACCDS syntax diagram

The possible options are:

- **PARM_LEVEL**: level of parameter list (always 1).

  This is an input parameter of type INTEGER.

- **DSNAME**: Data set name.

  This is an input parameter of type CHAR(44).

- **MEMBER_NAME**: Member name (blank for PS). For a GDS, specify the generation number e.g. 0, +1, -1.

  This is an input parameter of type CHAR(8).

- **PROC_OPTION**:

  This is an input parameter of type CHAR(2). Possible values are:

  – R: Replace
  – A: Append
  – NM: New member
  – ND: New data set (either PS, PDS, or GDS)

- **RETURN_CODE:**

  This is an output parameter of type INTEGER. Possible values are:

  – 0 Data set stored successfully
  – <> 0 Error occurred while storing data set (check MSG_AREA).

- **MSG_AREA**:

  Error message if RETURN_CODE <> 0. This is an output parameter of the type VARCHAR(1000).

- **TRACE**: Possible values are: Y or N

  This is an input parameter of the type CHAR(1).

DSNACCDS input records table

The input table is defined as:

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.DSNRECORDS
   (UTIL_SEQNBR INTEGER NOT NULL,
   UTIL_RECORD CHAR(72) NOT NULL);
```

Use a positive, incrementing sequence number (line number) when you insert rows into the input records table. The sequence numbers are just used for the input cursor. They will not be printed into the data set.

25.3.8 DSNACCDL

DSNACCDL is a DB2-supplied stored procedure, which can be used to list data sets, members, or generations for a GDG.

The syntax diagram in Figure 25-9 shows the SQL CALL statement for DSNACCDL.

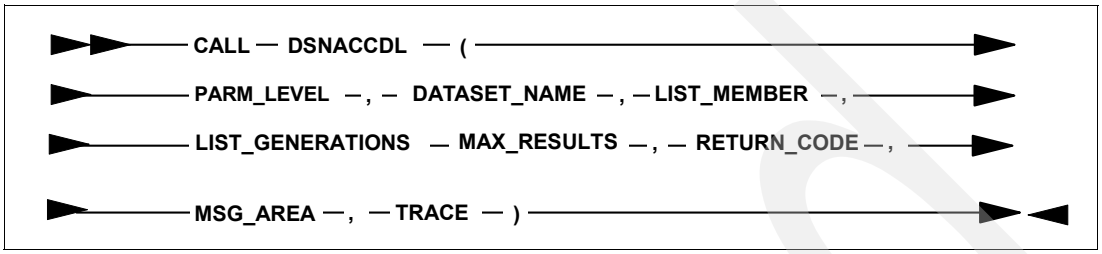


Figure 25-9 DSNACCDL syntax diagram

The possible options are:

- **PARM_LEVEL**: Level of parameter list (always 1).

  This is an input parameter of type INTEGER.

- **DATASET_NAME**: Data set name with masking characters e.g. PAOLOR1.** or without to list only 1 data set.

  This is an input parameter of type CHAR(44).

- **LIST_MEMBERS:** Possible values are: Y or N (only set to Y if DATASET_NAME is fully qualified PDS).

  This is an input parameter of type CHAR(1).

- **LIST_GENERATIONS:** Possible values are: Y or N (only set to Y if DATASET_NAME is fully qualified GDG).

  This is an input parameter of type CHAR(1).

- **MAX_RESULTS:** Max number of result set rows

  This is an input parameter of type INTEGER.

- **RETURN_CODE**:

  This is an output parameter of the type INTEGER. Possible values are:

  - 0 Data sets listed successfully
  - <> 0 Error occurred while listing data set (check MSG_AREA)

- **MSG_AREA**: Error message if RETURN_CODE <> 0

  This is an output parameter of the type VARCHAR(1000).

- **IN_TRACE**: Possible values are: Y or N

  This is an input parameter of type CHAR(1).

The DSNACCDL output table is listed in Example 25-5.

Example 25-5 DSNACCDL output table

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.DSLIST
   (DSNAME VARCHAR(44) NOT NULL
   ,CREATE_YEAR INTEGER NOT NULL
   ,CREATE_DAY  INTEGER NOT NULL
   ,DS_TYPE     INTEGER NOT NULL
   ,VOLUME      CHAR(6) NOT NULL
```

```
,PRIMARY_EXTENT INTEGER  NOT NULL
,SECONDARY_EXTENT INTEGER NOT NULL
,MEASUREMENT_UNIT CHAR(9) NOT NULL
,EXTENTS_IN_USE INTEGER NOT NULL,
,DASD_USAGE       INTEGER NOT NULL
,DS_HARBA         INTEGER NOT NULL
,DS_HURBA         INTEGER NOT NULL )
CCSID EBCDIC;
```

The columns have the following meaning:

- CREATE_YEAR (for example, 2002) and CREATE_DAY (for example, 240) are not available for data set type 5 (GDG) or migrated data sets (which return the year as 0).
- DS_TYPE:
  - 0: Unknown type of data set
  - 1: PDS data set
  - 2: PDSE data set
  - 3: Member of PDS or PDSE
  - 4: Physical seq data set
  - 5: Generation data group
  - 6: Generation data set
  - Everything else is unknown

The only way to find out if too few data sets were requested is to check if the returned number of data sets < MAX_RESULTS.

25.3.9 DSNACCDR

DSNACCDR is a DB2-supplied stored procedure that can be used to rename a data set or a member.

The syntax diagram in Figure 25-10 shows the SQLCALL statement for DSNACCDR.

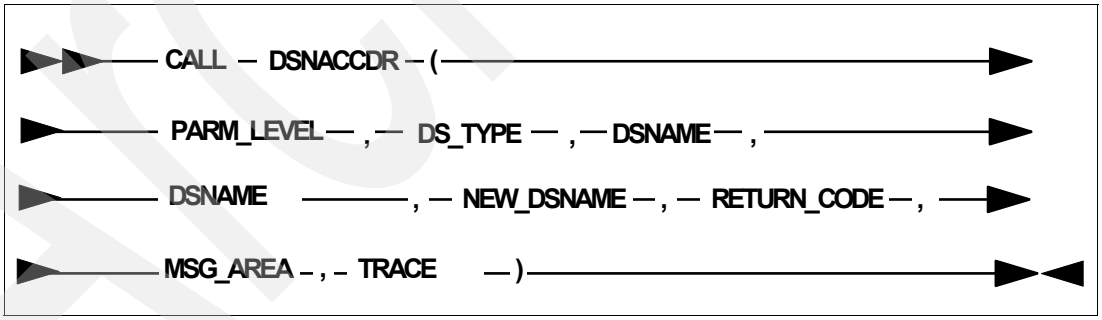


Figure 25-10 DSNACCDR syntax diagram

The possible options are:

- **PARM_LEVEL** level of parameter list (always 1).

  This is an input parameter of type INTEGER.
- **DS_TYPE:** Possible values are:
  - 1: PDS data set
  - 2: PDSE data set
  - 3: Member of PDS or PDSE
  - 4: Physical seq data set

This is an input parameter of the type INTEGER.

- **DSNAME** for members it is the member name, otherwise, the fully qualified file name.

  This is an input parameter of type CHAR(44).

- **PARENT_DSNAME** blank for ds types 1,2,4, for the member it is the PDS name.

  This is an input parameter of type CHAR(44).

- **NEW_DSNAME** for member it is the member name, otherwise, the fully qualified file name.

  This is an input parameter of type CHAR(44).

- **RETURN_CODE**

  This is an output parameter of the type INTEGER. Possible values are:

  - 0 Data set renamed successfully
  - <> 0 Error occurred while renaming data set (check MSG_AREA)

- **MSG_AREA:** Error message if RETURN_CODE <> 0

  This is an output parameter of the type VARCHAR(1000).

- **IN_TRACE:** Y or N

  This is an input parameter of the type CHAR(1).

25.3.10 DSNACCDD

DSNACCDD is a DB2-supplied stored procedure that can be used to delete a data set, member of a PDS or PDSE, or a GDS.

The syntax diagram in Figure 25-11shows the SQL CALL statement for DSNACCDD.

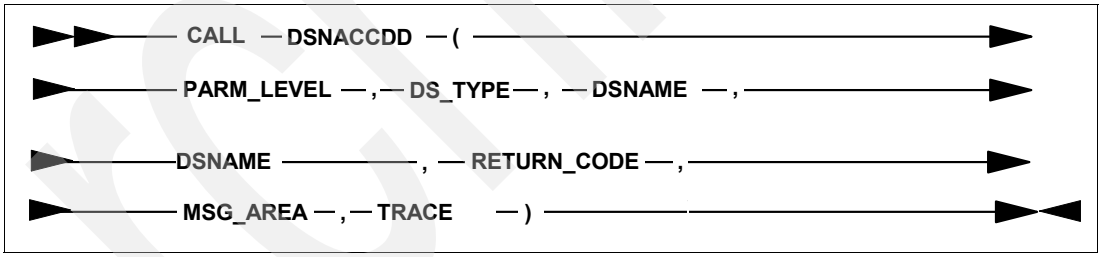


Figure 25-11 DSNACCDD syntax diagram

The options are:

- **PARM_LEVEL**: level of parameter list (always 1)

  This is an input parameter of type INTEGER.

- **DS_TYPE**: 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 seq data set
  - 6, Generation data set

  This is an input parameter of the type INTEGER.

- **DSNAME:** For a member it is the member name, for a GDS it is the absolute generation number such as G0001V00, otherwise, it is the fully qualified data set name.

This is an input parameter of the type CHAR(44).

- **PARENT_DSNAME**

  This is an input parameter for the type CHAR(44).

  - Blank for ds types 1,2,4,
  - For member it is the PDS name
  - For GDS it is the GDG name.

- **RETURN_CODE**

  This is an output parameter of the type INTEGER. Possible values are:

  - 0 Data set deleted successfully
  - > 0 Error occurred while deleting data set (check MSG_AREA)

- **MSG_AREA**: Error message if RETURN_CODE <> 0. This is an output parameter of type VARCHAR(1000).
- **IN_TRACE**: Y or N. This is an output parameter of type CHAR(1).

25.3.11 DSNACCDE

DSNACCDE is a DB2-supplied stored procedure that can be used to check if a data set is cataloged, or a member of a cataloged PDS exists, or if a GDG or a GDS in a GDG exists.

The syntax diagram in Figure 25-12 shows the SQL CALL statement for DSNACCDDE.

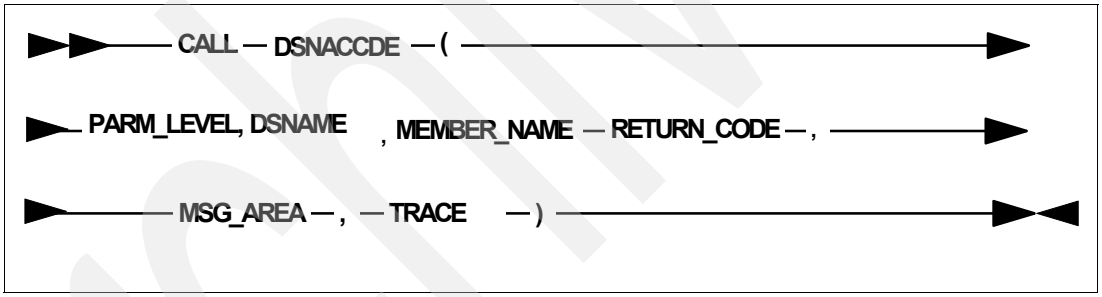


Figure 25-12 DSNACCDE syntax diagram

.The possible options are:

- **PARM_LEVEL**: Level of parameter list (always 1)

  This is an input parameter of the type INTEGER.

- **DSNAME:** Fully qualified file name

  This is an input parameter of the type CHAR(44)

- **MEMBER_NAME**: Member name for PDS and PDSE

  This is an input parameter for the type CHAR(8).

- **RETURN_CODE**

  This is an output parameter of the type INTEGER:

  - 0: Data set was found
  - 1: Data set was not found
  - 2: Member was not found
  - >2: Error occurred (check MSG_AREA)

- **MSG_AREA**: Error message

  This is an output parameter of the type VARCHAR(1000).

- **TRACE**: Y or N

  This is an input parameter of the type CHAR(1).

The following shows a GDG and a GDS:

```
USER01.GDG
USER01.GDG.G0001V00
```

DSNACCDL will only list the GDG. The GDSs will only be listed by DSNACCDL when the DSNAME is USER01.GDG and LIST_GENERATIONS=Y.
The DSNAME returned then is G0001V00. GDGs cannot be renamed.
If you want to check with DSNACCDE if a generation exists, check with a DSNAME of USER01.GDG.G0001V00.
To delete a GDS, enter G0001V00 as the DSNAME and USER01.GDG as the PARENT_DSNAME.

25.3.12 DSNACCSI

DSNACCSI is a DB2-supplied stored procedure that can be used to return the fully qualified domain name of the DB2 server.

The syntax diagram in Figure 25-13 shows the SQL CALL statement for DSNACCSI.

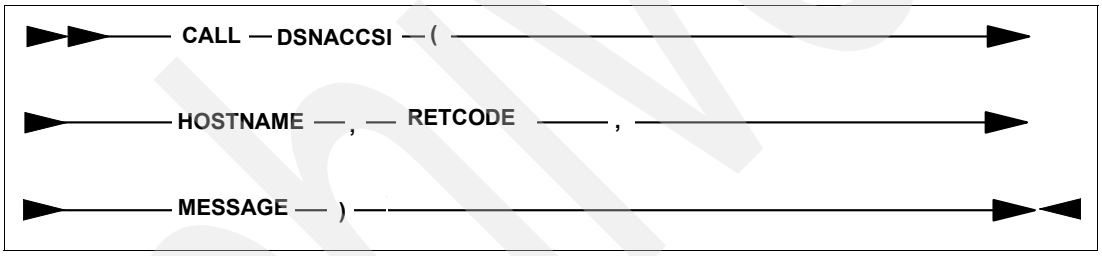


Figure 25-13 DSNACCSI syntax diagram

The possible options are:

- **HOSTNAME**: Fully-qualified domain name such as `wtsc63.itso.ibm.com`.

  This is an output parameter of the type VARCHAR(1024).

- **RETCODE**

  This is an output parameter of the type INTEGER:

  - 0: Success
  - 12: An error occurred

- **MESSAGE**: Error message

  This is an output parameter of type VARCHAR(120).

25.3.13 DSNACCSS

DSNACCSS returns the SSID of the connected DB2 subsystem.

The syntax diagram in Figure 25-14 shows the SQL CALL statement for DSNACCSS.

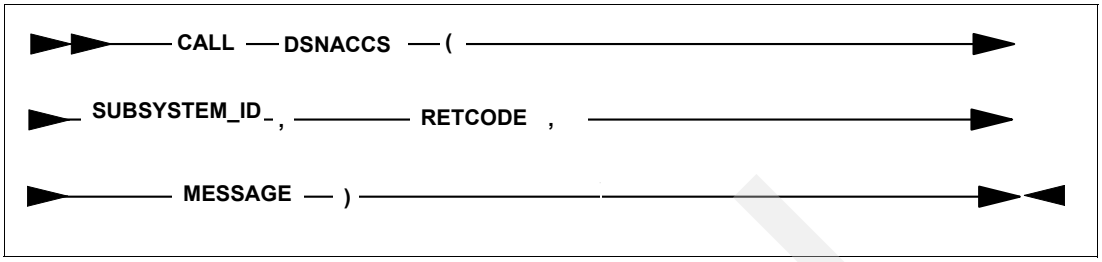


Figure 25-14 DSNACCSS syntax diagram

The possible options are:

- **SUBSYSTEM_ID:** SSID of the connected subsystem such as DSN1.

  This is an output parameter of the type CHAR(4).

- **RETCODE**

  This is an output parameter of the type INTEGER:

  - 0: Success
  - 12: An error occurred

- **MESSAGE:** Error message

  This is an output parameter of the type VARCHAR(120).

25.3.14 DSNACCMD

DSNACCMD is a DB2-supplied stored procedure that can be used to issue DB2 commands and parse the output.

The syntax diagram in Figure 25-15 shows the SQL CALL statement for DSNACCMD.

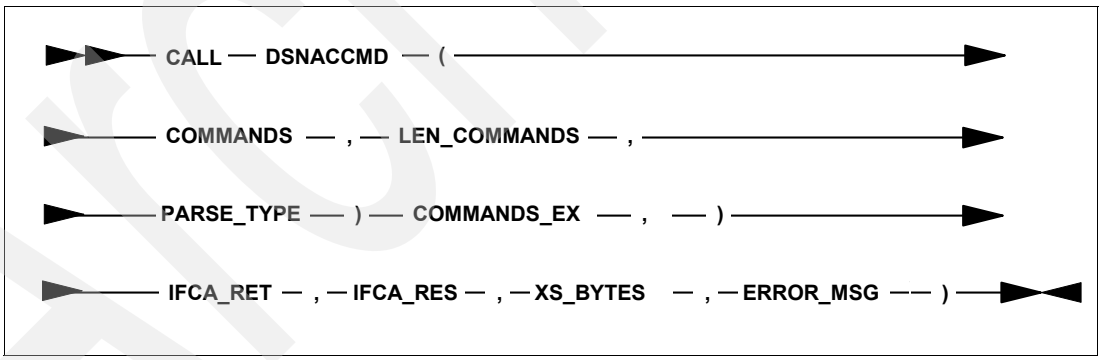


Figure 25-15 DSNACCMD syntax diagram

The possible options are:

- **COMMANDS:** Any DB2 command such as -DISPLAY THREAD(*)

  This is an input parameter of the type VARCHAR(32700).

- **LEN_COMMANDS**: Length of the command.

  This is an input parameter of the type INTEGER.

- **PARSE_TYPE**

  This is an input parameter of the type VARCHAR(3).

  Possible values are:

- "BP", if you issue -DISPLAY BUFFERPOOL
- "DB","TS","IX". if you issue -DISPLAY DATABASE, TS, IX
- "THD", if you issue -DISPLAY THREAD
- "UT", if you issue -DISPLAY UTILITY
- “NO" for any other command

If you specify a parse type, DSNACCMD parses the output and provides the result already formatted in a global temporary table.

- **COMMANDS_EX:** number of commands executure

  This is an output parameter of the type INTEGER.

- **IFCA_RET:** IFI return code

  This is an output parameter of the type INTEGER.

- **IFCA_RES:** IFI reason code

  This is an output parameter of the type INTEGER.

- **XS_BYTES:** Excess bytes

  This is an output parameter of the type INTEGER.

- **ERROR_MSG:** Error message

  This is an output parameter of the type VARCHAR(1331).

In case of error, if IFCA_RET=0 check that you have received the expected number of result sets. If IFCA_RET <> 0 check for IFIReasonCode == 15075360 && IFIExcessBytes > 0. In that case, increase the LIMIT(xxx) in your -DISPLAY command and retry.

DSNACCMD output tables

If a parse type other then NO is specified, two result sets are returned. The first result set contains IFI command messages, and the second result set contains an output table depending on the parse type specified.

The output tables have a different format in DB2 for z/OS Version 8. For example, HIPERPOOL values are no longer supported in version 8, hence, they have been removed from the BP_TBL table. The sample programs in Appendix B show how to check for the version number to build the correct SQL statement.

The bufferpool output table is shown in Example 25-6.

Example 25-6 Bufferpool output table

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.BP_TBL
   ( BP_SEQUENCE INTEGER    NOT NULL,
     BP_NAME CHAR(6) NOT NULL,
     BP_VPSIZE INTEGER NOT NULL,
     BP_VPSEQT INTEGER NOT NULL,
     BP_VPPSEQT INTEGER NOT NULL,
     BP_VPXPSEQT INTEGER NOT NULL,
     BP_DWQT INTEGER NOT NULL,
     BP_PCT_VDWQT INTEGER NOT NULL,
     BP_ABS_VDWQT INTEGER NOT NULL,
     BP_PGSTEAL CHAR(4) NOT NULL,
     BP_ID INTEGER NOT NULL,
     BP_USECOUNT INTEGER NOT NULL,
     BP_PGFIX CHAR(3) NOT NULL)
     CCSID EBCDIC;
```

The thread table is shown in Example 25-7.

Example 25-7 Thread table

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.THREAD_TBL
  ( THD_SEQUENCE INTEGER    NOT NULL,
    THD_TYPE  INTEGER NOT NULL,
    THD_NAME CHAR(8) NOT NULL,
    THD_STATUS CHAR(11) NOT NULL,
    THD_ACTIVE CHAR(1) NOT NULL,
    THD_REQ_CTR CHAR(5) NOT NULL,
    THD_CORR_ID CHAR(12) NOT NULL,
    THD_AUTH_ID CHAR(8) NOT NULL,
    THD_PNAME CHAR(8) NOT NULL,
    THD_ASID CHAR(4) NOT NULL,
    THD_TOKEN CHAR(6) NOT NULL,
    THD_COORIDINATOR CHAR(25) NOT NULL,
    THD_RESET CHAR(5) NOT NULL,
    THD_URID  CHAR(12) NOT NULL,
    THD_LUWID CHAR(35) NOT NULL,
    THD_LOCATION VARCHAR(4050) NOT NULL,
    THD_DETAIL VARCHAR(4050) NOT NULL)
    CCSID EBCDIC;
```

The utility table is shown in Example 25-8.

Example 25-8 Utility table

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.UTILITY_TBL
  ( UT_SEQUENCE INTEGER    NOT NULL,
    UT_CSECT CHAR(8) NOT NULL,
    UT_USERID CHAR(8) NOT NULL,
    UT_MEMBER CHAR(8) NOT NULL,
    UT_UTILID CHAR(16) NOT NULL,
    UT_STATEMENT INTEGER NOT NULL,
    UT_NAME CHAR(20) NOT NULL,
    UT_PHASE CHAR(20) NOT NULL,
    UT_COUNT INTEGER NOT NULL,
    UT_STATUS CHAR(18) NOT NULL,
    UT_DETAIL VARCHAR(4050) NOT NULL,
    UT_NUMOBJ INTEGER NOT NULL,
    UT_LASTOBJ INTEGER NOT NULL)
    CCSID EBCDIC;
```

The DB status table is shown in Example 25-9.

Example 25-9 DB status table

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.DBSTATUS_TBL
  ( DB_SEQUENCE INTEGER NOT NULL,
    DB_DBNAME CHAR(8) NOT NULL,
    DB_SPACENAM CHAR(8) NOT NULL,
    DB_TYPE CHAR(2) NOT NULL,
    DB_PART SMALLINT NOT NULL,
    DB_STATUS CHAR(18) NOT NULL)
    CCSID EBCDIC;
```

The IFI command message table is shown in Example 25-10.

Example 25-10 IFI command message table

```
CREATE GLOBAL TEMPORARY TABLE DSNACC.CMDMSG_TBL
  ( RS_SEQUENCE   INTEGER      NOT NULL,
    RS_DATA       CHAR( 80 )   NOT NULL )
    CCSID EBCDIC;
```

25.3.15 DSNAICUG

This DB2-supplied stored procedure can be used to provide a function which returns four types of lists:

- (a) - List of all users
- (b) - List of all groups
- (c) - List of all groups that include a user
- (d) - List of all users that belong to a group

It takes two parameters:

- The first one is the type of list to be returned.
- The second one is the user or group name for types (c) and (d).

The syntax diagram in Figure 25-16 shows the SQL CALL statement for DSNAICUG

Figure 25-16 DSNAICUG syntax diagram

The possible options are:

- **TYPE:** Integer value
  - 0: List groups
  - 1: List users
  - 2: List groups for user
  - 3: List users for group
- **KEY:** varchar(8)

  When the type is 2 or 3, provide a user or group in KEY, otherwise leave key blank.

DSNAICUG output table

DSNAICUG returns the following table:

```
( NAME VARCHAR(8) CCSID EBCDIC)
```

DSNAICUG errors

Table 25-9 shows the error messages issued by DSNAICUG.

Table 25-9 DSNAICUG errors

Exit	Return Code	Message
Normal	SQLSTATE = 00000	Message: none
	SQLSTATE = 02000 (end of input)	Message: none
Error	SQLSTATE = 38801	DSNAICUG Error: Parameter <number> missing
	SQLSTATE = 38802	DSNAICUG Error: Type of list: <type> invalid
	SQLSTATE = 38803	DSNAICUG Error: "User: <user> not found
	SQLSTATE = 38804	DSNAICUG Error: "Group: <group> not found
	SQLSTATE = 38805	DSNAICUG Error: Memory allocation error

25.4 Using the DB2 provided 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 B, “Samples for using DB2-supplied stored procedures” on page 595, and are available for download as described in Appendix D, “Additional material” on page 651. 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 DB2SystemInformation.java there, compile it with the following command:

```
javac DB2SystemInformation.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, you enter the following command:

```
java DB2SystemInformation DBALIAS USERID PASSWORD
```

Most of the programs require you to enter the database alias, the user ID, and password to run the program.

25.4.1 Source code for activating DB2-supplied stored procedures

We describe the sample source code for the other DB2-supplied stored procedures in Appendix B, “Samples for using DB2-supplied stored procedures” on page 595. The source code is available as described in Appendix D.1.8, “Sample code to invoke DB2-supplied stored procedures” on page 655.

Table 25-10 shows the functions available and where they are listed in the Appendix.

Table 25-10 Source code for DB2 stored procedures invocation

Name	Description	DB2 provided function	Source code
DB2SystemInformation	Display DB2 system information	DSNACCSS, DSNACCSI, DSNWZP, DSNUTILS,	Appendix B.1, "Display DB2 system information with DB2SystemInformation" on page 596
DB2WLMRefresh	Refresh a WLM environment	WLM_REFRESH	Appendix B.2, "Refresh a WLM environment with DB2WLMRefresh" on page 605
DB2USSUserInformation	Query the USS User Database	DSNAICUG	Appendix B.3, "Query the USS User DB with DB2USSUserInformation" on page 608
DB2Command	Issue DB2 commands	DSNACCMD	Appendix B.4, "Issue DB2 commands with DB2Command" on page 610
DB2Runstats	Automate RUNSTATS	DSNACCOR, DSNACCMO	Appendix B.5, "Automate RUNSTATS with DB2Runstats" on page 618
DB2DatasetUtilities	Manage data sets	DSNACCDS, DSNACCDR, DSNACCDE, DSNACCDL, DSNACCDD	Appendix B.6, "Manage data sets with DB2DatasetUtilities" on page 626
DB2JCLUtilities	Submit JCL	DSNACCJS, DSNACCJQ, DSNACCJF, DSNACCJP	Appendix B.7, "Submit JCL with DB2JCLUtilities" on page 632
DB2USSCommand	Issue USS commands	DSNACCUC	Appendix B.8, "Issue USS commands with DB2USSCommand" on page 639

25.5 Summary

In this chapter we have discussed the stored procedures that are shipped with DB2 and how to use them in an application program. Whether you will 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.

26

Using LOBs

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

26.1 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 *Large Objects with DB2 for z/OS and OS/390*, SG24-6571. For information on defining LOBs to DB2, see "Chapter 5" of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426. For information on how DB2 utilities manipulate LOB data, see "Part 2" of *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

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 a 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 a LOB column and the base table row is less than 32 KB, you can use the LOAD utility to put the data in DB2. Otherwise, you must use INSERT or UPDATE statements. Even though the data is stored in the auxiliary table, the LOAD utility statement or INSERT statement specifies the base table. Using INSERT can be complicated because your application will need enough storage to hold the entire value that goes into the LOB column.

26.2 Setting up the environment for sample LOB tables

DB2 provides you with sample programs and JCL to create and populate tables with LOB data. These samples can be found in <hlq>.SDSNSAMP data set. We ran the jobs listed in Table 26-1.

Table 26-1 Sample JCL for creating the LOB tables

Member Name	Type	Description
DSNTEJ7	JCL	It creates tables, table spaces and indexes to support LOB objects. Invokes the DB2 Utility to Load the CLOB data and EmployeeNo into EMP_PHOTO_RESUME table.
DSN8CLPL	COBOL This program is available in DB2 V8 only, it is similar to the C Language version DSN8DLPL available in DB2 V7 and V8.	It populates the pseg and bmp image columns in the DSN8810.EMP_PHOTO_RESUME sample LOB table. The input data is read from a TSO data set. This program demonstrates how to populate LOB columns with more than 32 KB of data.
DSN8CLTC	COBOL This program is available in DB2 V8 only, it is similar to the C Language version DSN8DLPC available in DB2 V7 and V8.	It fetches the data back from the table and verifies that it was the same as the source data.
DSNTEJ76	JCL	It prepares and runs DSN8CLPL and DSN8CLTC. Ensure that the run-time load module for the COBOL programs is defined as a PDSE.

We used the IBM supplied sample tables for our case study. The LOB table DDL used is listed in Example 26-1.

Example 26-1 LOB table used in the case study

```
CREATE TABLE  DSN8810.EMP_PHOTO_RESUME
 (      EMPNO  CHAR( 06 )   NOT NULL,
    EMP_ROWID  ROWID NOT NULL GENERATED ALWAYS,
   PSEG_PHOTO  BLOB( 500K ),
    BMP_PHOTO  BLOB( 100K ),
       RESUME  CLOB(   5K ),
  PRIMARY KEY  ( EMPNO ) )
           IN  DSN8D81L.DSN8S81B
       CCSID  EBCDIC;
```

Since the EMP_PHOTO_RESUME table has a CLOB column that is less than 32 KB, you can use the LOAD utility to load the CLOB data. The EMP_PHOTO and BMP_PHOTO columns are BLOB columns larger than 32 KB, and they require a program to be populated.

26.3 Support for LOBs in Java

Support for DB2 LOBs is added to the Version 7 JDBC 2.0 driver with APAR PQ51847. Previous versions, and the Version 7 JDBC 1.2 driver, do not support LOBs.

The maximum size allowed for a LOB parameter defined as OUT or INOUT on a CallableStatement 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.

26.4 Stored procedure returning a BLOB column

In this section we show how to implement a Java stored procedure handling a BLOB column.

26.4.1 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 listed in Example 26-2.

Example 26-2 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 DB8ADJC2
```

The sample code for the Java stored procedure is in Example 26-3.

Example 26-3 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();
        }
    }
}
```

Please be aware of the following points about the code listed in Example 26-2:

- 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 a 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 a LOB into or out of a stored procedure.
- You cannot use LOB locators as input or output parameters for stored procedures.

26.4.2 Invoking the EmpPhotJ stored procedure

We created a 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 26-4) 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 26-4 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/DB8A",
               "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 reported in Example 26-5.

Example 26-5 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 - DB8A (Location Name for OS390)
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 inputstream 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.

26.4.3 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

26.4.4 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 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 26-6 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 26-6 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 26-7 shows the DDL used for creating the stored procedure.

Example 26-7 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 DB8ADJC2
```

26.5 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 26-8.

Example 26-8 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 DB8ADJC2
```

The DDL used for creating the stored procedure can be found in Example 26-9.

Example 26-9 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 that:

- 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.

26.5.1 Invoking the EmpClobJ stored procedure

We created a Java servlet called 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 can be found in Example 26-10.

Example 26-10 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/DB8A",
                   "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());
          }

       }

    }
```

27

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 D, “Additional material” on page 651.

Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 22 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

27.1 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 the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415 for details.

Example 27-1 shows how a trigger on table EMP invokes a 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 functions are invoked with a VALUES statement.

Example 27-1 Trigger invoked with VALUES

```
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
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.10)
BEGIN ATOMIC
 VALUES DEVL7083.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY);
END#
```

An alternative way to invoke a user defined function from a trigger conditionally based on the number of rows of a table is to issue a SELECT statement against the transition table. Transition tables are discussed in 27.2.2, “Using transition tables” on page 454.

Similarly, Example 27-2 shows a call to a stored procedure EMPAUDTS under the same conditions, and passing the same parameters. In this case, the stored procedure is invoked with a CALL statement instead of a VALUES statement for a user defined function.

Example 27-2 Trigger invoked with CALL

```
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 parameters of a stored procedure call must be literals, transition variables (see 27.2.1, “Using transition variables” on page 453), transition tables (see 27.2.2, “Using transition tables” on page 454), 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 (see 27.3, “Error handling in triggers” on page 456) before making the decision. We discuss this in 27.4, “Stored procedures versus user defined functions” on page 456.

27.2 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 27.2.1, “Using transition variables” on page 453. Transition tables refer to the complete set of values of all affected rows, and this is discussed in 27.2.2, “Using transition tables” on page 454.

Table 27-1 summarizes the allowable combinations of transition variables and transition tables that you can specify for the various trigger types.

Table 27-1 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	-
		UPDATE	OLD,NEW	
		DELETE	OLD	-
	AFTER	INSERT	NEW	NEW TABLE
		UPDATE	OLD,NEW	OLD TABLE, NEW TABLE
		DELETE	OLD	OLD TABLE
FOR EACH STATEMENT	BEFORE	INSERT	-	-
		UPDATE	-	-
		DELETE	-	-
	AFTER	INSERT	-	NEW TABLE
		UPDATE	-	OLD TABLE, NEW TABLE
		DELETE	-	OLD TABLE

27.2.1 Using transition variables

Transition variables are similar to host variables in their behavior. A transition variable can be referenced in the search-condition of a triggered SQL statement of the triggered action wherever a host variable is allowed in the statement if the statement were issued outside the

body of a trigger. 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 update) or to the new value (after the update). In Example 27-3 we use O and N as the qualifiers to designate the before and after values.

Example 27-3 Trigger with before and after values

```
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.
```

27.2.2 Using transition tables

Transition tales are also similar to host variables in their behavior. The name of the transition table can be referenced in a triggered SQL statement of the triggered action wherever a table name is allowed in the statement if the statement were issued outside the body of a trigger. The name of the table can be specified in the search condition of a triggered SQL statement of the triggered action wherever a column name is allowed in the statement if the statement were issued outside the body of a trigger.

Transition tables are read-only. Like transition variables, transition tables also use the names of the columns of the subject table, but have a name specified that allows the complete set of affected rows to be treated as a table.

In Example 27-4 we use OT and NT as the qualifiers to designate the before and after table values.

Example 27-4 Trigger with transition tables

```
CREATE TRIGGER DEVL7083.EMPTRIG3
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.EMPPROPS(TABLE OT,TABLE NT);
END#
```

We describe below 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 UDB for z/OS Version 8 SQL Reference,* SC18-7426 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 27-5 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 27-5 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 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415 for information on the syntax for C, C++, COBOL, and PL/I. See Chapter 6 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426 for information on the syntax for SQL procedures. This is shown in Example 27-6.

Example 27-6 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 27-7.

Example 27-7 Declaring a cursor

```
****       CURSOR FOR RETRIVEING "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 27-8.

Example 27-8 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 27-9.

Example 27-9 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.
```

27.3 Error handling in triggers

Severe errors that occur during the execution of a trigger 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.

The ability to handle errors in a trigger is severely limited, especially when it calls a stored procedure. For this reason, you must decide carefully whether you use a stored procedure or a user defined function. This is discussed next in 27.4, “Stored procedures versus user defined functions” on page 456.

27.4 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 27-1 shows how a user defined function can be used for data validation.

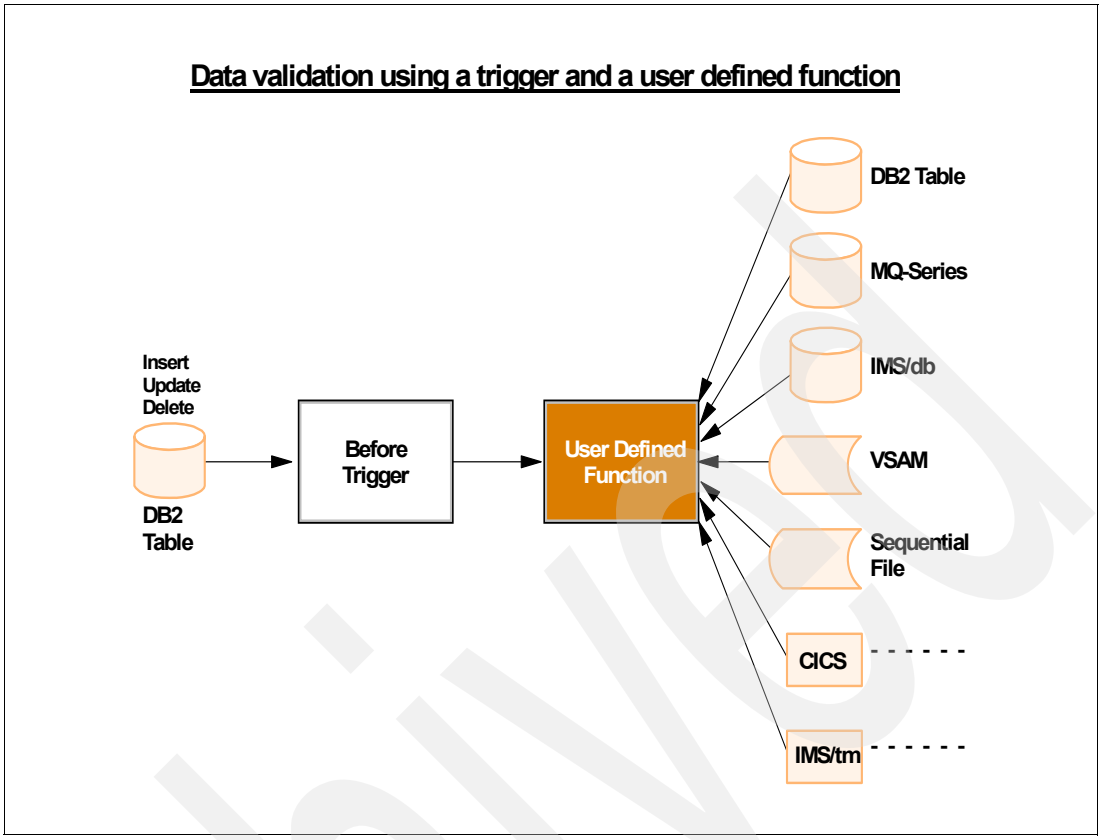


Figure 27-1 Data validation using a trigger and a user defined function

Example 27-10 shows how a user defined function can pass an output parameter back to the trigger. In this case, the user defined function sets the PRTNCD parameter.

Example 27-10 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 27-11 shows how you can use the result of a user defined function to return different error messages to the calling application.

Example 27-11 Generating error messages in a trigger

```
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 SQLSTATE value specified in the SIGNAL SQLSTATE statement 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 27-2 shows how a stored procedure can be used for data propagation.

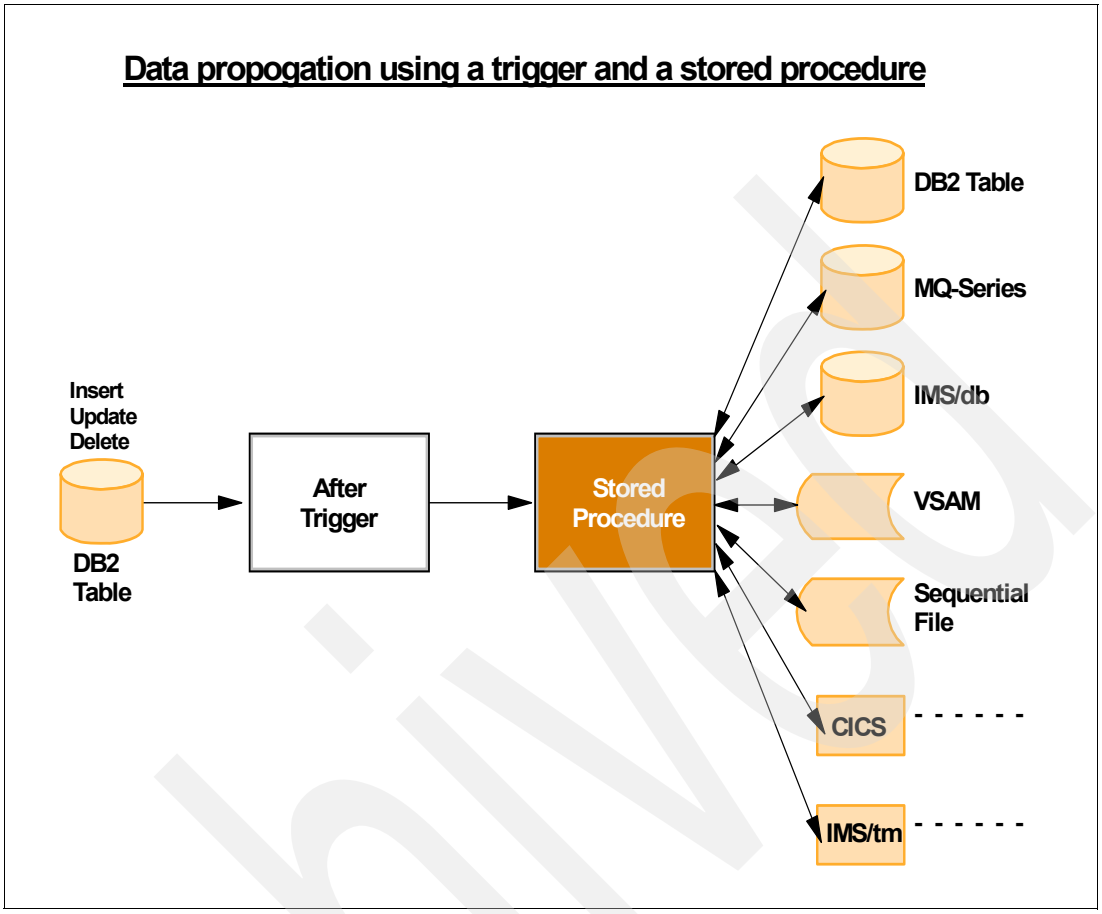


Figure 27-2 Data propagation using a trigger and a stored procedure

27.5 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 contains the invocation must have EXECUTE authority for the packages that the other programs use.

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), and this is discussed in 10.3.1, “Nested stored procedures” on page 111.

27.6 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), and this is discussed in 10.3.1, “Nested stored procedures” on page 111.

Part 7

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:

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 redbook 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
- Debugging SQL SPs on z/OS, Windows, UNIX, and Linux
- Debugging COBOL, PL/1, C/C++ SPs on z/OS
- Debugging options for DB2 Java SPs on z/OS
- Debugging Java SPs on Windows, AIX, and Sun

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 and OS/390

Language	DB2 Release	Client	Server
COBOL, PL/1, C/C++	All supported DB2 releases	Optional GUI, IBM Distributed Debugger (Windows, AIX, Sun)	DB2 IBM Debug Tool
COBOL, PL/1, C/C++	All supported DB2 releases	Optional GUI, IBM WebSphere Studio Enterprise Developer,(WSED) for Windows and Linux	DB2 IBM Debug Tool
SQL	DB2 V8	Requires DB2 Development Center, included in the DB2 UDB V8.1.4 Application Development Client at FixPak 4 or higher (Windows, AIX, Linux)	DB2 V8 DSNTPSMP set up including all prerequisites (C compiler, WLM, RRS, REXX) DSN810.SDSNSAMP (DSNTIJSD)
Java	N/A	See “Debugging options for DB2 Java SPs on z/OS” on page 494	

The **IBM Distributed Debugger** graphical user interface (GUI) is a cross-platform, multi-language Debugger used for debugging DB2 Java stored procedures on the distributed platforms. Additionally, it can be used in conjunction with the IBM Debug Tool product for debugging DB2 COBOL, C, C++, or PL/1 stored procedures on z/OS.

The I**BM WebSphere Studio Enterprise Developer** V5 (WSED) provides an application development framework for COBOL, PL/1, and Assembler. It is possible to debug C and C++ stored procedures using the Debug Perspective. WSED can be used with the IBM Debug Tool for debugging COBOL, PL/1, C, and C++ stored procedures, and provides the GUI on the workstation that would be used in lieu of the IBM Distributed Debugger.

The **Debug Tool** runs on the OS/390 or z/OS server.

The Distributed Debugger and WSED client interfaces run on the workstation.

DSNTPSMP is the REXX DB2-supplied stored procedure that builds SQL stored procedures on DB2 for z/OS V8 (see 29.1, “Development Center start up” on page 500 for details).

Table 28-2 summarizes the DB2 debugging options for the distributed platforms. While this redbook 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 DB2 Development Center 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 DB2 debugging options for the distributed platforms

Language	DB2 Release	Client	Server
SQL	V7.2	DB2 Stored Procedure Builder (Windows, AIX, Sun)	DB2 V7.2 (Windows, AIX, Sun, HPUX, Linux) C compiler
Java	V7.2	DB2 Stored Procedure Builder (Windows, AIX, Sun) IBM Distributed Debugger (Windows, AIX, Sun)	DB2 V7.2 (Windows, AIX, Sun) JDK™ 1.1.8 or JDK 1.2.2
SQL	V8.1	DB2 Development Center (Windows, AIX, Linux, Sun)	DB2 V8.1 (Windows, AIX, Sun, HPUX, Linux (Intel® and zSeries)) C compiler DB2 V8.1 ADC
Java	V8.1	DB2 Development Center (Windows, AIX, Sun) IBM Distributed Debugger (Windows, AIX, Sun)	DB2 V8.1 (Windows, AIX, Sun) JDK 1.3 or JDK 1.4
SQL	V8.1	WSAD, WSADIE or WSED V5 (Windows and Linux)	DB2 V8.1 (Windows, AIX, Sun, HPUX, Linux (Intel and zSeries)) C compiler DB2 V8.1 ADC

The integrated SQL Debugger is required to debug SQL stored procedures, and is included in the DB2 Stored Procedure Builder with DB2 V7.2 and the DB2 Development Center with DB2 V8.1. The Distributed Debugger is not used for debugging SQL stored procedures.

See the Stored Procedure Builder (V7.2) or Development Center (V8.1) online help for assistance in debugging SQL or Java stored procedures. Here are some of the options and settings that are described in the Development Center online help:

- Debugging SQL stored procedures
- Debugging Java stored procedures
- Specifying the SQL Debugger daemon location for debugging stored procedures
- Building a routine for debugging
- Running a stored procedure in debug mode
- Setting breakpoints
- Changing variable values
- Using the Call Stack
- Statements that support Debugger breakpoints

28.2 Debugging SQL SPs on z/OS, Windows, UNIX, and Linux

The Development Center with FixPak 4 and later versions provides the ability to debug SQL stored procedures on DB2 for z/OS V8. The SQL Debugger was introduced on the distributed platforms of Linux, UNIX, and Windows, with the Stored Procedure Builder in DB2 UDB V7.2. The same SQL debugging user interface is expanded for use on z/OS with DB2 for z/OS V8. The examples shown in this section also apply when debugging SQL stored procedures on Windows, UNIX, and AIX.

This section describes how to get you started with setting up the debugging environment:

1. Overview of debugging SQL stored procedures with the Development Center Integrated SQL Debugger
2. Prerequisites and setup:
   - Workstation
   - z/OS
3. Creating SQL stored procedures for debugging
4. Debugging SQL stored procedures
5. Defining the EMPDTLSS SQL case study for debugging
6. Debugging the EMPDTLSS SQL case study

28.2.1 Overview of the Development Center Integrated SQL Debugger

The Development Center is required for SQL stored procedure debugging. First, you start Development Center, then create a new or edit an existing SQL stored procedure, and build the stored procedure for debug. Once the *build for debug* is successful, select the **stored procedure** and right-click **Debug**. This starts the SQL Debug daemon on the workstation. The current workstation IP address is used. The default listening port is 4553, which can optionally be changed.

The chart in Figure 28-1 describes the processing flow for debugging SQL stored procedures using the Development Center Integrated SQL Debugger.

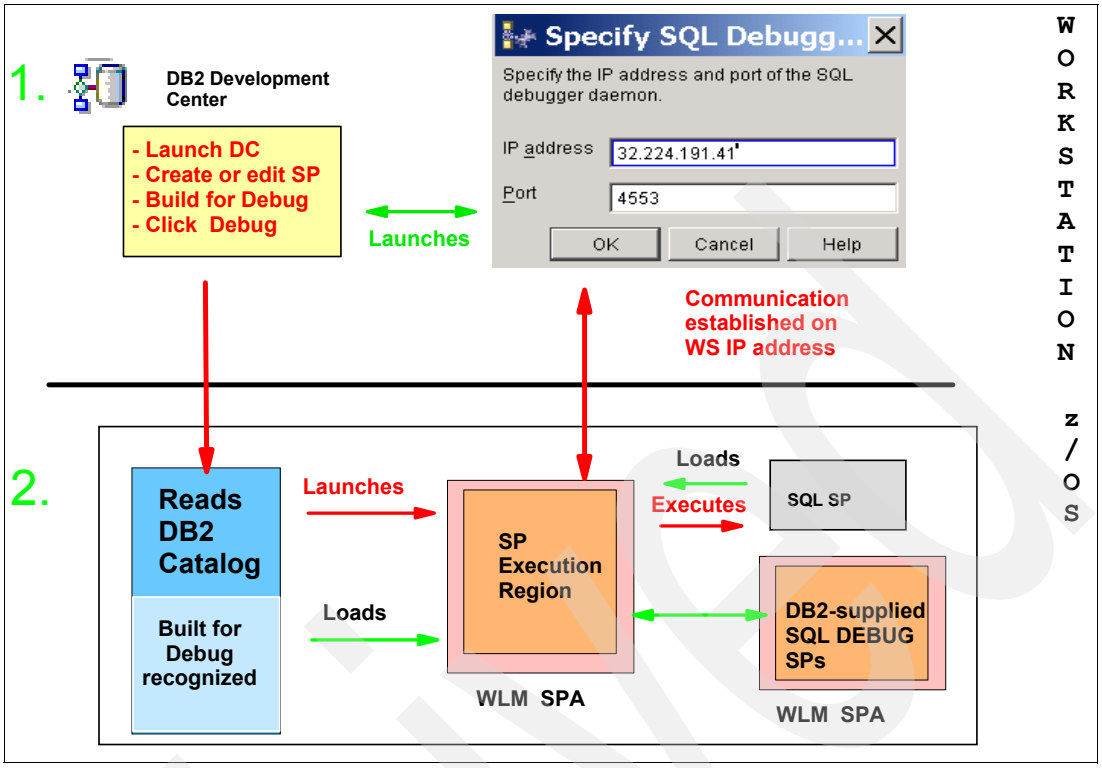


Figure 28-1 Processing Overview - SQL Debugger with DB2 V8 for z/OS

28.2.2 Prerequisites and set up

The prerequisites and setup for debugging SQL stored procedures on z/OS are:

- Workstation
  - DB2 Connect
  - DB2 Development Center UDB V8.1.4 (FixPak 4)
- z/OS
  - DB2 V8 for z/OS
  - DSNTPSMP
  - C Compiler
  - <hlq>.SDSNSAMP(DSNTIJSD) customization job. Six new DB2-supplied stored procedures are included in this job. It is recommended to process these stored procedures in the same WLM AE with the DB2-supplied stored procedures used by SQL Assist set up in <hlq>.SDSNSAMP(DSNTIJMS). This job includes support for the Debugger providing:
    - New DDL definitions
    - New BINDs
    - New authorizations
  - A WLM procedure must be defined for executing the SQL stored procedure. This WLM procedure optionally includes a //PSMDEBUG statement used to collect information when debugging problems with the SQL Debugger. The //PSMDEBUG statement defines a physical sequential data set with RECFM=VBA, LRECL=4096. This datatset should only be included in the WLM proc 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.

28.2.3 Creating SQL stored procedures for debugging

DB2 Development Center can be used to create the SQL stored procedure, though it is not required. However, DC 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 the Development Center (DC) from **Start -> Programs -> IBM DB2 -> Development Tools -> Development Center.**
2. Create a new or open an existing project.
3. Connect to a DB2 for z/OS V8 server.
4. Optionally update the SQL Debugger timeout value

   The default timeout value is 60 seconds. This means that while in SQL Debugger mode, once 60 seconds of inactivity occurs, the Debugger will terminate and any locks held by the stored procedure being debugged will be released, and the SQL stored procedure will run to completion.

   This value can be changed from the Environment Settings. Select the SQL Debugger window from **Project -> Environment Settings -> SQL Debugger ->Timeout value** in seconds. We modified this value to 120 seconds for our case study.
5. Create a new or edit an existing stored procedure.
6. Build the stored procedure for debug. This can be done in one of three ways:
   - From the tool bar, click the DC Editor **Build for Debug** icon (wrench with a bug); see Figure 28-2.

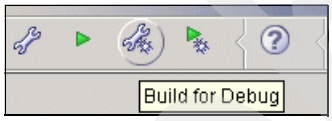


Figure 28-2 Building the SQL stored procedure for debug using the Build for Debug icon

   - During initial creation of the SQL stored procedure when using the wizards: This is done on the Options window by checking the **Enable debugging** check box.
   - After the stored procedure has been successfully created without debug, select the **stored procedure**, and right-click **Build for Debug.**

The DC Output window includes a message indicating if the build utility performed the BUILD_DEBUG function, and whether it was successful or not, as shown in Figure 28-1.

Example 28-1 BUILD_DEBUG function was completed successfully

```
Build utility function requested: BUILD_DEBUG
DSNT540I  DB2GDES1 WAS REFRESHED BY PEGGYR   USING AUTHORITY FROM SQL ID PEGGYR  : 0
DEVL7083.EMPDTLSS - Build successful.
```

28.2.4 Debugging SQL stored procedures

Once the SQL stored procedure has been successfully built for debug mode, we can debug the stored procedure. The Debugger can be launched in the following ways:

- From Project View, select the **SQL stored procedure** and right-click **Debug**.
- From Project View, select the **SQL stored procedure** and right-click the **Run with Debug** icon from the tool bar, as shown in Figure 28-3.

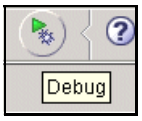


Figure 28-3 Launching SQL Debugger using the toolbar with the Debug icon

- With the SQL stored procedure opened in Editor View, which can be done by selecting the **SQL stored procedure** in Project View, right-click **Edit**. The stored procedure is now in Editor View. Select the **Debug** icon from the tool bar to start debugging.

When the stored procedure is launched in debug mode, the editor window opens allowing you to set breakpoints in the prefix area to the left of a valid statement, monitor variables, and interactively debug.

28.2.5 Defining the EMPDTLSS SQL case study for debugging

EMPDTLSS was initially created without Development Center directly on DB2G, the DB2 for OS/390 V7 server described in 13.2.6, “Passing parameters” on page 166.

We copied the source of this stored procedure to our workstation allowing us to create the same stored procedure on DB8A, our DB2 for z/OS V8 server. We made the following changes to this stored procedure before using Development Center to build and execute the stored procedure on our DB8A, DB2 for z/OS V8 server:

- Qualify the EMP table using DSN8810 for the DB2 for z/OS V8 server.
- Change the WLM ENVIRONMENT to point to DB8ADS1, the WLM proc for DB8A.
- We encountered an error in Development Center when PROGRAM TYPE MAIN is included in the CREATE PROCEDURE definition; see Example 28-2. That option is the default on z/OS, so we commented the statement to get past the error. This defect is being corrected in a future DC FixPak.

Example 28-2 Modified EMPTDTLSS source for Development Center to build/debug on DB8A

```
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
NO DBINFO
WLM ENVIRONMENT DB8ADS1
STAY RESIDENT NO
COLLID DEVL7083
--PROGRAM TYPE MAIN
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF)'
COMMIT ON RETURN NO
LANGUAGE SQL
BEGIN
```

```
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
SELECT
       FIRSTNME
     , MIDINIT
     , LASTNAME
     , WORKDEPT
     , HIREDATE
     , SALARY
INTO   PFIRSTNME
     , PMIDINIT
     , PLASTNAME
     , PWORKDEPT
     , PHIREDATE
     , PSALARY
FROM   DSN8810.EMP
WHERE  EMPNO = PEMPNO
;
 SET PSQLCODE   = SQLCODE ;
 SET PSQLSTATE  = SQLSTATE;
 SET PSQLERRMC  = 'ADIOS';
END
```

This SQL stored procedure can be brought into Development Center in one of two ways:

- Using the Import wizard
- Using the editor

Both ways are shown next.

Using the Import wizard

With DC started, and the DEVL7083 project open with a connection to DB8A, select the **Stored Procedure** folder and right-click **Import**, as shown in Figure 28-4.

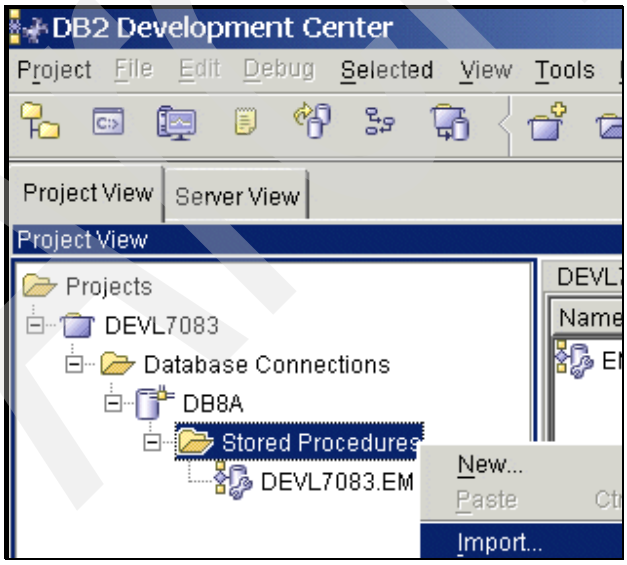


Figure 28-4 DC Import wizard for SQL EMPDTLSS

The Import wizard is opened and the Import Stored Procedure window is displayed. Select **File System -> Source file ->OK** as shown in Figure 28-5.

Figure 28-5 Select File System->Source file

The Import wizard includes the following six steps to be completed. The wizard parses the imported source to fill-in the appropriate fields when possible:

- Source File

  Click the **Browse** button 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 is listed and nine output parameters are listed. Click **Next** to continue.

- Stored Procedure Name

  The stored procedure name is automatically filled in as EMPDTLSS in this window. Click **Next** to continue.

- Options

  The COLLID of DEVL7083 is automatically filled in from our source. Next, select the **Advanced** button on this window, which opens the z/OS Options window. The WLM ENVIRONMENT name we included in our source DB8ADS1 is automatically filled in. 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.

- Summary

  The above settings are summarized. Click **Finish** to build the stored procedure for debug.

Using the editor

With DC started and the DEVL7083 project open with a connection to DB8A, select the **Stored Procedures folde**r and right-click **New** and then click **SQL Stored Procedure** as shown in Figure 28-6.

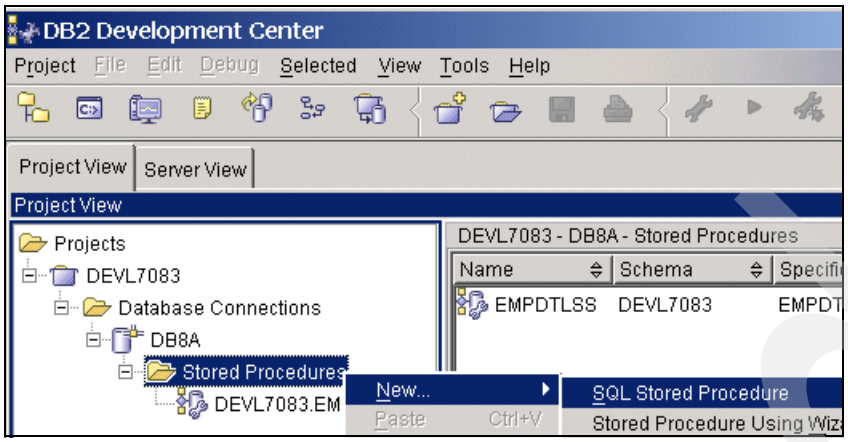


Figure 28-6 Create new SQL stored procedure

Selecting this option opens the Editor View where we delete the template syntax in the editor, and copy and paste our EMPDTLSS syntax from our workstation. Since our CREATE PROCEDURE parameters included COLLID and WLM ENVIRONMENT, there are no additional changes to make. Then select the **Build for Debug** icon from the tool bar to start the build process. Since we did not save our changes before selecting the Build for Debug icon, the editor automatically prompts us if we want to `procedure has been changed save the changes` before building, which we reply yes. Now the build process starts, and DSNTPSMP is called on DB8A to build EMPDTLSS for debugging.

Whether we used the Import wizard, or the New->SQL stored procedure 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, DB8ADS1 where EMPDTLSS executes is automatically refreshed to pick up our latest changes. DC returns the information in Example 28-3 in the DC Output View, Messages window.

Example 28-3 BUILD_DEBUG successful

```
Build utility function requested: BUILD_DEBUG

DSNT540I  DB8ADS1 WAS REFRESHED BY PEGGYR   USING AUTHORITY FROM SQL ID PEGGYR  : 0
DEVL7083.EMPDTLS3 - Build successful.
```

Using the SQL Debugger

We only mention some starting information for the SQL Debugger. There are numerous articles about the SQL Debugger on the IBM Developerworks Web site for DB2. The following article applies to the SQL Debugger in DB2 UDB V7.2 and DB2 UDB V8.1:

`http://www.ibm.com/developerworks/db2/library/techarticle/alazzawe/0108alazzawe.html`

Next, we describe the toolbars as they relate to the SQL Debugger.

Debug toolbars

The SQL Debugger graphical interface is made up of the following related toolbars:

- **Build and Run toolbar** for the build and run in debug or non-debug modes, as well as pause and run to completion for debug mode
- **Execution toolbar** for the various debug execution step commands including: step into, step over, step return, and run to line (cursor)

- **Break points toolbar** for the various debug break point commands including, set, toggle (enable/disable), remove, and remove all

Build and Run toolbar

The DB2 Development Center Build and Run toolbar includes SQL Debugger commands for building and running a stored procedure in debug mode as illustrated in Table 28-3.

Table 28-3 Build and Run toolbar

Icon	Command description
	Build (compile) selected stored procedure in debug mode.
	Run or resume running of stored procedure in debug mode.
	Pause (break) execution of stored procedure at next possible line.
	Resume running of stored procedure to completion, ignoring all break points.

Execution toolbar

The Debugger execution toolbar includes the debug step commands illustrated in Table 28-4.

Table 28-4 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.
	Run to line (cursor) causes the stored procedure being debugged to run and break at the line where the cursor is currently positioned unless an earlier break point is encountered.

Break points toolbar

The Debugger break points toolbar incorporates the Debugger break point management commands illustrated in Table 28-5.

Table 28-5 Break points toolbar

Icon	Command description
	Add new break point for the currently selected source code line or add a variable value change event break point for the currently selected variable.
	Toggle the state of a break point between enabled and disabled.
	Delete the currently selected break point.
	Delete all break points for the current SQL stored procedure.

Debugger views

The SQL Debugger graphical interface is made up of the following related views:

- Editor View: Shows the SQL code
- Break Points View: Shows the list of break points currently set
- Call-Stack View: Shows the list of nested stored procedures
- Variables View: Shows the list of defined variables
- Status Bar: Shows the occurrence of SQL exceptions

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 Editor View. Switching to a different procedure in either the project tree or the call-stack view causes the break points and variables view to display the debug data for the newly selected stored procedure code that is shown in the Editor View.

Valid SQL Debugger breakpoint statements

The Development Center SQL 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 28-6.

Table 28-6 Valid SQL Debugger breakpoint and change variable statements

Category of statements	Statements

Statements that accept breakpoints (highlighted statements):	ALLOCATE CURSOR
	ASSOCIATE LOCATORS
	CASE (EXPRESSION)
	COMMIT
	CREATE PROCEDURE (1st line highlighted)
	CREATE <table, view, index
	DEFAULT (VALUE)
	DROP <table, view, index>
	ELSEIF (EXPRESSION)
	EXECUTE
	EXECUTE IMMEDIATE
	FETCH <..> INTO FOR v1 AS <SQLstmt>
	GET DIAGNOSTICS
	GOTO(LABEL)
	IF (EXPRESSION)
	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 nothing	BEGIN
	BEGIN NOT ATOMIC
	BEGIN ATOMIC
	CLOSE CURSOR
	DECLARE cursor WITH RETURN FOR <sql statement>
	DECLARE , var without default
	DECLARE CONDITION (CONDITION) FOR SQLSTATE (VALUE) "..."
	DECLARE CONTINUE HANDLER
	DECLARE CURSOR
	DECLARE EXIT HANDLER
	DECLARE RESULT_SET_LOCATOR [VARYING]
	DECLARE SQLSTATE
	DECLARE SQLCODE (unless there is a default)
	DECLARE UNDO HANDLER (unless they are entered)
	DO
	ELSE
	END
	END CASE
	END IF
	END FOR
	END REPEAT
	END WHILE
	ITERATE
	LEAVE
	LOOP
	OPEN CURSOR
	REPEAT (as a keyword alone)
	RESIGNAL
	SIGNAL
	THEN
	labels, e.g. P1: :

28.2.6 Debugging the EMPDTLSS SQL case study

Launch the SQL Debugger from DC using any of the following instructions:

- **Project View -> Database Connections -> DB8A ->Stored Procedures->DEVL7083.EMPDTLSS** and right-click **Debug**
- **Project View -> Database Connections -> DB8A ->Stored Procedures ->DEVL7083.EMPDTLSS ->Edit,** and select the **Debug icon** from toolbar
- **Project View -> Database Connections -> DB8A -> Stored Procedures ->DEVL7083.EMPDTLSS**, and select the **Debug icon** from toolbar

Figure 28-7 shows starting debug mode using the method 1 above.

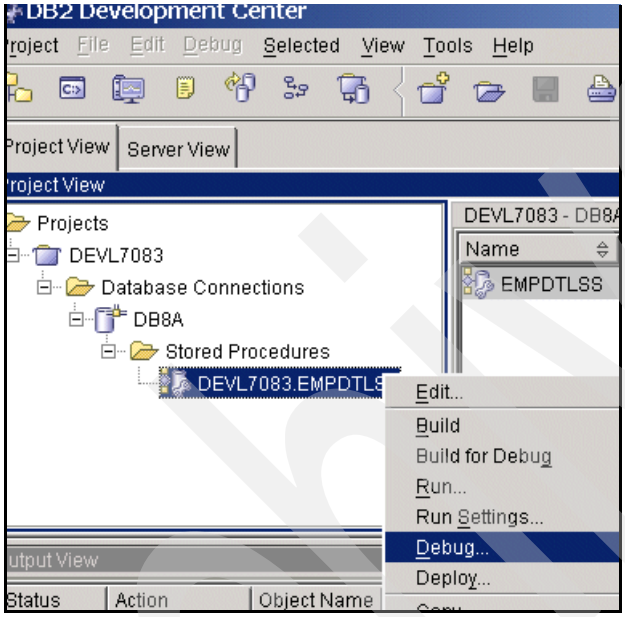


Figure 28-7 Start Debugger for EMPDTLSS

Once we select **Debug,** the SQL Debugger daemon is returned and our workstation IP address is filled in along with the default port of 4553. See Figure 28-8. The default port can be changed, but the IP address needs to be your workstation IP as it appears when typing `IPCONFIG` from a DOS command window. Furthermore, only a single IP network is supported in this release of the SQL Debugger. Specifically, home or multiple node networks are not supported.

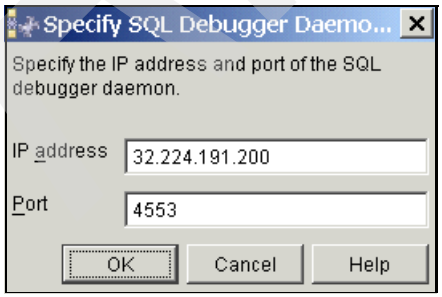


Figure 28-8 SQL Debugger daemon

Once the daemon is started, the Editor View opens displaying the EMPDTLSS source as shown in Figure 28-9. When the first line of our SQL stored procedure appears (highlighted in yellow*)* we are ready to start debugging.

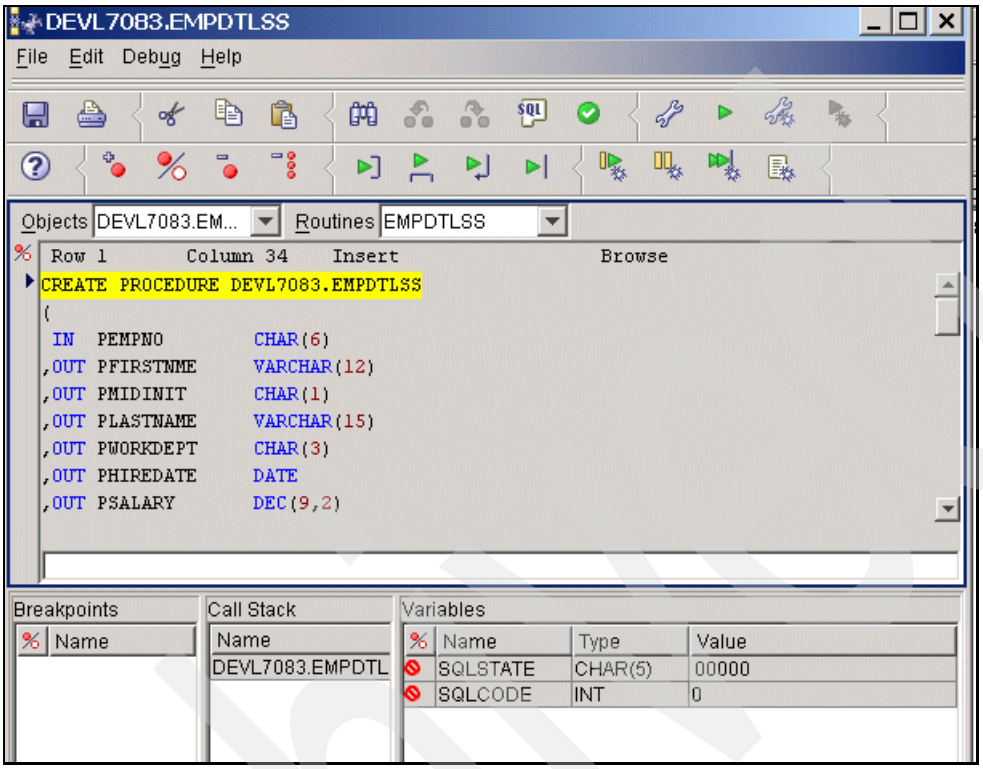


Figure 28-9 EMPDTLSS in the DC editor in Debug mode

Next, we set a breakpoint on the `DECLARE SQLSTATE CHAR(5)` statement:

1. Locate the **Select**, and place the cursor in the left-hand prefix.
2. Right-click to set the breakpoint. See Figure 28-10. The Development Center Debugger highlights certain statements during a debug session.

The highlighted statements are the only locations that you can step into or set breakpoints on. See Table 28-6 on page 475 for a list of these.

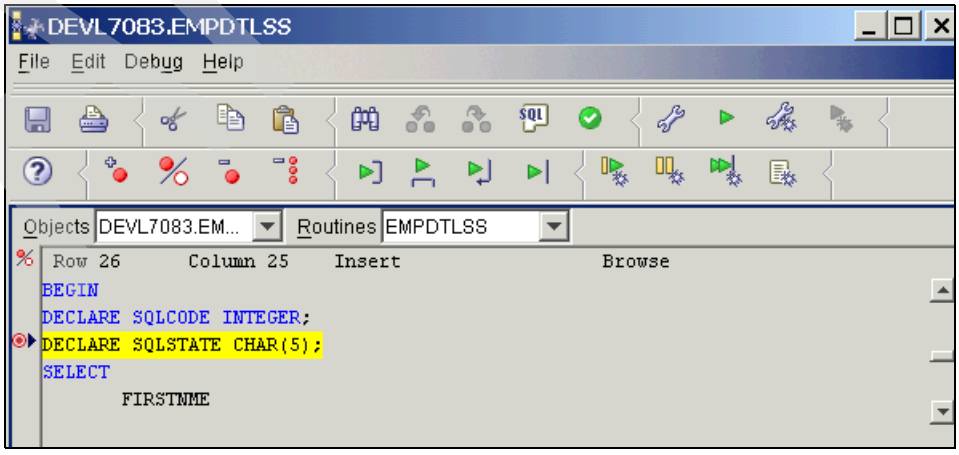


Figure 28-10 Set breakpoint on SELECT statement on Row 27

To run to this breakpoint, we select the **Step Over** 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 progressed 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 28-11.

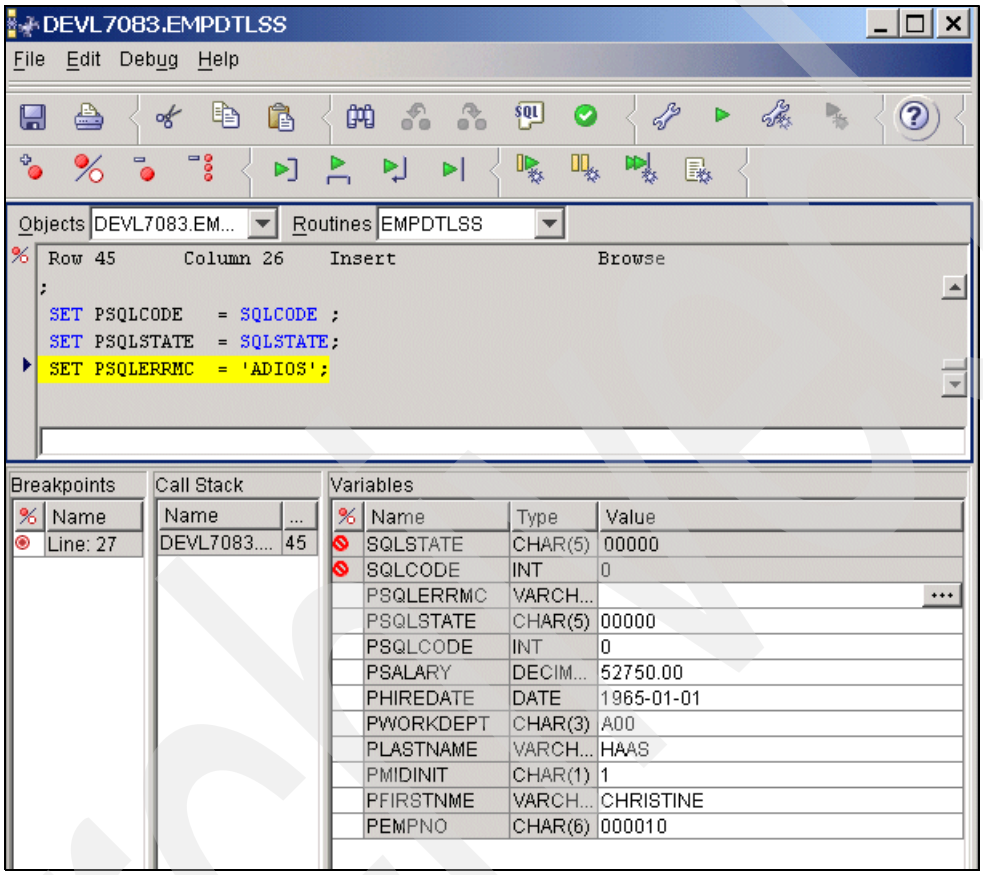


Figure 28-11 SQL Debugger Breakpoints, Call Stack, Variables display

For more information on the SQL Debugger, see the on-line Help included with Development Center UDB V8.1.4 (FixPak 4).

28.3 Debugging COBOL, PL/1, C/C++ SPs on z/OS

Our COBOL debugging example used the case study for this section used the COBOL stored procedure, DEVL7083.EMPDTLSC, created in Example 10-1 on page 95. Similar setup steps described in this section can be used for debugging PL/1 and C/C++ stored procedures.

COBOL, PL/1 and C/C++ require the IBM Debug Tool product on z/OS. Optionally, one of two client tools can be used. In this section we describe the steps to get you started:

1. Overview of debugging COBOL stored procedures with the IBM Distributed Debugger
2. Prerequisites and setup
   - Workstation
   - z/OS
3. Creating COBOL stored procedures for debugging
4. Debugging COBOL stored procedures:

– Using the IBM Distributed Debugger client (optional - no charge)
– Using WSED client (optional - fee based)
– Using TSO (3270 interface only)

28.3.1 Overview of debugging COBOL SPs with the IBM Distributed Debugger

The chart in Figure 28-12 describes the processing flow for debugging COBOL stored procedures using the IBM Distributed Debugger.

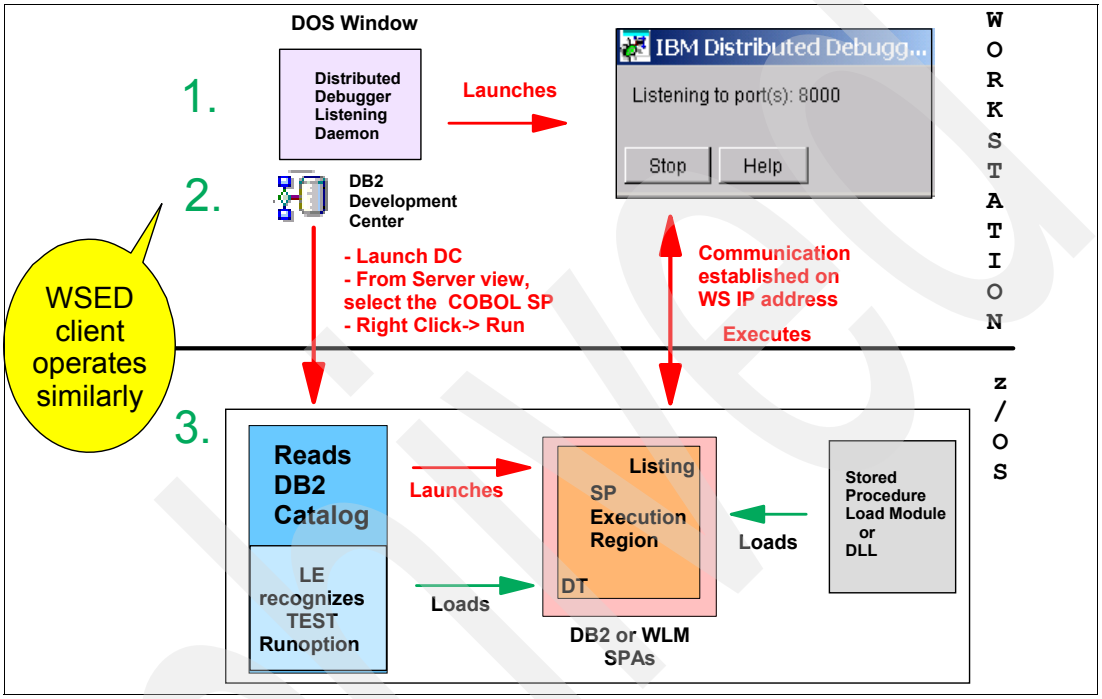


Figure 28-12 Processing Overview - COBOL and the Distributed Debugger

The chart shows how the Development Center, Debug Tool (DT), and the IBM Distributed Debugger interact to launch and invoke the IBM Distributed Debugger on the workstation and Debug Tool (DT) on z/OS and OS/390, accessing the language listing file. Using the WSED client operates similarly on the workstation and exactly the same on z/OS:

1. The IBM Distributed Debugger, (IDEBUG) daemon is started directly from a DOS command prompt. Alternatively, some workstation language installations add a Start Menu option to activate it.
2. The DB2 Development Center (DC) is used to launch the stored procedure. Alternatively, DB2 Stored Procedure Builder (SPB) can be used, though this redbook does not describe using the DB2 SPB client. Using either DC or SPB is not required. A user program could be written to call the stored procedure.
3. 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.

28.3.2 Prerequisites and set up

Next, we describe the prerequisites for the workstation and z/OS environments.

Workstation

No client tool is required to debug COBOL, PL/1, or C/C++ stored procedures on z/OS. Only the fee based product called IBM Debug Tool on z/OS is required, which can use a 3270 TSO interface for the debugging. See 14.5, “IBM Debug Tool” on page 195 for more information on Debug Tool, and 14.5.2, “IBM Debug Tool on z/OS, VTAM MFI example” on page 198.

We used both the IBM Distributed Debugger client and the WebSphere Studio Enterprise Edition clients in our case study as this made the debugging very easy for an application developer used to workstation Debuggers. DB2 Connect is required when using either client tool as described in this section.

- IBM Distributed Debugger (optional - no charge)

  We used Version 9.2 of the Distributed Debugger. The IBM Distributed Debugger code is delivered with the following IBM high-level language compilers:

  - Enterprise COBOL for z/OS and OS/390
  - COBOL for OS/390 and VM or COBOL for MVS and VM

  Additionally, the IBM Distributed Debugger code is included in many of the DB2 UDB V8.1 editions, and the following VisualAge products:

  - C, C++ for MVS and VM or OS/390 C, C++
  - Enterprise PL/I for z/OS and OS/390 or PL/I for MVS and VM
  - VisualAge for Java, Enterprise Edition for OS/390
  - IBM VisualAge PL/I Enterprise for Windows
  - IBM VisualAge COBOL for Windows NT
  - IBM C/C++ Productivity Tools for OS/390

  It is also included as a no-charge download from the Web site:

  - http://www7b.boulder.ibm.com/dmdd/library/techarticle/0305rader/0305rader.html

- WebSphere Studio Enterprise Developer (optional - fee based)
- We used DB2 UDB V8.1 Enterprise Server Edition (ESE), which contains a Personal Edition of DB2 Connect V8. If DB2 Connect is not currently installed, a restricted use license included in the z/OS Management Client Package can be used for development purposes. FMIDs for the z/OS DB2 Management Client Package follow:

  - JDB661D - DB2 for OS/390 V6
  - JDB771D - DB2 for OS/390 V7
  - JDB881D - DB2 for z/OS V8

Both the IBM Distributed Debugger and WebSphere Studio Enterprise Developer (WSED) are optional client tools that work in conjunction with the IBM Debug Tool on the z/OS server to debug COBOL, PL/1, or C/C++ stored procedures. If a client tool is being used, then DB2 Connect is also required. We used DB2 Connect at a V8.1 level since our case study used Development Center to launch the COBOL stored procedure. While this redbook does not discuss using the DB2 Stored Procedure Builder included in the DB2 UDB V7.2 Application Development Client, it can optionally be used as well, which will require DB2 Connect V7.2 or higher.

Installing the Distributed Debugger

We took the defaults when installing the Distributed Debugger with the exception that we selected a *full* install instead of a *typical* install. The selected options are listed in Table 28-7.

Table 28-7 Installing the IBM Distributed Debugger code

Window	Option selected
Welcome window	Next
Set up Type	We selected Full (default is Typical)
Choose Default Location	Default c:\IBMDebug folder used
Start Copying Files	Next

WSED used for debugging from the client

Alternatively, the workstation product WebSphere Studio Enterprise Developer (WSED) at a V5 or higher level works in conjunction with Debug Tool for debugging COBOL, PL/1, and C/C++ stored procedures on z/OS or OS/390.

Note: When using WSED for the client, PTF UQ77541 is required on z/OS.

z/OS or OS/390

The products needed on the host are the LE run-time and the Debug Tool (DT) for z/OS and OS/390 library. More information about Debug Tool can be found at the Debug Tool for z/OS and OS/390 Web site:

`http://www.ibm.com/software/awdtools/debugtool/about/`

- LE run-time
  - hlq.SCEERUN
- Debug Tool (DT) library
  - hlq.SEQAMOD

Once Debug Tool is installed on z/OS, it needs to be enabled. This enablement is done by updating the SYS1.PARMLIB member IFAPRDxx, where xx is the active suffix on your system. The syntax of this update is described in the figure below. If multiple languages, COBOL, PL/1 and C/C++ are being debugged with Debug Tool, only one SYS1.PARMLIB(IFAPRDxx) member needs to be updated. See Info APAR PQ27840 for further information.

Optionally: Use a WLM AE set up for debugging

Since Debug Tool is loaded in the WLM environment when the COBOL, PL/1, 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.

28.3.3 Creating COBOL stored procedures for debugging

Once the z/OS server and the workstation of a client tool is being used is set up, the COBOL stored procedure has to be created for debug:

The steps that must be completed include:

1. Create the DB2 COBOL stored procedure, including compile and linkedit JCL.
2. Create and register the procedure in the DB2 Catalog.
3. Set up a WLM AE with required data sets.
4. Debugging using the IBM Distributed Debugger and the DB2 Development Center:

a. Identify the workstation IP address, and update the z/OS COBOL PROCEDURE DDL RUNOPTS TEST parameter definition with this value.

b. Compile the COBOL program for parm TEST.

c. Start the IBM Distributed Debugger daemon.

d. Start the DB2 Development Center (DC) (optional):

- Alternatively, write a calling client program to call the stored procedure.

e. From Server View, select and run the **Stored Procedure.**

f. Select the **Distributed Debug** window, where the source listing is displayed.

g. Set a breakpoint.

h. Run the stored procedure to breakpoint, check variable values, and interactively debug.

5. Debugging using WSED
6. Debugging using TSO

Create the DB2 COBOL stored procedure

COBOL stored procedures use the DSNHICOB or comparable procedure. The changes to the default procedure to enable use by Debug Tool include:

- 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 and OS/390 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.
- 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 COBOL procedure is shown in Example 28-4.

Example 28-4 COBOL compile procedure example

```
//PH061S03 EXEC DSNHICOB,MEM=EMPDTLSC,
...
//  PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)',
//         TEST(ALL),MAP,OFFSET)
//COB.SYSPRINT DD DSN=SG247083.CBLSRC.LISTING(EMPDTLSC),
//            DISP=SHR
```

Create and register the procedure in the DB2 Catalog

When using either the Development Center or the WSED client tools, the RUN OPTIONS parameter must include the workstation IP address. The current workstation IP address is determined by issuing an IPCONFIG command from a DOS command prompt on the workstation, as shown in Example 28-5. Workstations that are processing on a multi-network node, such as a home network, can be specified. This was not possible when using the SQL Debugger as described in 28.2.6, “Debugging the EMPDTLSS SQL case study” on page 476.

Example 28-5 Determine workstation IP address

```
C:\WINNT>ipconfig
Windows 2000 IP Configuration
Ethernet adapter Ethernet Rear:
```

```
Connection-specific DNS Suffix  . :
IP Address. . . . . . . . . . . . : 9.112.68.25
Subnet Mask . . . . . . . . . . . : 255.255.255.0
Default Gateway . . . . . . . . . :
```

The CREATE PROCEDURE registration statement needs to specify the RUN OPTIONS parm which includes:

- Parm TEST
- Sub-parm IP address.

We used the default listening port of 8000, 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 28-6.

Example 28-6 CREATE PROCEDURE definition

```
CREATE PROCEDURE
DEVL7083.EMPDTLSC
...
LANGUAGE COBOL
WLM ENVIRONMENT 7083DDC1
RUN OPTIONS 'TEST(,,,VADTCPIP&9.112.68.25%8000:*)'
```

A DB2 ALTER command can be used to change the IP address for a previously created COBOL stored procedure. See Example 28-7.

Example 28-7 Altering the IP address

```
ALTER PROCEDURE DEVL7083.EMPDTLSC RUN OPTIONS 'TEST(,,,VADTCPIP&9.112.68.25%8000:*)'
```

Set up a WLM AE with required data sets

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

Additionally, if the z/OS TCP/IP procedure is not using the default //SYSTCPD data set, which is TCPIP.TCPIP.DATA, then a //SYSTCPD DD needs to be added to the WLM proc. Locate the //SYSTCPD data set in the TCP/IP procedure, and add the same DD statement and data set to the WLM procedure. The TCP/IP data set used in our system was:

```
//SYSTCPD DD DSN=TCP.SC63.TCPPARMS(TCPDATA),DISP=SHR
```

Using the IBM Distributed Debugger and the DB2 Development Center

Now that the environment is set up, we performed the following steps and started debugging:

1. First, we identified the workstation IP address, and altered the z/OS COBOL PROCEDURE Run Options with this value. See Example 28-7.

   Debug Tool on z/OS communicates with the workstation IBM Distributed Debugger client using the workstation IP address. The current IP address can be determined by entering `IPCONFIG` from a DOS command prompt.

   Once the current IP address is identified, it needs to be updated in the PROCEDURE definition for the stored procedure being debugged. This can be done during initial creation of the DDL for the stored procedure in the run options parameter, or later using

the ALTER command. If a multi-network environment such as a home network is being used, the PPP IP address is the one that needs to be specified in the RUN OPTIONS parameter.

2. Compile the COBOL program for parm TEST:

   Follow steps described in “Create the DB2 COBOL stored procedure” on page 482.

3. Start the IBM Distributed Debugger daemon:

   From a DOS command prompt on the workstation enter:

   ```
   idebug -qdaemon -quiport=8000
   ```

   The `-quiport` value of 8000 matches the value specified in the Run Options specification. The daemon appears with the following window on your workstation until the COBOL stored procedure that we are debugging is started. See Figure 28-13.

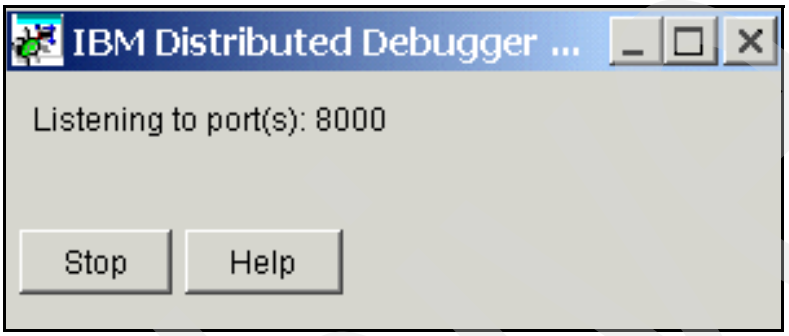


Figure 28-13 IBM Distributed Debugger daemon listening on port 8000

4. Start the DB2 Development Center (DC)

   We started the DB2 Development Center with **Start -> Programs -> IBM DB2 -> Development Tools -> Development Center**.

   Since we are just running an existing stored procedure, no project has to be selected or created. Once Development Center is started, select **Server View**. From the Database Connections folder, right-click **Add Connection** and select the **Alias** for our DB2 server, DB2G as shown in Figure 28-14. See Table 29-2 on page 505 for directions on how to define a DB2 server alias.

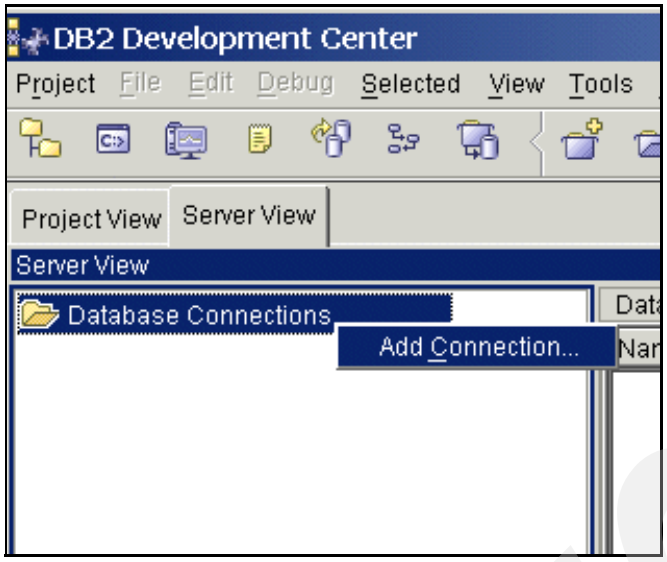


Figure 28-14 Server View add database connection

Since we want to limit the objects returned to Development Center, we set a filter for Schema and Language. This is done, by selecting the **Stored Procedures** folder and right-clicking **Filter** as shown in Figure 28-15.

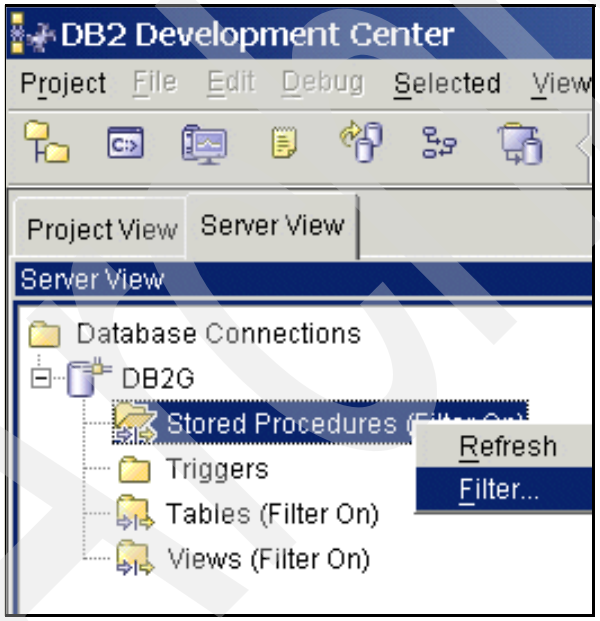


Figure 28-15 Set filter

In the Filter Stored Procedures window, enter the schema and click **Starts with the characters -> DEVL7083.** Since we do not want SQL, C, or Java stored procedures, we selected **Language** of Other. Click **OK** to request filtered objects shown in Figure 28-16.

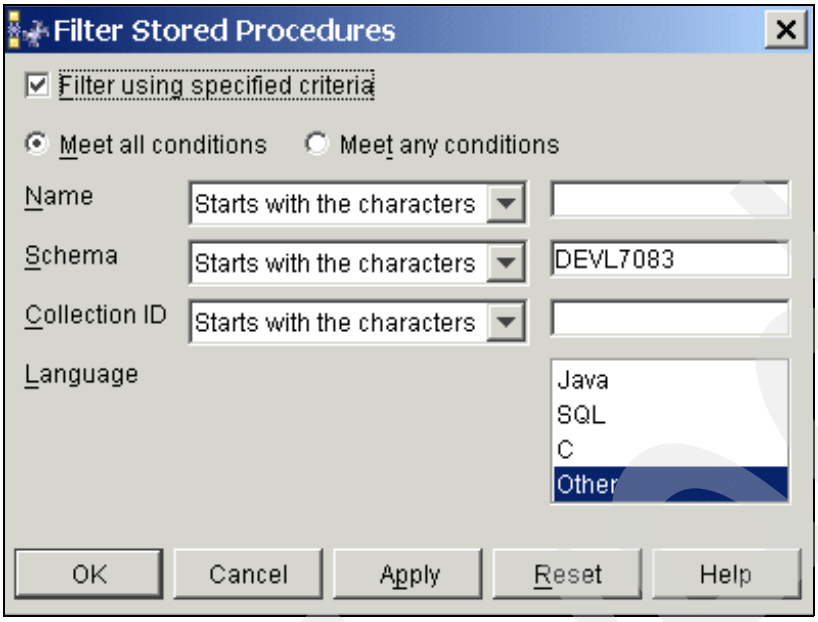


Figure 28-16 Filter Stored Procedures by Schema and Language

5. From Server View, select and run the stored procedure:

   We select **EMPDTLSC**, which is the COBOL stored procedure we want to debug. Select **Run** as shown in Figure 28-17.

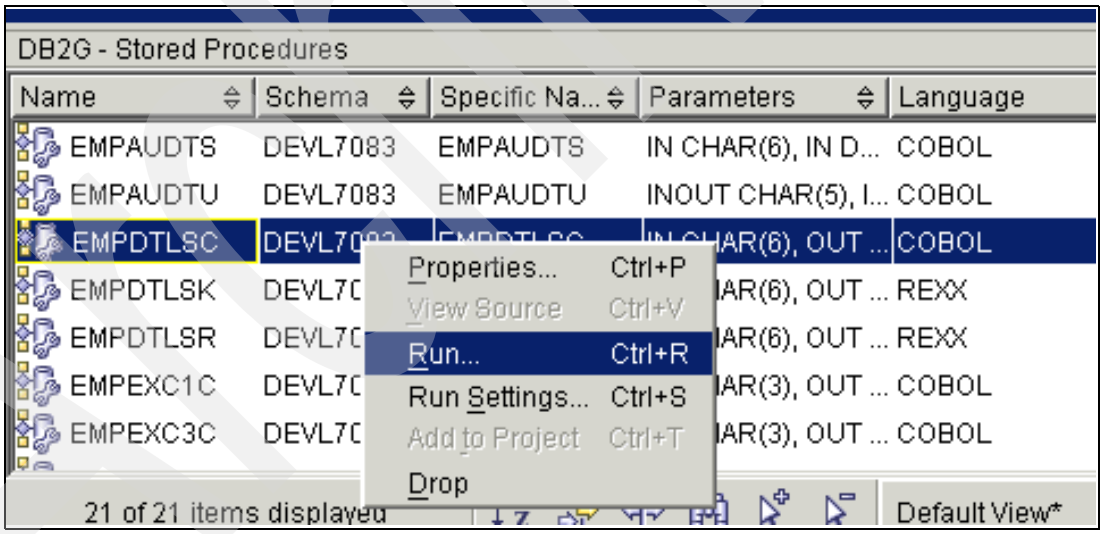


Figure 28-17 Run EMPTDLSC COBOL stored procedure

EMPTDTLSC is started with an input parameter. A Specify Parameter Values window appears where we entered `PEMPNO` of `000010` as shown in Figure 28-18. Clicking **OK** starts the execution of the stored procedure.

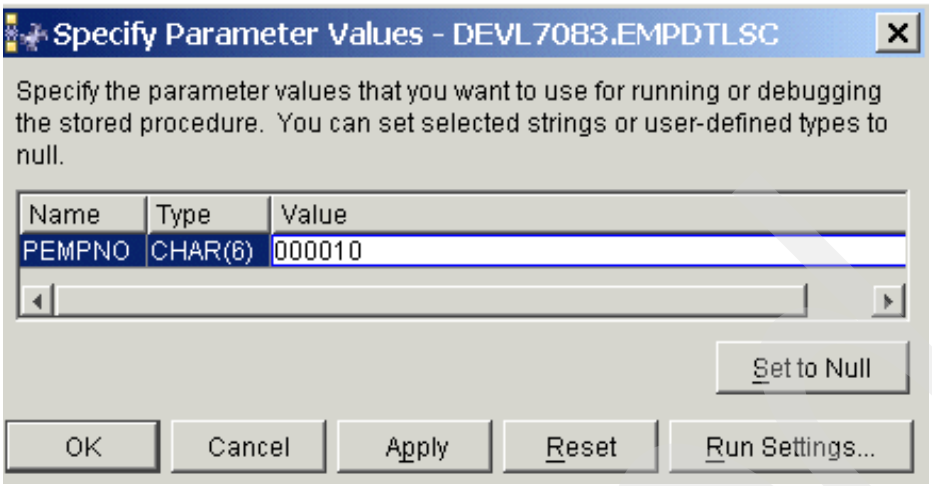


Figure 28-18 Input parameter window

6. Select the **Distributed Debug** window where the source listing is displayed.

   The IBM Distributed Debugger loads the source listing from the COB.SYSPRINT data set we defined in our Compile procedure. The listing file is displayed in the right hand window of Figure 28-19.

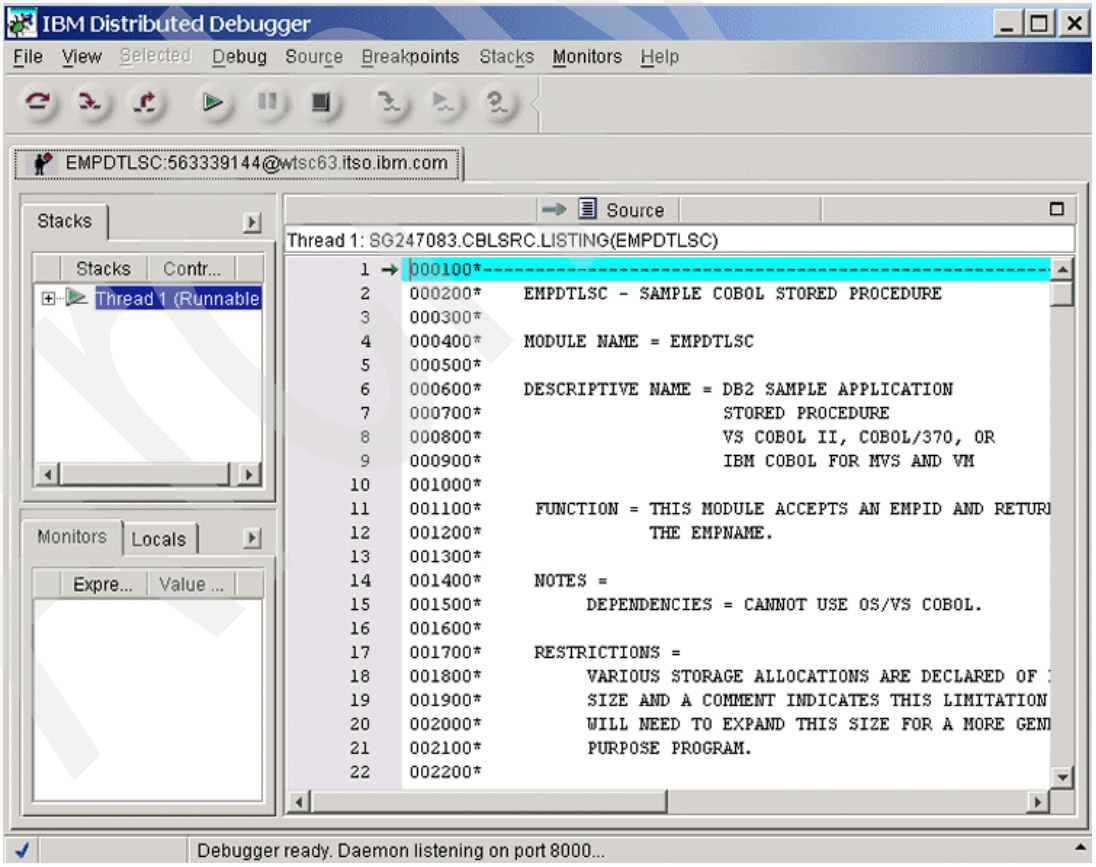


Figure 28-19 COBOL listing displayed in the IBM Distributed Debugger

7. Set a breakpoint.

To set a breakpoint, locate line **315**, we placed our cursor in the prefix area on the left, and right-clicked **Set Breakpoint** as shown in Figure 28-20.

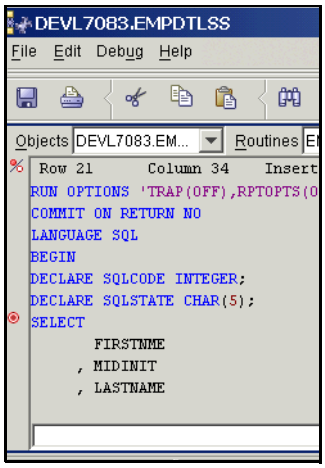


Figure 28-20 Set a breakpoint at line 315

8. Run the stored procedure to the breakpoint.

   Select the **Run** icon on the top tool bar, which executes the EMPDTLSC stored procedure to the breakpoint we set on line 315 as shown in Figure 28-21.

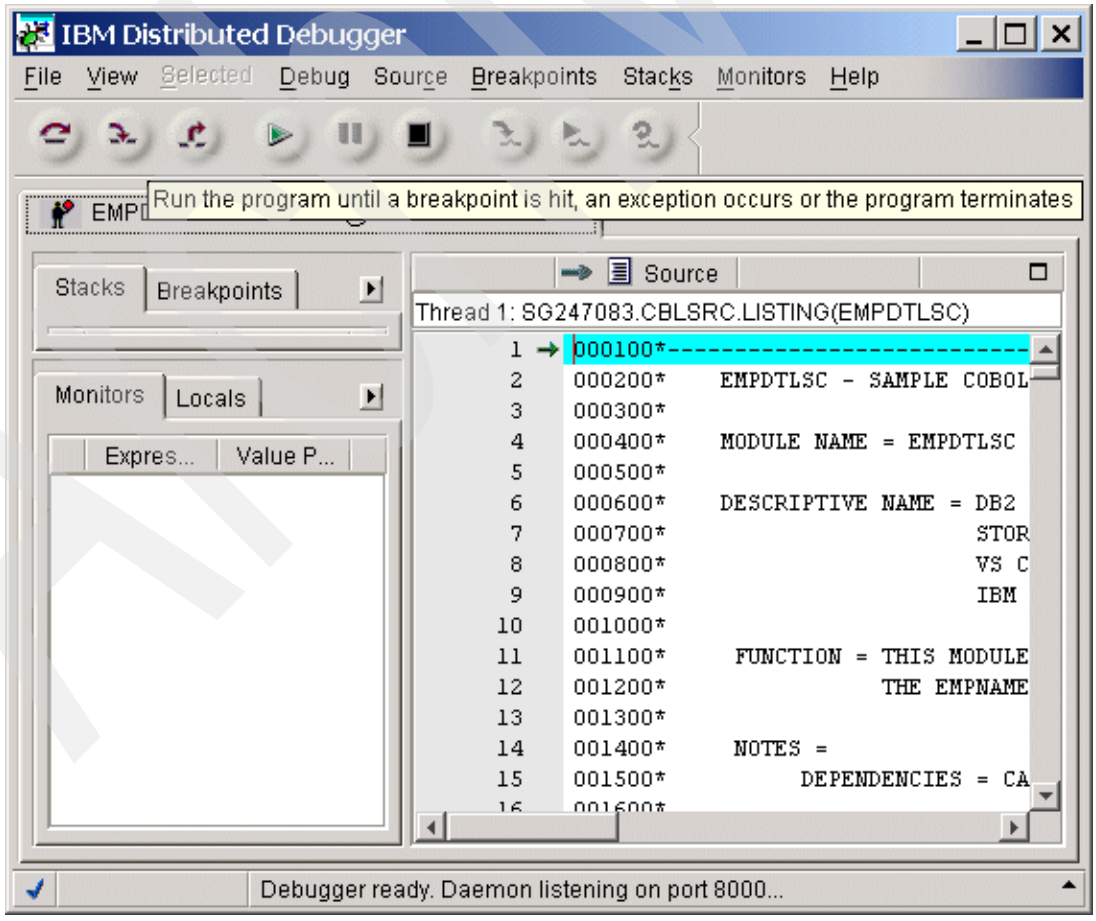


Figure 28-21 Select the > icon on the tool bar to run to the breakpoint on line 315

9. Check variable values and interactively debug.

   The IBM Distributed Debugger stops on line 315. Selecting the **Locals** tab displays variables that have been defined and set in our stored procedure through line 315 in our code as shown in Figure 28-22.

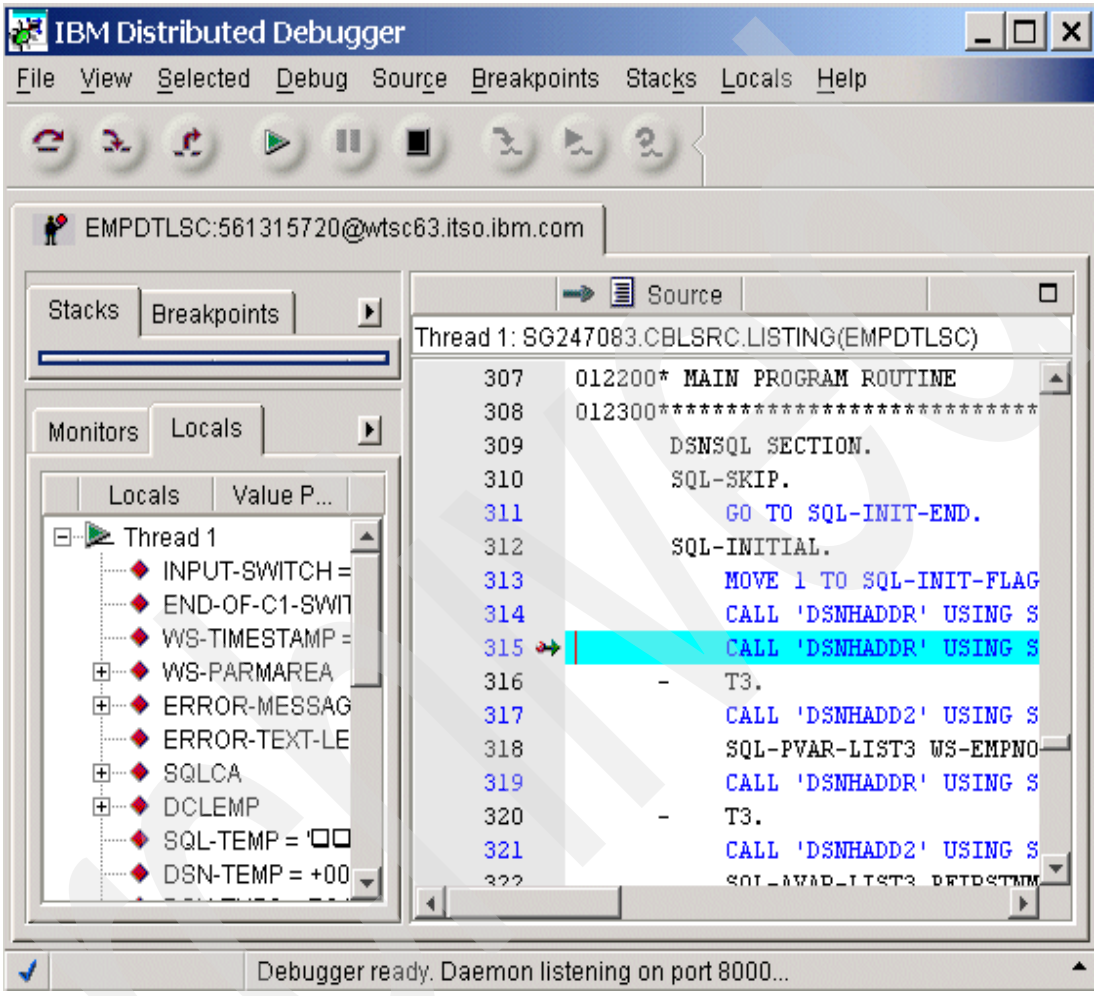


Figure 28-22 IBM Distributed Debugger - Local variables displayed

We get you started for debugging your COBOL stored procedure using the IBM Distributed Debugger on your workstation interfacing to Debug Tool on z/OS. Detailed information on using the IBM Distributed Debugger can be found in both the online help in the IBM Distributed Debugger product and online at the URL:
http://www.ibm.com/software/awdtools/debugger/

Debugging using WSED

This section describes how to debug the same EMPDTLSC COBOL stored procedure using the WebSphere Studio Enterprise Developer (WSED) client Debugger. Debug Tool must be set up as described in 28.2.2, “Prerequisites and set up” on page 467, and steps 1 to 3 in 28.2.3, “Creating SQL stored procedures for debugging” on page 468 must be completed.

First we start WSED with **Start -> Program Files -> IBM WebSphere Studio ->WebSphere Studio Enterprise Developer**. Next, we select the **Debug Perspective**. From the Menu bar, select **Window ->Open Perspective -> Other ->Debug**.

The default Debug daemon port is 8001. This port is specified directly after the IP address in the RUN OPTIONS, TEST sub-parameter. This is different than the IBM Distributed Debugger, which used port 8000. We used the default daemon in our example, which meant we had to change the RUN OPTIONS for our stored procedure to match. Specifically, once we knew our workstation IP address, we issued the ALTER DDL statement as shown in Example 28-8.

Example 28-8 ALTER PROCEDURE for TCP/IP address

```
ALTER PROCEDURE DEVL7083.EMPDTLSC RUN OPTIONS 'TEST(,,,VADTCPIP&9.112.68.25%8001:*)'
```

The default Debug daemon port can be changed to a different value by updating the Debug Preferences under Debug Daemon as in Figure 28-23.

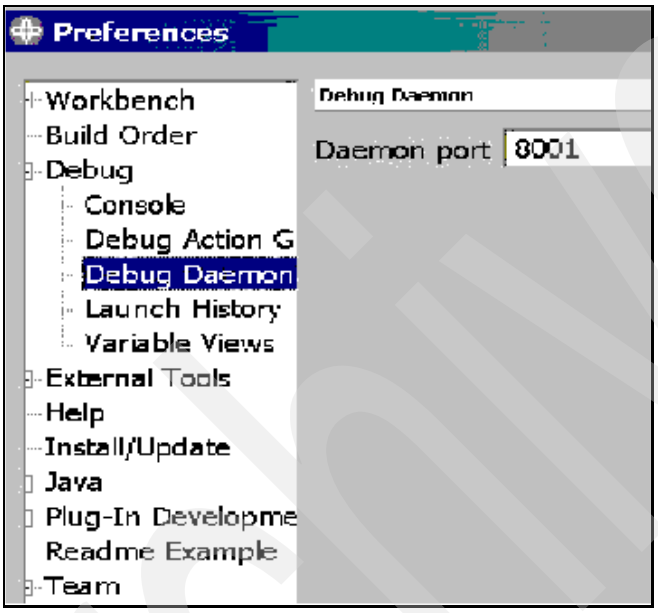


Figure 28-23 Debug Daemon port default

From the Debug Perspective, start the Debug daemon by clicking the **Debug ICON** on the tool bar as shown in Figure 28-24.

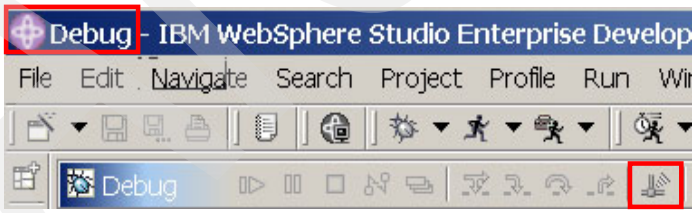


Figure 28-24 Start the WSED debug daemon

Running our COBOL stored procedure is performed from the Data Perspective, DB Servers View. There are multiple ways to open the Data Perspective view. We opened the Data Perspective view from the left-side toolbar, selecting the **table icon with a +**. Clicking this icon expands the Perspective selection list. We clicked the **Data Perspective**, which opens the window in Figure 28-25.

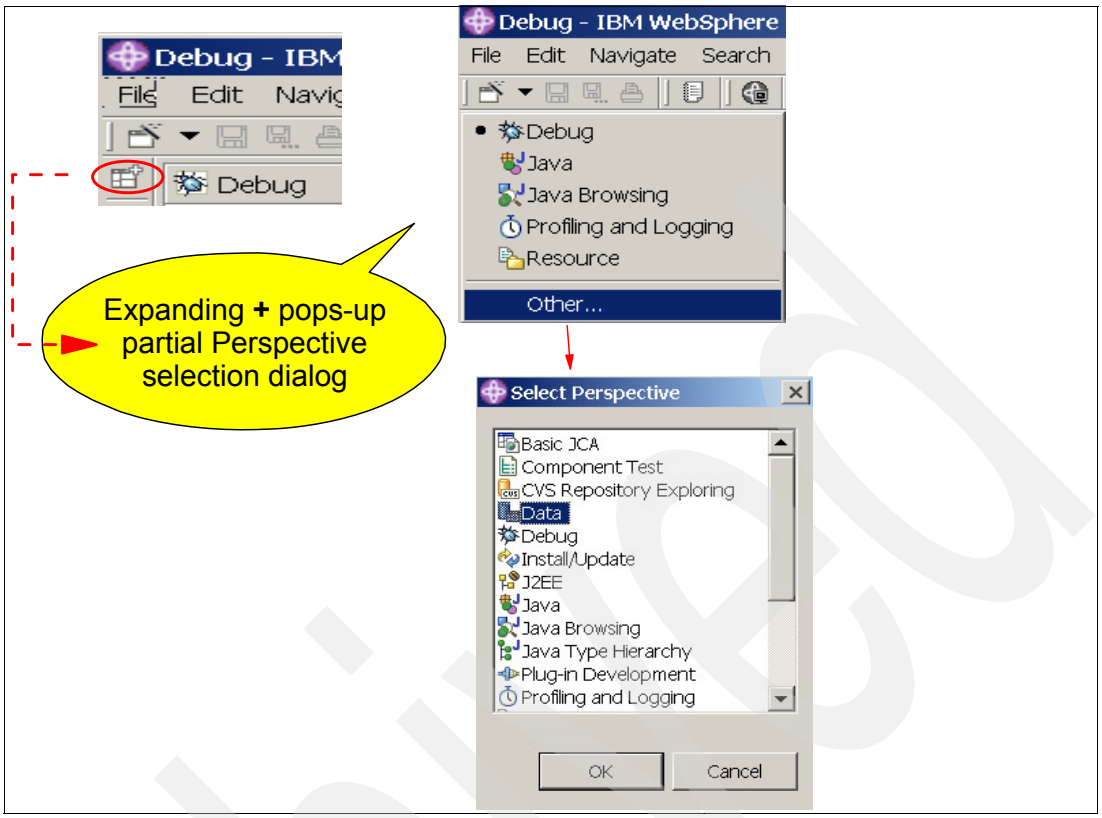


Figure 28-25 Open the WSED Data Perspective

From the Data Perspective, in the DB Servers window in the lower left hand side, we positioned our mouse in the white space in this window and right-clicked **New Connection** as shown in Figure 28-26.

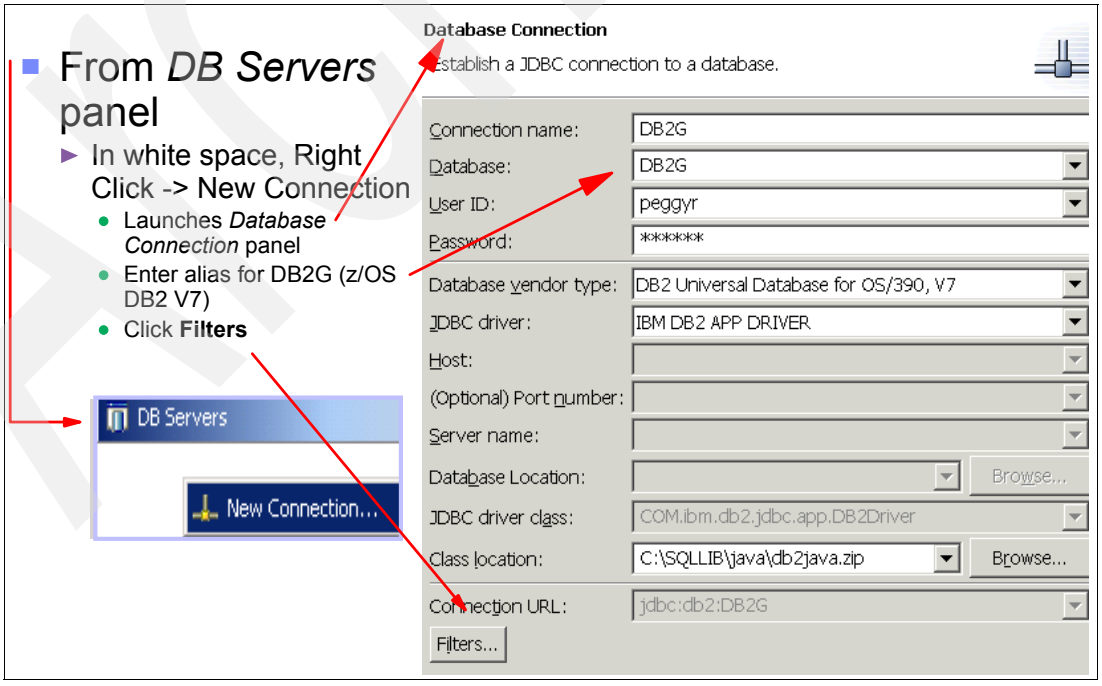


Figure 28-26 Define a new connection

This opens the Database Connection window used to establish a JDBC connection to a database. We entered the following information:

1. DB2G for Connection name
2. DB2G for the Database name. This value is what has previously been configured with the DB2 Configuration Assistant for DB2 servers accessibly from WSAD or other DB2 client software. DB2G is a DB2 for OS/390 V7 Server.
3. Specify the DB2 user ID and password for this DB2 server.
4. Select the database vendor type of DB2 Universal Database for OS/390 V7.
5. Select the IBM DB2 APP DRIVER.
6. Include the class location for the db2java.zip file.
7. Click **Filters** to filter the objects that will be returned.

We entered our filtering information in the window shown in Figure 28-27.

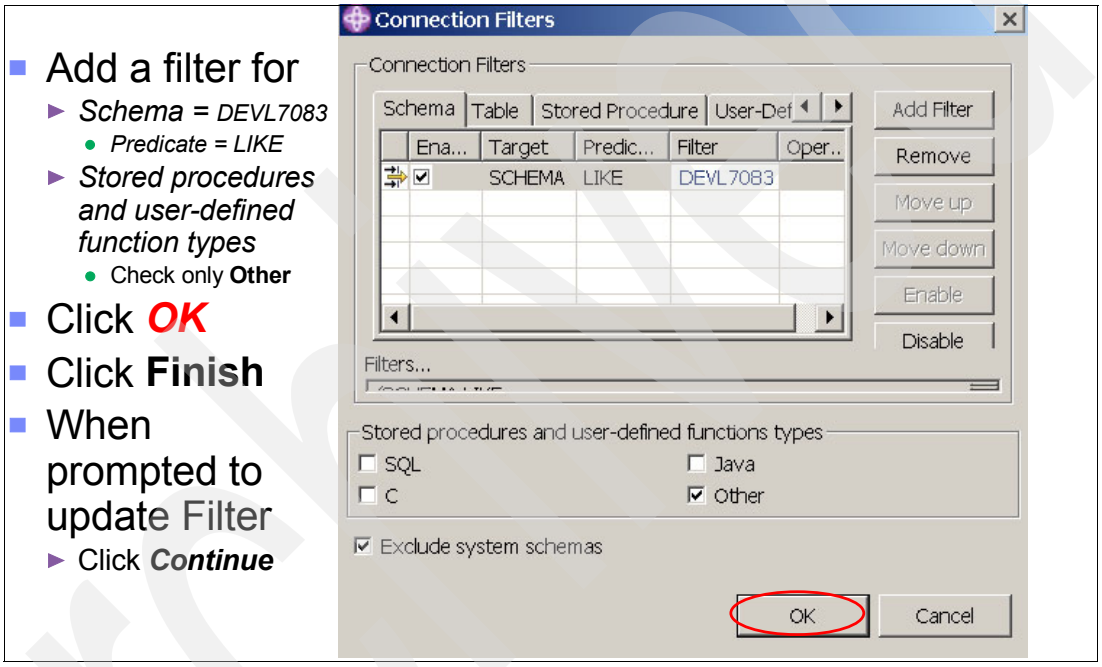


Figure 28-27 Filter objects returned

Before we run the stored procedure in debug mode, we need to compile the stored procedure with the test option, and set the IP address on the procedure DDL using the steps from 28.3.3, “Creating COBOL stored procedures for debugging” on page 481. We launched the stored procedure from the Data Perspective, DB Servers view. We selected the **EMPDTLSC stored procedure** and right-clicked **Run** as shown in Figure 28-28.

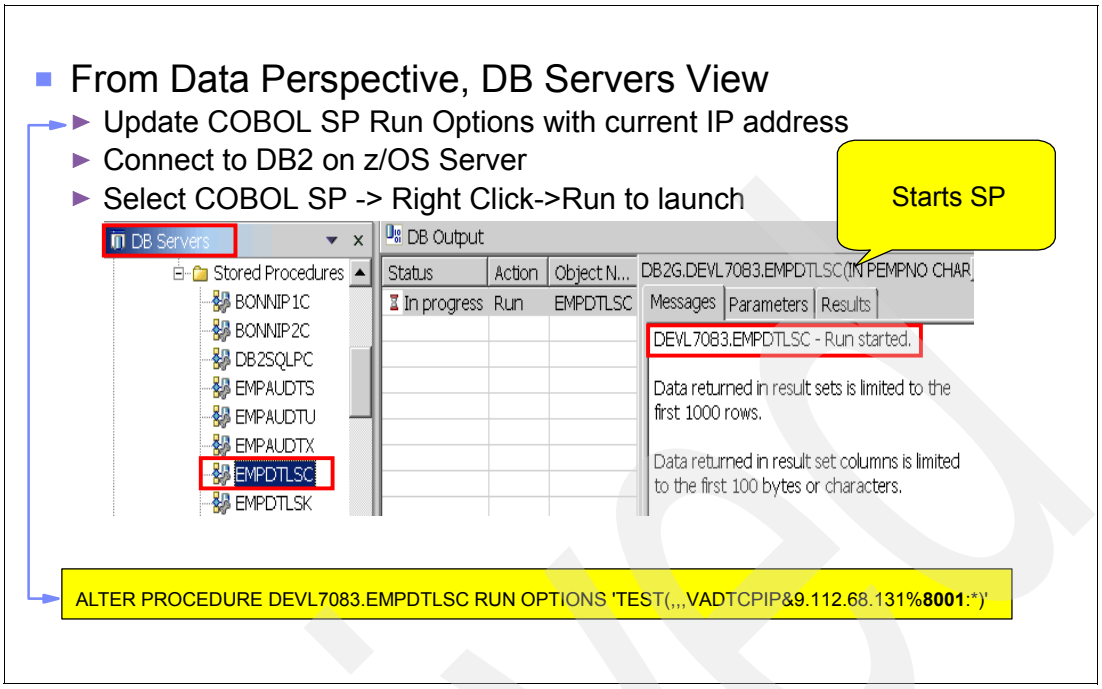


Figure 28-28 Start COBOL stored procedure

Now, we switch to the WSED Debug Perspective where the source listing is displayed, where we start interactive debugging; see Figure 28-29. The WSED online help provides additional information on using the Debug Perspective, setting breakpoints, monitoring variables, etc.

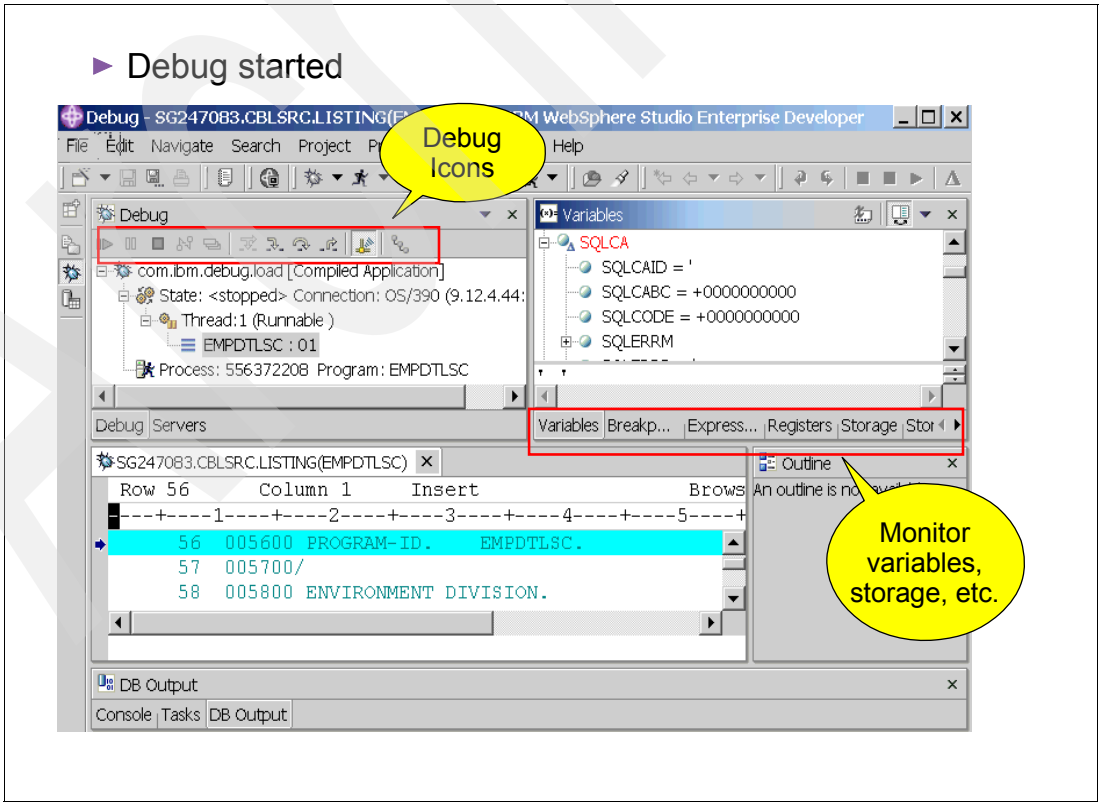


Figure 28-29 WSED Debugger launched

For more information on WSED refer to *Exploring WebSphere Studio Enterprise Developer 5.1.2,* SG24-6483. In this recent redbook you can see that this new version of WSED is a fully functional z/OS-based DB2 Stored Procedure builder on the workstation.

28.4 Debugging options for DB2 Java SPs on z/OS

The IBM Distributed Debugger provides a GUI interface for debugging Java stored procedures on DB2 UDB for Linux, UNIX, and Windows. This option is not available for debugging DB2 interpreted Java stored procedures on DB2 for z/OS. There are three alternatives that you can use to debug DB2 Java stored procedures on z/OS and OS/390:

- Create comparable DB2 for z/OS DDL and SQL on a DB2 for Windows or UNIX; then use the IBM Distributed Debugger as described in the DB2 Stored Procedure Builder (V7.2 client) or Development Center (V8.1 client) online help, under *Remotely debugging stored procedures.* 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 "Using DC to copy from one server and paste/build on another server" on page 541. This requires the same z/OS set up for Java stored procedures as described in section "z/OS setup" on page 506.
- 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. This requirement is true when using either the SDK 1.3.1 or the SDK 1.4.1.

  Alternatively, you may want to isolate the Java sysprint lines more granularity 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 proc JAVAENV statement.
- Convert your Java stored procedure to a Java application, and debug using one of the WebSphere Studio editions:
  - WebSphere Studio Site Developer (WSSD)
  - WebSphere Studio Application Developer (WSAD)
  - WebSphere Studio Application Developer Integrated Edition (WSADIE)
  - WebSphere Studio Enterprise Developer (WSED)

  See Chapter 30., "Using WSAD to debug Java stored procedures converted to Java applications" on page 553.

28.5 Debugging Java SPs on Windows, AIX, and Sun

While the main focus of this redbook is stored procedures on DB2 for z/OS and OS/390, this next section describes debugging Java stored procedures using the Development Center and the Distributed Debugger on Windows, AIX, and Sun. When this redbook was written, this is the only IBM environment for debugging Java stored procedures. Some customers prototype their application development on DB2 for Windows, AIX, or Sun before porting to DB2 for z/OS. When this environment is possible, then the following debugging option can be used.

We used DC to copy the EmpDtlsJ stored procedure from our DB8A on z/OS then pasted this stored procedure to our DB2 UDB SAMPLE database on Windows. 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. Workstation setup
2. DB2 Server setup
3. Start Development Center and create database connections:
   - Create a database connection to DB8A, our DB2 for z/OS V8

     Add DEVL7083.EMPDTLSJ to project.
   - Create a database connection to SAMPLE, our DB2 UDB on Windows.
4. Using Development Center, copy EmpDtlsJ from DB8A and paste to SAMPLE.
5. From DC Editor View, change table DSN8810.EMP to EMPLOYEE.
6. Run the stored procedure in debug mode.

28.5.1 Workstation set up

Set up the client for debugging by installing the IBM Distributed Debugger:

- IBM Distributed Debugger installed as described in “Installing the Distributed Debugger” on page 480
- DB2 Development Center installed as described in “Install Development Center” on page 504

28.5.2 DB2 server set up

Enable the DB2 server to use the IBM Distributed Debugger to debug Java stored procedures:

- On the DB2 server, enter the following command from a DOS command prompt:

```
DB2SET DB2ROUTINE_DEBUG=ON
```

- Disconnect all DB2 applications from the DB2 server, and restart the server.

28.5.3 Start Development Center and create database connections

We started Development Center and opened our project DEVL7083 that was created in 29.4.1, “Starting the Development Center for the first time” on page 528. If not already added, add database connections for DB8A and SAMPLE.

From Server View, we selected the DB8A database, filtered the stored procedures in Schema by selecting **Equal to the names** and entering `DEVL7083` and for Language selected **Java,** as shown in Figure 28-30.

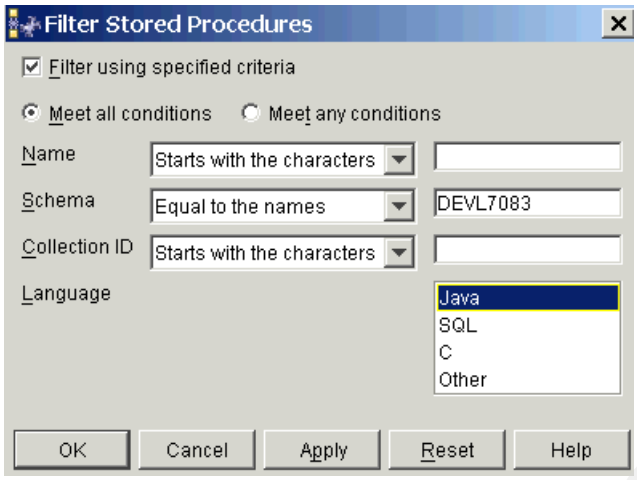


Figure 28-30 Filter stored procedures

We located the DEVL7083.EMPDTLSJ stored procedure in the list that was returned to Server View and right-clicked **Add** to Projec as shown in Figure 28-31.

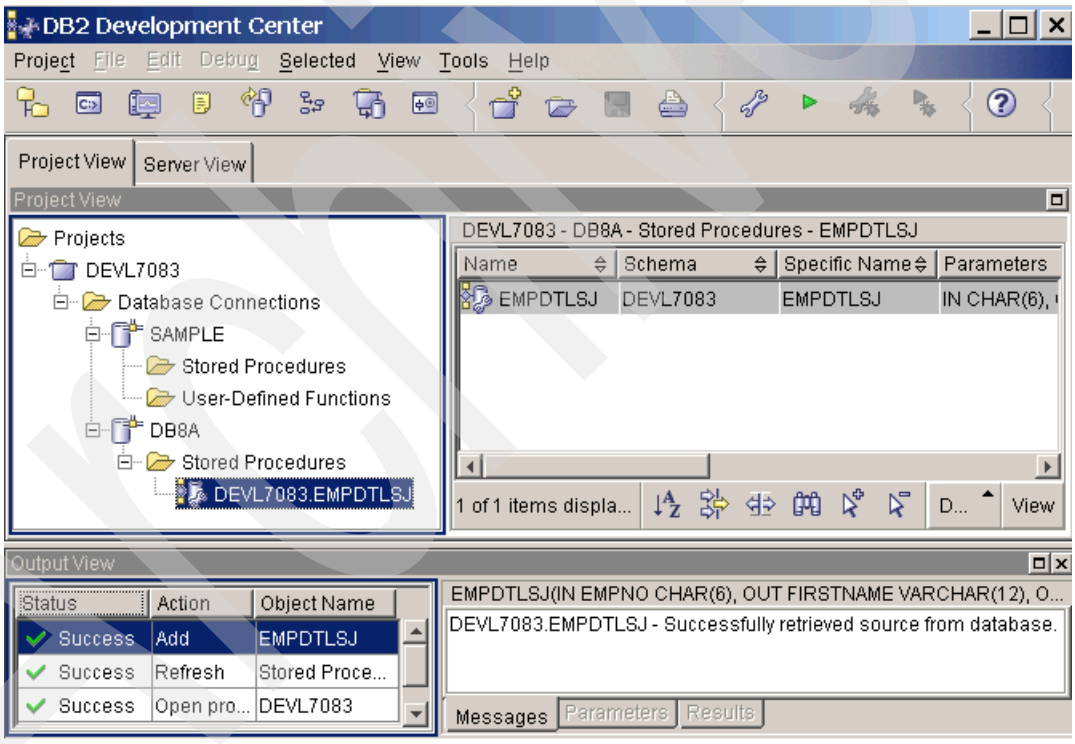


Figure 28-31 Add stored procedure from server to project

28.5.4 Using Development Center copy EmpDtlsJ from DB8A and paste to SAMPLE

The Development Center (DC) copies a specific stored procedure from one server and pastes to another server and can process between like and unlike servers. That is, we can copy a stored procedure created on z/OS DB2 and paste, modify as necessary, and build on a Windows server. This is the scenario we are performing in this example.

In Project View, select **DEVL7083.EMPDTLSJ** from our DB8A Stored Procedures folder. Right-click **Copy** then select the **Windows SAMPLE server** and **Stored Procedures** folder and right-click **Paste** as shown in Figure 28-32. See 29.5.4, “Using DC to copy from one server and paste/build on another server” on page 541.

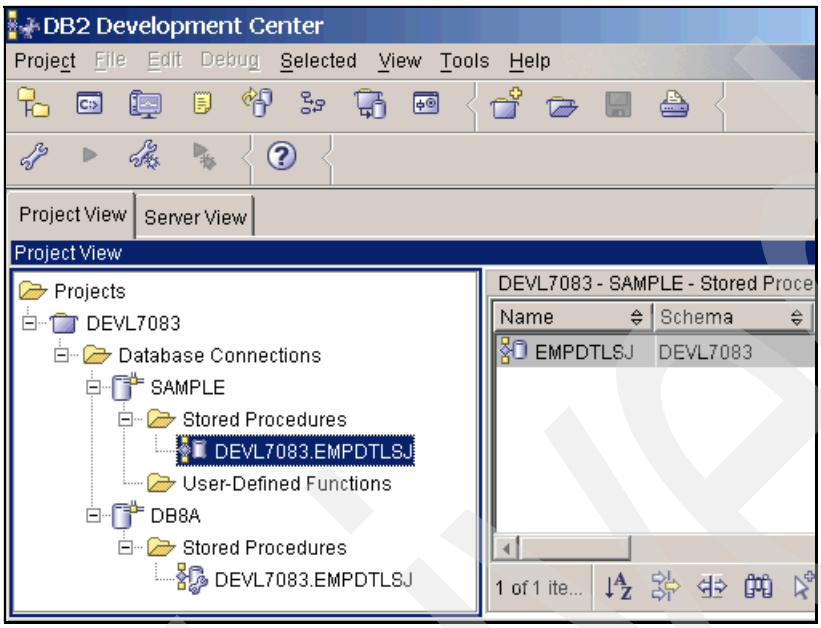


Figure 28-32 Copy and paste stored procedure from z/OS to Windows

28.5.5 From DC Editor View, change table DSN8810.EMP to EMPLOYEE

Then we changed the EMP table to EMPLOYEE table. We did this by locating on the SAMPLE server, and double-clicking the **DEVL7083.EMPDTLSJ** stored procedure, which opened this stored procedure in Editor View. While in Editor View, we located the line:

```
sql = "SELECT * FROM DSN8810.EMP WHERE EMPNO = '"+ empno + "'";
```

and changed it to the following:

```
sql = "SELECT * FROM EMPLOYEE WHERE EMPNO = '"+ empno + "'";
```

Finally, we saved the stored procedure and built the stored procedure for debug. In Editor View, we clicked the **wrench and bug** icon from the toolbar. When the stored procedure is successfully built in debug mode, the message in Example 28-9 is returned to the DC Output View Messages window.

Example 28-9 Build Java stored procedure in debug mode on Windows

```
DEVL7083.EMPDTLSJ - Source updated.
DEVL7083.EMPDTLSJ - Build for debug successful.
```

28.5.6 Run the stored procedure in debug mode

The final step for our Java stored procedure debugging example on Windows is to start the debugging session:

- Start a DOS command window. Start the Debug daemon by issuing the command in Example 28-10, where 8000 is the listening port you want to use. Alternatively, select any open port on your Windows workstation that is not in use.

Example 28-10 Start IBM Distributed Debugger, debug daemon listening on port 8000

```
idebug -qdaemon -quiport=8000
```

- From Development Center Project view, select the **SAMPLES -> Stored Procedure folder-> DEVL7083.EMPDTLSJ Java stored procedure** and right-click **Debug**.

  This starts running the Java stored procedure in debug mode as shown in Figure 28-33. Use the IBM Distributed Debugger documentation, located at:

  `http://www.ibm.com/software/awdtools/debugger/`

  to debug this Java stored procedure.

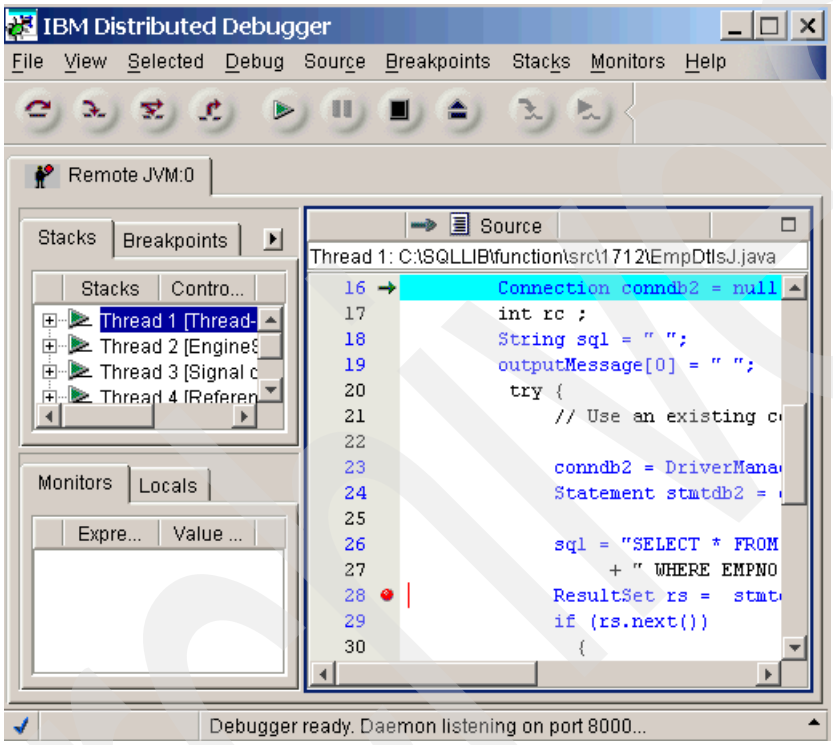


Figure 28-33 Distributed Debugger debugging Java stored procedure on Windows

29

The DB2 Development Center

The DB2 Development Center (DC), included in the V8.1 UDB Application Development Client (ADC) component is the follow-on product to the DB2 Stored Procedure Builder (SPB) in the DB2 V7.2 UDB Application Development Client.

Development Center supports the entire family of DB2 servers using the DRDA architecture. It communicates with the DB2 UDB V8.1 distributed servers (Linux, UNIX, and Windows), and with DB2 UDB for OS/390 V6 and V7 and DB2 UDB for z/OS V8, as well as currently supported DB2 UDB releases on iSeries.

Development Center supports creating SQL stored procedures on all supported versions of DB2 for OS/390 and z/OS (currently V6, V7, and V8). Java stored procedures can be created with Development Center on DB2 V7 and V8. Many additional enhancements beyond creating stored procedures are included in Development Center and are described in this chapter. The Development Center Online Help includes additional information to complement this chapter.

The OS/390 and z/OS set up described in this chapter for creating SQL and Java stored procedures for Development Center also applies when using the Data Perspective in the WebSphere Studio editions, WSAD, WSADIE, WSED for creating SQL or Java stored procedures on OS/390 and z/OS. The setup also applies when using .NET to create SQL stored procedures on OS/390 and z/OS.

We created SQL and Java stored procedures on two DB2 subsystems: DB2G, a DB2 V7 server; and DB8A, a DB2 V8 server. In addition to the new stored procedures, we imported the case study EMPDTLSS SQL language and EMPDTLSJ Java stored procedures.

This chapter contains the following:

- Development Center start up
- Prerequisites and setup steps
- A guided tour through Development Center
- Getting started with Development Center
- Advanced Development Center topics
- Future Development Center enhancements

29.1 Development Center start up

When you start Development Center, the first thing you do is open a project to create or update objects or select the **Server View** to view or run server objects. You connect to a DB2 server using an alias that has been defined to the Configuration Assistant (CA). Next, the DB2 catalog is read to return a list of objects, which can be filtered from the DB2 server. When in Project View, you can create new or update existing SQL or Java stored procedures. Development Center processing is described in Figure 29-1.

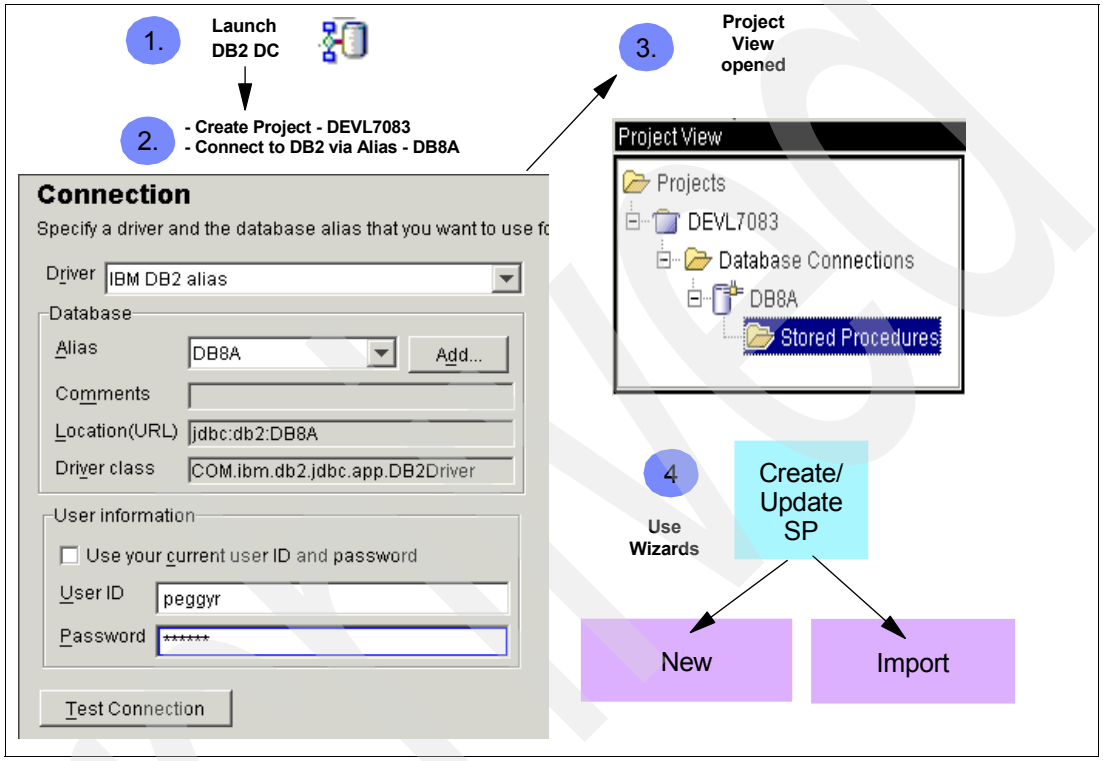


Figure 29-1 Starting Development Center

Development Center creates SQL and Java stored procedures on OS/390 and z/OS using multiple DB2-supplied stored procedures. The main DB2-supplied stored procedures that perform the processing on z/OS for Development Center are:

- DSNTPSMP for SQL stored procedures
- DSNTJSPP and DSNTBIND for Java stored procedures

SQL stored procedures built with DSNTPSMP

DSNTPSMP is a REXX DB2-supplied stored procedure that builds SQL stored procedures on DB2 for OS/390 V6 and V7, and DB2 for z/OS V8. Multiple *build* functions are supported by this stored procedure. Table 29-1 shows the functions that are supported.

Table 29-1 DSNTPSMP supported functions

Type	Name	Function
Basic	BUILD	Creates a new SQL procedure only
	ALTER	Updates (most) stored procedure options
	REBIND	Perform Bind Package again (not REBIND command)

Type	Name	Function
	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 Development Center to create SQL stored procedures on DB2 V6, V7 and V8:

1. Start DC and connect to a DB2 server. Then use the wizards to create a new SQL stored procedure where you can use the SQL Assist to help develop your SQL statements. Alternatively, you can import the source of an existing SQL stored procedure into DC, or open up a DC window and start coding your SQL stored procedure directly. You can edit the source in DC, and add any valid SQL syntax as described by the ANSI standard for SQL stored procedures.
2. Once you are finished with your code, you click the **Build** (wrench) icon, which starts building the SQL stored procedure into your DB2 subsystem. The steps start out on the workstation.
3. Now DSNTPSMP is called.
4. 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 29-2 describes how the Development Center creates SQL stored procedures on z/OS.

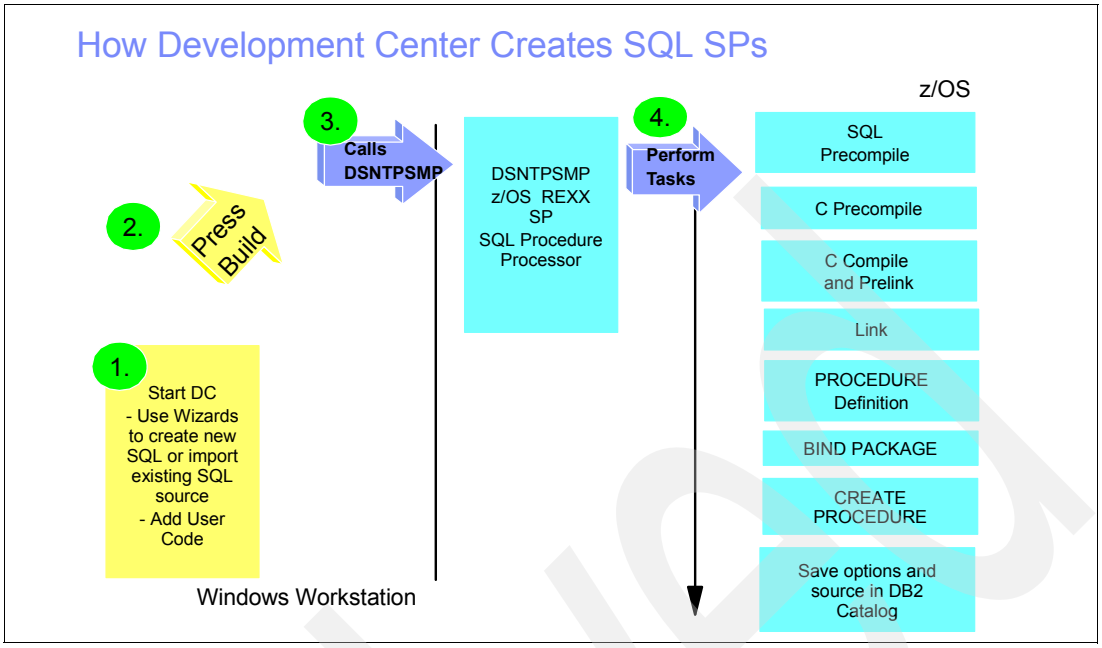


Figure 29-2 How Development Center creates SQL stored procedures

Java stored procedures built with DSNTJSPP and DSNTBIND

Two DB2-supplied stored procedures may be used to create Java stored procedures on z/OS. DSNTJSPP is a C stored procedure used in the build processing for both JDBC and SQLJ stored procedures. DSNTBIND is a REXX stored procedure used for doing the DBRM bind for SQLJ stored procedures. Neither DSNTJSPP nor DSNTBIND are supported through external interfaces. They work in conjunction with the Development Center in the DB2 UDB V8.1 ADC, and the Stored Procedure Builder in the DB2 UDB V7.2 ADC. See Chapter 25, “DB2-supplied stored procedures” on page 403 for an overview of DB2 provided stored procedures.

The following steps are performed by Development Center to create interpreted Java stored procedures on DB2 V7 and V8:

1. Start DC and connect to a DB2 server. Then use the wizards to create a new Java stored procedure where you can use the SQL Assist to help develop your SQL statements. Alternatively, you can import the source of an existing Java stored procedure into DC. You can edit the source in DC, and add any valid Java syntax.
2. Once you are finished with your code, you click the **Build** (wrench) icon, which starts building the Java stored procedure.
3. The steps start out on the workstation. Initially, the **`sqlj`** command is invoked if SQLJ is being used. Next, **`javac`** is invoked. Finally, the files are “jared” together.
4. Now, DSNTJSPP is called and processing continues on the host.
5. DSNTJSPP performs the following:
   a. The UNIX System Services (USS) environment is set up by reading the DSNTJSPP.properties file. This file is used in lieu of a .profile, which is required for manual Java stored procedure setup on z/OS.
   b. Next, the properties file db2sqljjdbc.properties is read, which defines the DBRMLIB to be used, and optionally sets additional environment variables such as a trace file.
   c. When SQLJ is being used, the **`db2profc`** command is invoked, which customizes the SQLJ profiles for DB2. Profiles are vendor-independent; therefore, they must be

customized. The customization process essentially transforms the generic SQLJ profiles into DB2-specific DBRMs, which can then be bound to DB2 in the normal way.

d. Then one of the following DB2-supplied stored procedures, SQLJ.INSTALL_JAR or SQLJ.REPLACE_JAR is called. These stored procedures save the Java *.class, *.ctx (SQLJ only) and *.ser (SQLJ only) as a BLOB in the DB2 catalog table, SYSIBM.SYSJAROBJECTS. Saving the Java stored procedure in the DB2 catalog eliminates the need to update the CLASSPATH in the JAVAENV data statement of the WLM AE where this stored procedure will execute. See “With jars” on page 261 of the section starting with 17.5.3, “Preparing SQLJ stored procedures” on page 257.

e. If SQLJ is used, the package is bound using the DSNTBIND REXX stored procedure.

f. The build options, including build utility, compile options, and bind options are saved in the DB2 catalog table SYSIBM.SYSJAVAOPTS.

6. Control is returned to the workstation, which registers the stored procedure in the DB2 Catalog, and saves options that were specified.

Figure 29-3 describes how the Development Center creates Java stored procedures on z/OS.

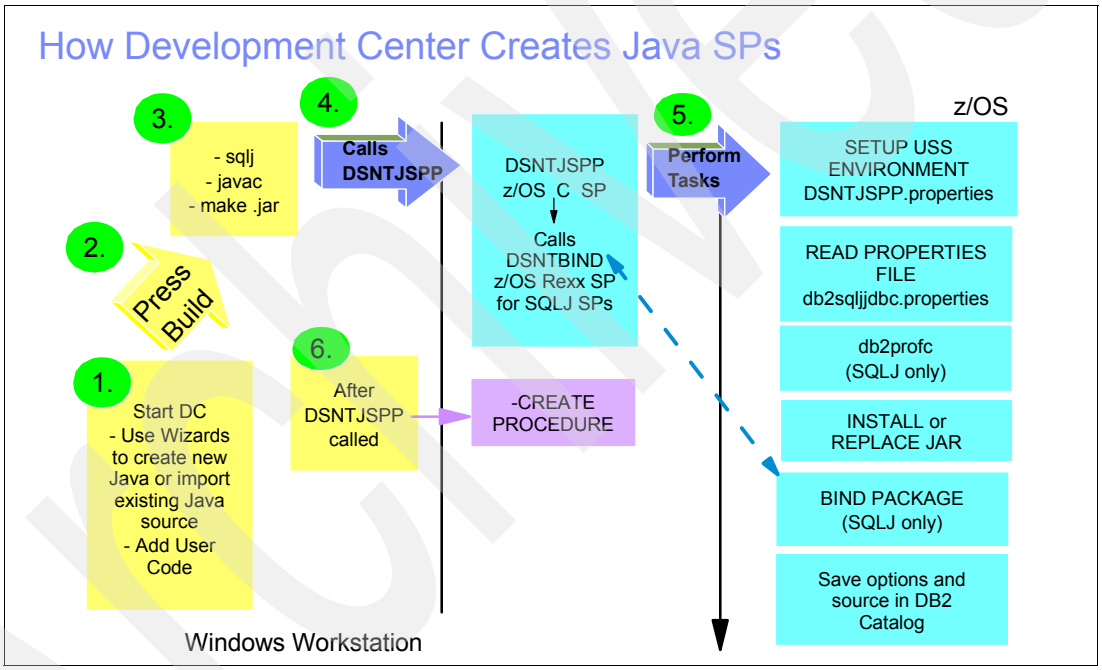


Figure 29-3 How Development Center creates Java stored procedures

29.2 Prerequisites and setup steps

In this section we describe the prerequisites and setup steps for both the client and the z/OS Server.

29.2.1 Client setup

The client setup is made up by the following steps:

1. Review prerequisites
2. Install Development Center
3. Install DB2 Connect

4. Bind the client to the server
5. Configure remote DB2 server access

Review prerequisites

The following lists the minimum prerequisites:

- DB2 V8.1 UDB Application Development Client (ADC)

  Development Center, the Configuration Assistant (CA) and the JDK 1.3 are all required, and all are included in the ADC. The Configuration Assistant is used to configure the connection to the remote DB2 server.

  The ADC is included in the following editions of DB2 UDB V8.1:

  - DB2 Universal Database Enterprise Server Edition
  - DB2 Universal Database Workgroup Server Unlimited Edition
  - DB2 Universal Database Workgroup Server Edition
  - DB2 Universal Database Personal Edition
  - DB2 Universal Database Universal Developer's Edition
  - DB2 Universal Database Personal Developer's Edition
  - DB2 UDB Express Edition
  - DB2 UDB Data Warehouse Edition

  The ADC is also available as a download from DB2 for z/OS Web site:

  `http://www.ibm.com/software/data/db2/os390/spb/client.html`

- DB2 V8.1 Connect, which can be an enterprise server edition installed on a server or a personal edition installed on the client workstation (z/OS requirement):

  A restricted use license of DB2 Connect for development purposes can be obtained in the z/OS Management Client Package. See the section on setup for “Workstation” on page 480 to obtain the FMIDs for the Management Client Package.

Optionally, if you want to prototype your SQL or Java Stored procedures for DB2 on your client workstation, you will need to install the following products on your workstation:

- DB2 V8.1 UDB (any edition)
- C compiler (MicroSoft C++ V6 is what we used) - required only for SQL stored procedures
- JDK 1.3, included with the DB2 V8.1 ADC
- We recommend using the most recent FixPak on top of the base DB2 UDB V8.1 installation. DB2 V8.1.4 (FixPak 4) was used in the case studies and examples in this manual.

Install Development Center

Development Center is included in the Application Development Client component. Install any components that include the ADC. We installed the Development Center as part of the *typical* install of DB2 UDB V8.1 since we wanted a full DB2 UDB on our Windows workstation in addition to the Development Center. We then applied FixPak 4.

Development Center is installed into the default directory c:\Program Files\IBM\sqllib.

Install DB2 Connect

We used the Enterprise Server Edition of DB2 UDB V8.1 at FixPak 4, which includes a personal edition of DB2 Connect and DB2 UDB for our workstation. This allowed us to prototype our stored procedure development on DB2 UDB on Windows before installing the stored procedure on DB2 for z/OS. However, installing DB2 UDB on the workstation is not a

prerequisite for using Development Center to create SQL and Java stored procedures for DB2 for OS/390 or z/OS.

Bind the client to the server

Once the installation is completed, the following bind needs to be performed for the first user that uses Development Center on this DB2 server. The bind is performed on the workstation from a DB2 command line processor window. Click **Start-> Programs -> IBM DB2 -> Command Line Tools -> Command Line Processor**. Position to the directory .\SQLLIB\bnd before issuing the BIND command shown in Example 29-1.

Example 29-1 Connecting and binding DC

```
DB2 CONNECT TO ssid USER username USING  password
DB2 BIND @ddcsmvs.lst BLOCKING ALL SQLERROR CONTINUE GRANT PUBLIC
```

The `ssid` parm is the DB2 alias for the z/OS DB2 server that has been set up with Configuration Assistant; see “Configure remote DB2 server access” on page 505. The “username” is the TSO logon ID and password is the TSO logon ID password. The bind may need to be repeated after applying a FixPak. If maintenance is applied, and you do not re-perform the bind, you will receive `-805` at the workstation when trying to use Development Center if it is necessary to perform the bind again.

Configure remote DB2 server access

Development Center connects to a remote DB2 server using an alias that has been set up either in the client workstation or in the DB2 Connect Server. When connecting to the alias, all the necessary information to make the connection is available including the DB2 Server database name, the host domain name server, and the TCP/IP listening port.

There are multiple ways to specify the connection information to a DB2 server. We used the steps in the following example to connect to our DB8A DB2 V8 server. Similar steps were performed to define the connection to our DB2G DB2 V7 server.

Start the Configuration Assistant (CA) by selecting **Start ->Programs ->IBM DB2 ->Set-up Tools ->Configuration Assistant** in the Configuration Assistant panel.

Check menu option **Selected ->Add Database Using Wizard**. Next, we went through the panels in Table 29-2 to define a connection to DB8A.

Table 29-2 Define DB2 alias DB8A using CA on z/OS

Panel	Panel Title	Action	DB8A set up
1. Source	Select how you want to set up a connection	Select a configuration option.	Check **Manually configuring a connection to a database**. Click **Next**
2. Protocol	Select a communications protocol	Selecting TCP/IP causes an additional check box to be displayed at the bottom of the panel.	We use TCP/IP for our protocol, and check the check box at the bottom of the panel, The database physically resides on a host or OS/400® system, connect directly to the server. Click **Next**.
3. TCP/IP	Specify TCP/IP communication parameters	Enter the MVS domain name and DB2 DDF port.	Host name-> WTSC.ITSO.IBM.COM Port->23456. Click **Next**.

Panel	Panel Title	Action	DB8A set up
1. Source	Select how you want to set up a connection	Select a configuration option.	Check **Manually configuring a connection to a database**. Click **Next**
4. Database	Specify the name of the database to which you want to connect	Enter the DB2 database location name in the Database name field. By default the same name is entered into the Database alias field, which you can override.	Enter DB8A for the Database name, which is automatically entered in the Database alias field. Click **Next.**
5. Data Source	Register this source as a data source	If Open Database Connectivity (ODBC) applications are using this database, then you specify how the database is registered on this panel.	Since we did not use ODBC, we skipped the Data Source panel and clicked **Next.**
6. Node Options	Specify the node options	The node for the database connection is specified here. This includes specifying the operating system that is connected to.	Select the **Operating system** from the pop-down list as OS/390 or z/OS. We left the other fields empty and clicked **Next.**
7. System Options	Specify the system options	This panel is filled in from entries in the previous panels.	We entered nothing on this panel and clicked **Next.**
8.Security Options	Specify the security options	This panel specifies where security authentication is performed.	We selected Server authentication (SERVER) and clicked **Next.**
9. DCS Options	Specify the DCS options	This panel customizes the direct connection to our z/OS DB2 server.	We checked Configure DCS options, which checks Disconnect if client generates an interrupt (INTERRUPT_ ENABLED) and clicked **Finish**.

29.2.2 z/OS setup

Next, we now show the steps for the z/OS setup. The APARS listed in Table 29-3 need to be applied to the system before using Development Center.

Table 29-3 APARs for Development Center

Language	Component supported	APAR
Java	SQLJ	PQ65125
Java and SQL	SQL Assist	PQ62695
Java and SQL	DB2 Connect V8	PQ62695, PQ55393, PQ56616, PQ54605, PQ46183, PQ62139
SQL	DB2 V6 and V7 DSNTPSMP 1.15 or higher	PQ45854
Java	DB2 V7	PQ59735, PQ52329

The minimum prerequisites are listed next.

- SQL and Java stored procedures:
  - OS/390 2.8 or greater
  - LE
  - WLM
  - RRS

  - REXX language
  - Unicode support
- SQL stored procedures
  - C compiler
- Java stored procedures
  - JDBC driver
  - SDK 1.3.1

Development Center interacts with the OS/390 and z/OS DB2 server 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
- An Explain-like stored procedure can be set up that is optional on the SQL Statement panel with both the SQL and Java wizards used by Development Center.
  - SYSPROC.DSNWSPM (Actual Costs) setup is currently a manual job submission.

Development Center authorization set up

This section describes general authorities and privileges needed for Development Center tasks when there are no secondary authorization IDs being used. Table 29-4 summarizes them.

Table 29-4 General authorities and privileges for all platforms using DC

Task	Authorities and privileges
Access target databases	CONNECT
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

Task	Authorities and privileges
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
Windows, AIX and Sun only: to use the IBM Distributed Debugger to debug Java stored procedures.	Table privileges (SELECT, UPDATE, etc.) for the debug table (DB2DBG.ROUTINE_DEBUG) and the source table.
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 Development Center accesses a number of DB2 system catalog tables on z/OS and OS/390. The user connecting to DB2 for z/OS and OS/390 must hold privileges described in Table 29-5, Table 29-6 for SQL stored procedures, and Table 29-7 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 29-5 Execute privileges required to use Development Center

Additional required privileges:
- EXECUTE privilege on DSNTPSMP- for SQL
- EXECUTE privilege on DSNTJSPP - for Java, for JDBC and SQLJ
- EXECUTE privilege on DSNTBIND - for Java, for SQLJ

For DB2 for OS/390 V6 and V7, and DB2 for z/OS V8, the Development Center accesses the following catalog and non-catalog tables when creating SQL stored procedures.

Table 29-6 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 OS/390 V7 and DB2 for z/OS V8, the Development Center accesses the following catalog tables when creating JAVA stored procedures.

Table 29-7 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

For DB2 for OS/390 V7 and DB2 for z/OS V8, the WLM address space where SYSPROC.DSNTJSPP executes requires to be authorized to access the USS /tmp directory. Additionally, the WLM procedures where the user's Java stored procedures run also require to be authorized to access the USS /tmp directory.

This access is set up under the userid associated with the jobname for these WLM procedures.

29.2.3 Unicode support

Starting in DB2 for OS/390 V7, DB2 Development Center users creating SQL and Java Stored Procedures will experience incorrect codepage translation when Unicode Conversion Services (UCS) is not set up. See the Web site and the manual *Support for Unicode: Using Conversion Services*, SC33-7050 for more information:

```
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 will 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:

- Installing PTFs (see II13048 and II13049)
- 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 against DB2 for OS/390 V7. However, restoring the source from the database of a previously created Java stored procedure on DB2 for OS/390 V7 will return the source as a single line with red blocks interspersed, which represent line feeds that have not been translated correctly. The Development Center support for SQL stored procedures handles the code conversion, and UCS is not required. The support for SQL code page translation was introduced in DB2 Stored Procedure Builder FixPak 8, and is included in the base Development Center code.

The workaround until UCS is installed is to add the DISABLEUNICODE=1 parm to the workstation or DB2 Connect server ..\sqllib\db2cli.ini file. Include this parm for each z/OS DB2 server alias without Unicode Conversion Services installed. It is recommended to put this

parm at the alias level and not the [COMMON] level. This workaround will not support those environments where multiple CCSIDs are required for processing.

If the DISABLEUNICODE=1 work around is used initially when creating SQL or Java stored procedures, then Unicode Conversion Services is installed and set up, no impact occurs to the previously created SQL or Java stored procedures. The source can be retrieved with UCS or with DISABLEUNICODE=1 set.

29.2.4 Set up for SQL and Java stored procedures

Development Center uses DB2-supplied stored procedures to build both SQL and Java stored procedures customized by the following jobs:

- DSNTIJSD
- DSNTIJRX
- DSNTIJTM
- DSNTIJMS
- DSNTEJ6W
- DSNTIJSG

DSNTIJSD

This customization job sets up SQL Debugger support for language SQL stored procedures in DB2 for z/OS V8.

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 Assist support. SQL Assist is used in many of the client development tools including DB2's tools:

- Command Center
- Control Center
- Development Center

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 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=DB8AWLM,REFRESH
```

where DB8AWLM is the environment name we want to refresh.

Table 29-8 describes the WLM commands.

Table 29-8 WLM commands entered from SDSF

D WLM,APPLENV=DB8AWLM	Displays the environment status
V WLM,APPLENV=DB8AWLM,QUIESCE	Stops the environment
V WLM,APPLENV=DB8AWLM,RESUME	Resumes the environment
V WLM,APPLENV=DB8AWLM,REFRESH	Refreshes the environment

Development Center 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 Development Center 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 29-9.

Table 29-9 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 Development Center
- Miscellaneous stored procedures

Set up specific to SQL stored procedures

The following setup is needed for Development Center to create SQL stored procedures. The same set up applies when creating SQL stored procedures using the WSAD, WSADIE, WSED 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

APAR PQ45854 provided support for DSNTPSMP at V1.15. This level of DSNTPSMP includes information to assist with problem determination of the build process. An optional DD statement introduced with this maintenance is //CFGTPSMP and 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 DB8A is listed in Example 29-2.

Example 29-2 Sample CFGTPSMP configuration data set

```
;-THE CONFIGURATION KEYWORDS AND VALUES ARE AS FOLLOWS:
;-
;-                   .-CBCDRVR--.
;-C_COMPILER--=--+----------+----------------------------|
;-               |-CCNDRVR--|
;-               |-CBC32OPP-|
;-               '-EDCDC12O-'
;-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.
;
```

Set up specific to Java stored procedures

The following set up is needed for Development Center to create Java stored procedures. The same set up applies when creating Java stored procedures using the WSAD, WSADIE, and WSED products:

- Install JDBC drivers

- Create permanent mount point on USS
- Customize DSNTIJSG for DSNTJSPP and DSNTBIND
- Create DSNTJSPP.properties file on USS
- Create db2sqljjdbc.properties file on USS
- Create the JAVAENV data set
- Verifying the DB2 and Java set up 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 a permanent mount point on USS

DSNTJSPP executes part of its code under USS. Two files are used by DSNTJSPP and need to be created on the HFS system. If you are new to HFS, Example 29-3 shows how you create a mount point and HFS file system that is needed by the DSNTJSPP.properties and db2sqljjdbc.properties.

Example 29-3 Create a Mount Point

```
TSO example:
   MKDIR '/u/DB8AU' MODE(7 5 5)
```

Create a new HFS data set (as shown in Example 29-4) to hold small data sets and temp files used by DSNTJSPP during the build of the Java stored procedure.

Example 29-4 Create an HFS data set

```
JCL example:

   //MYHFS   DD   DSN=MY.HFS,
   //             SPACE=(TRK,(10,1,1)),
   //             DCB=DSORG=PO,
   //             DSNTYPE=HFS,
   //             DISP=(,CATLG),
   //             STORCLAS=SCCOMP
```

Once the mount point is defined and the HFS file is created, the HFS is mounted on the mount point as shown in Example 29-5.

Example 29-5 Mount the HFS file

```
TSO example:

MOUNT FILESYSTEM('MY.HFS') TYPE(HFS) MODE(RDWR) MOUNTPOINT('/u/SSIDU')
```

Finally, we added the MOUNT directive in the BPXPRMxx member used at IPL to make this HFS system always available to us. This is shown in Example 29-6.

Example 29-6 Add MOUNT directive example

```
BPXPRMxx example:

   MOUNT FILESYSTEM('MY.HFS')
      MOUNTPOINT('/u/DB8AU')
      TYPE(HFS)  MODE(RDWR)
```

Customize DSNTIJSG for DSNTBIND and DSNTJSPP

You must customize and run the DSNTIJSG member on the target DB2 subsystem. The member contains the DDL that defines new procedures for DSNTJSPP and DSNTBIND, binds the packages of the stored procedures.

DSNTBIND procedure registration customization needs the WLM environment defined. Specify the same WLM environment used by DSNTPSMP as shown in Example 29-7.

Example 29-7 Define DSNTBIND to execute in DB8AWLMR with DSNTPSMP

```
...
WLM ENVIRONMENT DB8AWLMR
...
```

DSNTJSPP procedure registration customization needs the WLM environment defined. Additionally, the RUN OPTIONS, “HOME” parameter needs to specify the path-name of the UNIX System Services directory we defined in Example 29-4 on page 513. This directory includes the DSNTJSPP.properties and the db2sqljjdbc.properties files that are defined next. See Example 29-8. Notice that all HFS files are case sensitive.

Example 29-8 Partial DSNTJSPP registration information

```
...
WLM ENVIRONMENT DB8AWLM
...
RUN OPTIONS 'POSIX(ON),ENVAR("HOME=/u/DB8AU")'
...
SECURITY DB2;
```

Example 29-9 shows our DSNTJSPP.properties file.

Example 29-9 Our /u/DB8AU/DSNTJSPP.properties file

```
#DB2_HOME is where the JDBC driver is installed
DB2_HOME=/usr/lpp/db2/db2810
#JAVA_HOME is where the current SDK is installed
JAVA_HOME=/usr/lpp/java/IBM/J1.3
DB2SQLJPROPERTIES=/u/DB8AU/db2sqljjdbc.properties
DB2SQLJPLANNAME=SQLJPLAN
DB2SQLJSSID=DB8A
CLASSPATH=.:$DB2_HOME/classes/db2j2classes.zip
PATH=$JAVA_HOME/bin:$DB2_HOME/bin:/bin
LIBPATH=$DB2_HOME/lib
STEPLIB=DB8A8.SDSNEXIT:DB8A8.SDSNLOAD:DB8A8.SDSNLOD2:CEE.SCEERUN
```

The continuation character, if needed, is the UNIX \ with all statements starting in column 1. Comments are denoted with a # in column 1. The STEPLIB informationHFS defined below is needed even if the data sets are defined in Linklist. The <hlq>.SDSNLOD2 data set is new for Java stored procedure support. The data set pointed to by the DB2SQLJPROPERTIES file can be any name you like, with a default of ../db2sqljjdbc.properties. Whatever HFS file is specified in the DB2SQLJPROPERTIES parameter must exist. The two files cannot be combined into one, that is, you cannot add the DB2SQLJDBRMLIB parameter to DSNTJSPP.properties and remove the HFS file pointed to by DB2SQLJPROPERTIES.

Create db2sqljjdbc.properties file on USS

This HFSHFS file only requires the DB2SQLJDBRMLIB parm to identify the DBRM library used for SQLJ stored procedures, as shown in Example 29-10.

Example 29-10 Our /u/DB8AU/db2sqljjdbc.properties file

```
DB2SQLJDBRMLIB=DEVL7083.DEVL.DBRM
```

Create the JAVAENV data set

See 17.2.6, “Setting up the JAVAENV data set for Java stored procedure execution” on page 249.

Verifying the DB2 and Java set up on z/OS

Before using DC, you can verify your system set up by running the ACMEJOS example. 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 DC, requires a .profile to be set up, and an example exists for creating this as well. Sample code for all steps can be found in the IBM Redbook *DB2 Java Stored Procedures Learning by Example*, SG24-5945.

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 or DSNTJSPP

Multiple versions of DSNTPSMP for SQL stored procedures or DSNTJSPP for Java 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 SQL stored procedures, or for the subsystem ID in the DSNTJSPP.properties file for Java stored procedures.

When multiple versions of either of these DB2-supplied stored procedures are required, first register a new copy of either DSNTPSMP or DSNTJSPP with a new schema (SYSPROC is the default). In that registration, specify a new WLM proc where this copy of DSNTPSMP or DSNTJSPP will execute and complete other similar steps as described in “Set up specific to SQL stored procedures” on page 511 for SQL stored procedures, or see “Set up specific to Java stored procedures” on page 512 for Java stored procedures.

Selecting a different version of DSNTPSMP or DSNTJSPP

Java stored procedures can specify a different schema value to create multiple versions of DSNTJSPP, where all versions execute the same DB2 supplied DSNTJSPP C stored procedure. 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 from clicking **Project -> Environment Settings -> Build options**. Figure 29-4 shows an example for language *SQL* where a different schema value from the SYSPROC default, and optionally a different build utility name from the DSNTPSMP default can be specified. Similarly, changing the build utility schema for Java is done by selecting language **Java,** then entering the new schema value.

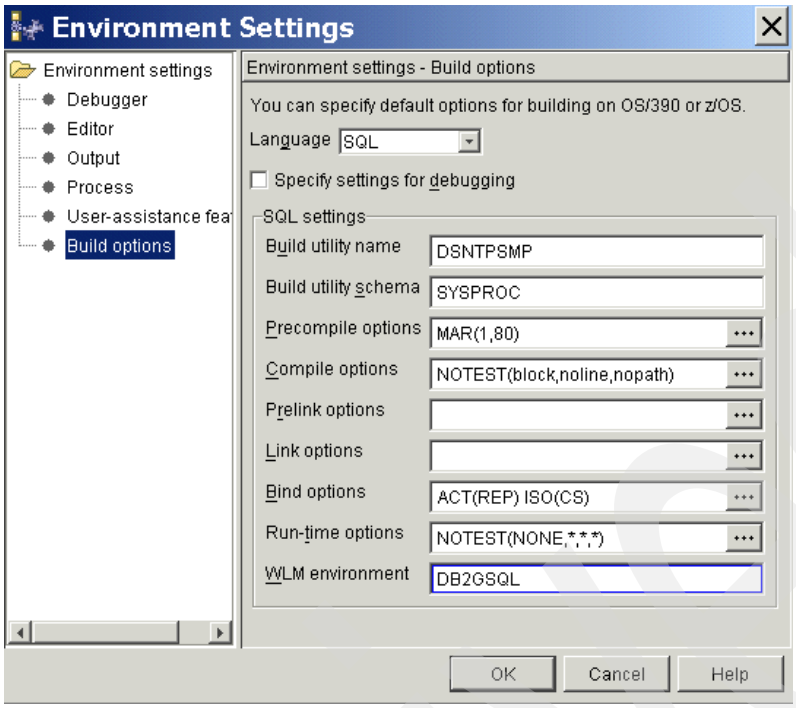


Figure 29-4 Environment setting with different schema

When creating a new SQL or Java stored procedure using the **New->Stored Procedure** using wizard, a different build schema and utility (SQL only) can be selected from the **Options -> Advanced->z/OS Options->Build Options->Build utility** as shown in the example of Figure 29-5. You can also override the build utility for an existing SQL or Java stored procedure from Project View by selecting the stored procedure, then right-click **Properties->Options->Build utility.**

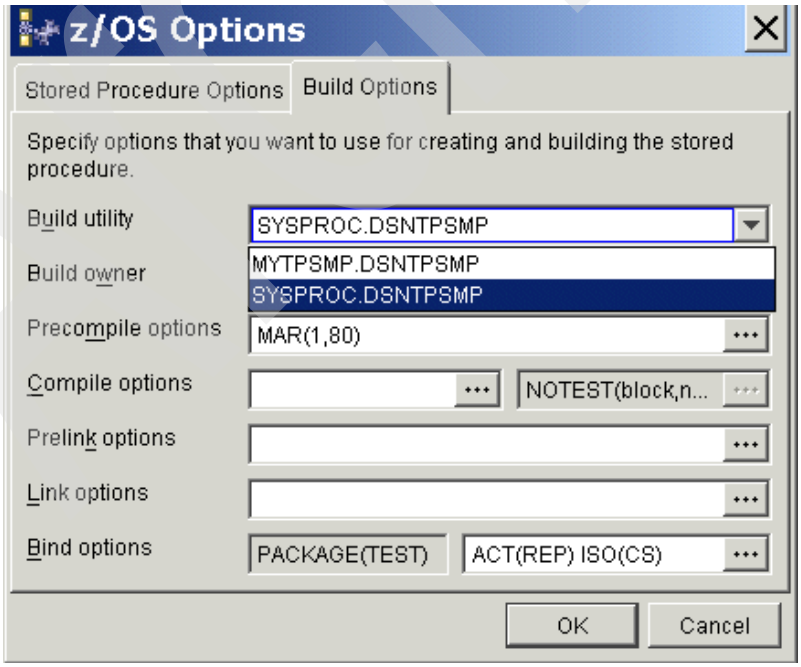


Figure 29-5 Multiple versions of schema

29.2.5 Development Center Actual Costs set up

SYSPROC.DSNWSPM is a stored procedure optionally used on the SQL Statement panel panel 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

Define DSNWSPM as a stored procedure

The CREATE PROCEDURE definition is described below. 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. See Example 29-11.

Example 29-11 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 Example 29-12.

Example 29-12 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 set up 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)
```

29.2.6 Development Center and JDBC Driver selection

This section describes your options for JDBC Driver selection when using the DB2 SPB or the DB2 DC to create Java stored procedures on OS/390 and z/OS platforms. You can find this information at:

http://www-106.ibm.com/developerworks/db2/library/techarticle/dm-0408rader/index.html

Overview

Both the DB2 SPB and the DB2 DC use a JDBC driver in the following three areas:

- The server connection
- The generated SP (only Java SQLJ includes Driver significance 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.

This is depicted in Figure 29-6.

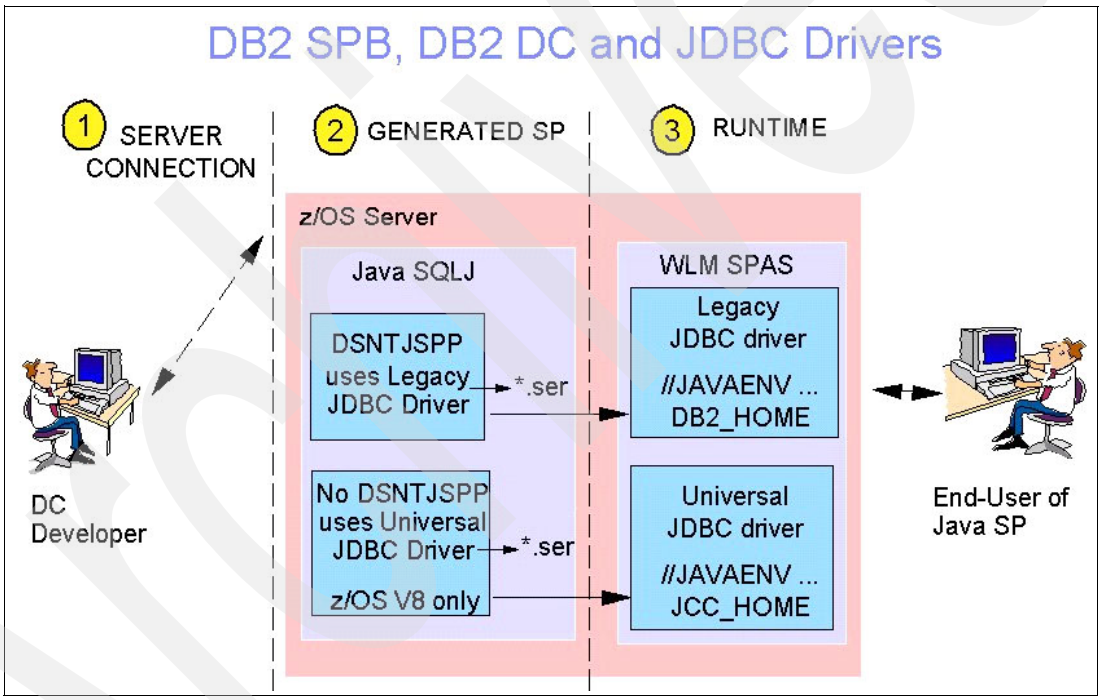


Figure 29-6 SPB, DC and the usage of JDBC Drivers

When using SPB included in DB2 UDB V7.2 or DC included in DB2 UDB V8.1, the Legacy JDBC Driver is used for the server connection as well as the Driver in the generated Java stored procedure created during the build process. When using DC included in DB2 UDB V8.2, either the Legacy JDBC Driver or the Universal JDBC Driver can be selected for the server connection. When using DC included in DB2 UDB V8.2 to create Java stored procedures on a DB2 for z/OS V8 server, either the Legacy JDBC Driver or the Universal JDBC Driver can be selected. APAR PK01445, currently open, will make the Universal JDBC Driver available with DB2 Version 7.

The Legacy JDBC Driver is used when the generated SP is built using DSNTJSPP. The Universal JDBC Driver is used when the generated SP is built without DSNTJSPP. This selection is either determined by the default set in the DC Environment settings -> Build options -> Java, or overridden from the Options panel for a specific stored procedure. The

build process includes all the steps to create and install the Java SP on the z/OS server, including the generated stored procedure.

Build process generates Java stored procedure

The build process performed by SPB and DC 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 SPB or DC using DB2PROFC for the Legacy JDBC Driver and by DC using DB2SQLJCUSTOMIZE for the Universal JDBC Driver (applies to DB2 for z/OS V8 only).

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 executes. When the DB2_HOME environment variable is specified, the Legacy JDBC Driver is used. When the JCC_HOME environment variable is specified, the Universal JDBC Driver is used. If both DB2_HOME and JCC_HOME are included in the //JAVAENV DD statement, then the Legacy JDBC driver is selected.

However, once APAR PQ84490 is applied, if both DB2_HOME and JCC_HOME are included, the Java WLM SPAS will fail initialization and a DSNX961I message will be returned. The default directory structure for DB2_HOME is /usr/lpp/db2/db2710. The default directory structure for JCC_HOME is /usr/lpp/db2/db2710/jcc. For more details on the //JAVAENV DD statement, see 17.2.6, “Setting up the JAVAENV data set for Java stored procedure execution” on page 249.

In Table 29-10 we show the SPB and DC JDBC Driver options for Server Connection, Generated SP and Runtime.

Table 29-10 SPB and DC JDBC Driver options

DB2 UDB V8.1 or V8.2	Legacy Driver Type 2 connectivity [a]	Universal Driver Type 2 connectivity [a]	Universal Driver Type 4 connectivity [b]
SPB or DC included in DB2 UDB V8.1			
Server Connection	Y		
Generated SP (uses DSNTJSPP on z/OS)	Y		
Runtime for Java JDBC [c]	Y	Y	
Runtime for Java SQLJ	Y		
DC included in DB2 V8.2			
Server Connection	Y	Y	Y
Generated SP (uses DSNTJSPP on z/OS)	Y		
Generated SP (not using DSNTJSPP on z/OS)		Y[d]	

DB2 UDB V8.1 or V8.2	Legacy Driver Type 2 connectivity [a]	Universal Driver Type 2 connectivity [a]	Universal Driver Type 4 connectivity [b]
Runtime for Java JDBC [c]	Y	Y	
Runtime for Java SQLJ	Y	Y[d]	

a. It requires DB2 Connect
b. It requires a DB2 Connect license only
c. Since Java JDBC stored procedures do not include significance in the stored procedure that is created by SPB or DC as is done for Java SQLJ stored procedures, for example there is no *.ser file, they can be created using the DB2-supplied stored procedure, DSNTJSPP (used on DB2 for OS/390 V7 or DB2 for z/OS V8) or not (DB2 for z/OS V8 only option). Java JDBC stored procedures only determine the driver that is used for the stored procedure during runtime of that stored procedure. The Legacy JDBC driver is used when the WLM proc where the Java stored procedure executes has the //JAVAENV DD statement pointing to DB2_HOME, with a default directory of /usr/lpp/db2/db2710. The Universal JDBC driver is used when the WLM proc where the Java stored procedure executes has the //JAVAENV DD statement pointing to JCC_HOME, with a default directory of /usr/lpp/db2/db2710/jcc.
d. It applies to DB2 for z/OS V8 only, see 29.6, “Future Development Center enhancements” on page 550.

See *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java,* SC18-7414 for more information on the Legacy JDBC Driver (described under JDBC/SQLJ Driver for OS/390) and the Universal JDBC Driver.

29.2.7 Java SDKs used by DB2 Development Center on OS/390 and z/OS

In this section we describe your options for SDK selection when using the DB2 Development Center (DC) to create Java stored procedures on OS/390 and z/OS platforms.

Overview

Both Java Software Development Kit (SDK) 1.3.1 and SDK 1.4.1 are supported when creating Java stored procedures on DB2 for OS/390 V7 and DB2 for z/OS V8. The two areas that reference an SDK are:

- The build time of the stored procedure by DC, which includes the generated Java source and the subsequent compilation of that source
- 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 will occur in the WLM SPAS. For instance, using a Java stored procedure that includes a Java SDK 1.4.1 method will fail if it executes in a WLM SPAS referencing a Java 1.3.1 JRE.

Development Center is included in the Application Development Client (ADC) of DB2 UDB. Two releases are now available. DB2 UDB V8.1, which includes FixPaks (FP) from FP1 through FP7, and DB2 UDB V8.2. DB2 UDB V8.1 at a FP7 level is equal to the base DB2 UDB V8.2 code level. Development Center included with DB2 UDB V8.1 is referred to as DC 8.1 and included with DB2 UDB V8.2 is referred to as DC 8.2.

The term SDK is used on the z/OS server and either Java Development Kit (JDK) or SDK is used on the client workstation.

Build time processing by DC

Considerations during build time include the selection of using the DB2-supplied DSNTJSPP stored procedure or not. When processing against a DB2 for OS/390 V7, using DSNTJSPP is

required for the build process. When using DC 8.2 and processing against a DB2 for z/OS V8 server, DSNTJSPP can be selected for the build process or not.

When using DSNTJSPP, the set up of the Hierarchical File System (HFS) DSNTJSPP.properties file is required. This file includes the JAVA_HOME environment variable that identifies whether SDK 1.3.1 or SDK 1.4.1 is being used. The default directory for the Java SDK 1.3.1 is /usr/lpp/java/IBM/J1.3. The default directory for the Java SDK 1.4.1 is /usr/lpp/java/IBM/J1.4. To determine the specific level you have installed for either the SDK 1.3.1 or SDK 1.4.1, see 17.2.2, “Checking that the Java SDK is at the right level” on page 247.

When DSNTJSPP is not used for the build processing, supported only from DC 8.2 processing against a DB2 for z/OS V8, the generated Java stored procedure source and subsequent compilation of that source only uses the SDK specified on the workstation.

A default SDK is included with DB2 UDB and used by Development Center. For DB2 UDB V8.1, including FixPaks through FP6 the SDK is at a 1.3.1 level. For DB2 UDB V8.2 an DB2 UDB V8.1- FP7 the default SDK is at a 1.4.1 level. The default directory for the SDK is ..\sqllib\java\jdk.

Overriding the default SDK

Overriding the default SDK used by the Development Center client takes two steps. In this example we show a directory location for JDK 1.3.

First the SDK directory location needs to be entered for a specific SDK. This is done from the Project->Environment Settings->Process->Java Home panel. See Figure 29-7.

The SDK is referred to as *JDK* on this panel. The JDK level is selected from a list box and the directory associated with that JDK is entered in the directory field.

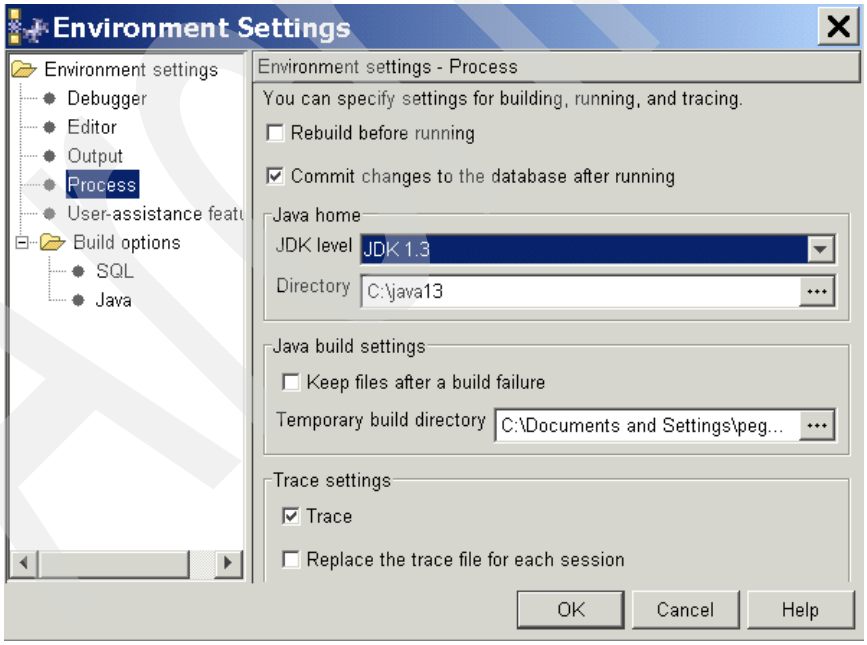


Figure 29-7 Overriding the default SDK - Step 1

Secondly, this SDK is selected for the client build process after the Database Connection is added to a project. See Figure 29-8.

From Project View positioned on the database alias, right click->Properties->Java Build Settings. Specify the JDK release from the list box and click OK.

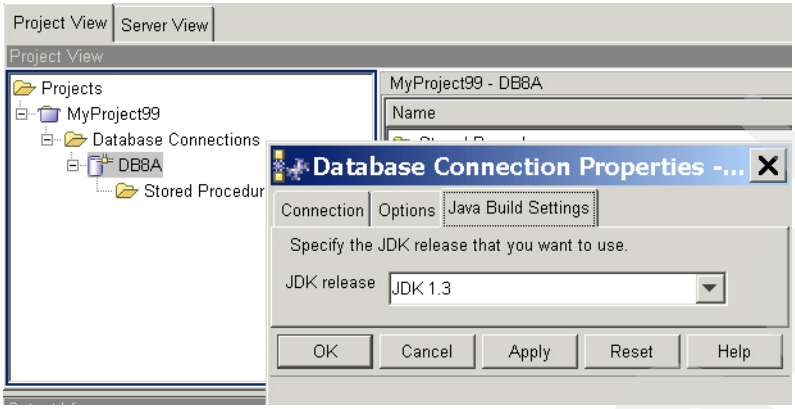


Figure 29-8 Overriding the default SDK - Step 2

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 OS/390 V7 includes support for the SDK 1.3.1, with support for the SDK 1.4.1 provided in APAR PQ76769. 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 either DB2 V7 or 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 will not initialize and the following error message will be 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 will also be 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 29-13 is for SDK 1.3.1.

Example 29-13 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 29-14 is for SDK 1.4.1.

Example 29-14 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 29-15 is for SDK 1.3.1.

Example 29-15 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 29-16 is for SDK 1.4.1.

Example 29-16 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)
```

29.3 A guided tour through Development Center

This section applies to all platforms supported by Development Center. Development Center includes the following four views that can be docked into a single window, or undocked to a separate window:

- Project View
- Server View
- Output View
- Editor View

Project View

The Project View is the main development view for managing your Development Center projects. The Project View consists of two areas: the object tree and the contents pane.

In the object tree of the Project View, you can:

- Manage multiple projects
- Manage multiple database connections
- Under each connection, create and manage objects such as stored procedures, UDFs, and structured types. UDFs and structured types are supported for the Windows, UNIX, and Linux platforms only.

The Project View shows objects that are under development. To see a complete or filtered list of objects in the database, including tables, triggers, and views, you can use the Server View.

The contents pane of the Project View gives you a closer look at the contents of the object that you select in the object tree. For example, if you select the Stored Procedures folder in the object tree, the contents pane shows you a list of the stored procedures in that folder. If you select a specific stored procedure or UDF in the Project View, the contents pane shows you the details of the selected object. The details of a particular stored procedure or UDF can include the name, the schema, the specific name, parameters, SQL package, result sets, language, fenced status, and comments.

You can re-size a column by dragging the column header border. You can also change the order of the columns by dragging and dropping.

You can use the push buttons at the bottom of the contents pane of the Project View to customize the contents pane. You can:

- Sort the columns
- Filter the rows of the view, using one or more columns
- Customize which columns you want to see listed
- Search the contents for strings

Server View

The Server View gives you a look at the contents of the server for each database connection that you have opened in the Project View, or that you explicitly added to the Server View. In addition to viewing the stored procedures and UDFs (Windows, UNIX, and Linux DB2 servers only) that are on the server, you can also use the Server View to view and work with other database objects such as tables, triggers, and views.

The Server View consists of two areas: the object tree and the contents pane.

To open the Server View, click **View -> Server View**. With the Server View, you can:

1. Place a filter on the object type folders in the object tree to control the number of objects that appear in the contents pane:
   - View the properties of the objects
   - Sample the contents of tables and views by querying the first *n* rows
   - View the source of stored procedures and UDFs
   - View the definitions of tables, views, and triggers
   - Add a Server View connection to your project
   - Add a stored procedure or UDF to your project.
2. You can use the push buttons at the bottom of the contents pane of the Server View to customize the contents pane. You can:
   - Sort the columns
   - Filter the rows of the view, using one or more columns
   - Customize which columns you want to see listed
   - Search the contents for strings

Output View

The Output View shows the results of the development tasks that you performed or attempted to perform. The Output View consists of two areas: the action list and the output pane.

The action list has 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 build, add, export, import, or run.

- The Object Name column shows the name of the object for each action.

In the output pane, you can select one of three pages 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.

You can remove an item from the action list by right-clicking the action and clicking **Remove** or **Remove All.**

Editor View

Use the Editor View for editing and manipulating the source code of an existing routine such as a stored procedure or UDF (UDF support applies to distributed DB2 servers only). The default editor supports cut, copy, paste, find and replace, menu and keyboard shortcuts, and syntax highlighting. You can change the default key behavior of the editor to match vi or Emacs keyboard commands by using the Environment Settings notebook.

The Editor View includes the Breakpoints, Call Stack, and Variables views for debugging SQL routines.

You can edit the source code of both SQL and Java routines. If you open a project that contains an existing SQL or Java routine, you can modify the source code (including SQL statements) in the editor. Development Center either drops the old routine from the database and creates a routine that reflects the changes you made, or it alters the routine.

Changes to the source code of Java stored procedures rarely cause the procedure to be dropped. Where possible, changes to the source code of SQL stored procedures results in an ALTER command rather than a DROP command.

Requirement: Java routines built by the Development Center 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.

To edit the source code of a routine:

a. In the Project View, right-click the routine that you want to modify, and click **Edit**. The source code of the routine displays in the Editor View.

b. Edit the source code. You can:

   - Change or add SQL statements directly in the editor
   - Click **Edit -> Insert SQL** to open SQL Assist.
   - Click **Edit -> Insert SQL Fragment** to insert one of the following code fragments: IF, CASE, LOOP, REPEAT, and WHILE.
   - Click **Edit -> Check** to check the validity of the source.

c. To save your changes, click **File -> Save Object** or **File -> Save All Objects**.

d. To close the object, click **File -> Close Object** or **File -> Close All Objects**.

Import wizard

Use the Import wizard to import routines to your project. To open the Import wizard:

1. In the Project View, select a folder for a type of object.
2. Click **Selected -> Import**. The Import Source window opens.
3. In the Import Source window, select the type of object or file that you want to import, and select the source from which you want to import.
4. Click **OK**. The Import wizard opens.

Export wizard

Use the Export wizard to export routines from your current project for later deployment. You can choose from two main export options, both resulting in a zip file that you can use later for deployment.

- The wizard can export a project, including supporting files, that you can deploy later using the Deployment Tool.
- The wizard can export a deployment script and a set of supporting files. You use the generated deployment script for manual deployment from a command line.

To export routines using the Export wizard:

1. In the Project View, select a database connection or an object folder.
2. Click **Selected --> Export**.
3. Complete the necessary steps of the wizard.
4. Click **Finish**. The wizard exports the routines that you specified

Deployment 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 Project View, select a database connection.
2. Click **Selected -> Deploy.**
3. Complete the necessary steps of the wizard.
4. Click **Finish**. The wizard deploys the routines to the target database.

Deployment tool

Use the Deployment Tool to deploy to a target database the routines that are saved in a file. The target database must be compatible with the database that the object was created for. You normally use the Deployment Tool to deploy routines that are contained in a zip file that you exported using the Export wizard.

To deploy exported routines using the Deployment Tool:

1. Open the Deployment Tool.
2. On the Deployment Source page, choose the file that you want to deploy (typically, a zip file that you exported using the Export wizard).

3. On the Target Database page, select the target database for the deployment and specify your user ID and password for the target database.
4. On the Select Object page, select the specific routines in the deployment file that you want to deploy.
5. On the Options page, specify deployment and error-handling options.
6. On the Deploy page, click **Deploy**. A status message displays the success or failure of the deployment.

Menu Bar

The Development Center menu bar includes the following main selections:

- Project

  Use this menu to create a new or open an existing project, access the Environment Settings notebook, or the launchpad, and to exit the Development Center.

- File

  Use this menu to save or close objects that you are currently editing. You can also get a file to insert in your current object.

- Edit

  Use this menu to work with the object that you are currently editing.

- Selected

  Use this menu to display and select the available actions for the object that is selected in the object tree or contents pane.

- View

  Use this menu to open additional Development Center views. You can open the Editor View, Output View, and the Server View. See 29.4.3, “Arranging the development views” on page 533.

- Tools

  Use this menu to open any of the DB2 centers. Some of the centers are also available by clicking the icons in the toolbar.

- Help
- Use this menu to display online help, product information, and to open the Information Center.

29.4 Getting started with Development Center

This section describes how Development Center is used to perform the following tasks:

1. Starting Development Center for the first time
   - Configuring environment settings
2. Using SQL Assist
3. Arranging the development views
4. Creating a new stored procedure:
   - Use the new object wizards
   - Use the Import wizard
5. Creating a sample SQL stored procedure for z/OS

   - Creating an SQL stored procedure on DB2 for z/OS V8 for debugging.
6. Creating a sample Java stored procedure on z/OS:
   - JDBC
   - SQLJ
7. Importing the SQL stored procedure from case study
8. Importing a sample Java stored procedure on z/OS:
   - JDBC
   - SQLJ
9. Using the DC SQL wizard to generate multiple SQL statement with single result set
10. Using the DC Java wizard to generate multiple SQL statements returned to multiple result sets

29.4.1 Starting the Development Center for the first time

We start Development Center by selecting **Start -> Programs -> IBM DB2 -> Development Tools ->Development Center.**

The first time Development Center is started, a **Development Center Launchpad** is displayed. The launchpad guides you through the steps to:

- Create a project
- Create a connection
- Create an object

Create a project

We used the Launchpad and entered `DEVL7083` for our project name when prompted by the wizard and clicked **OK** to continue.

Create a connection

The Add Database Connection Wizard is launched when we clicked **Create a connection.** A set of four panels is presented:

- Connection type

  Either Offline or Online types are available. Since we were going to access the database during the connection, we chose Online.

- Connection

  On the connection panel, we select our DB8A alias that has previously been configured with CA for our DB2 for z/OS V8. If the alias had not been previously set up, we would select **Add** to add the alias.

- Options

  See “Development Center authorization set up” on page 507 for more information on the use of these fields.

- Summary

  This panel summarizes our DB2 connection information.

Figure 29-9 shows how Development Center looks with our DEVL7083 project and our DB8A database connection added.

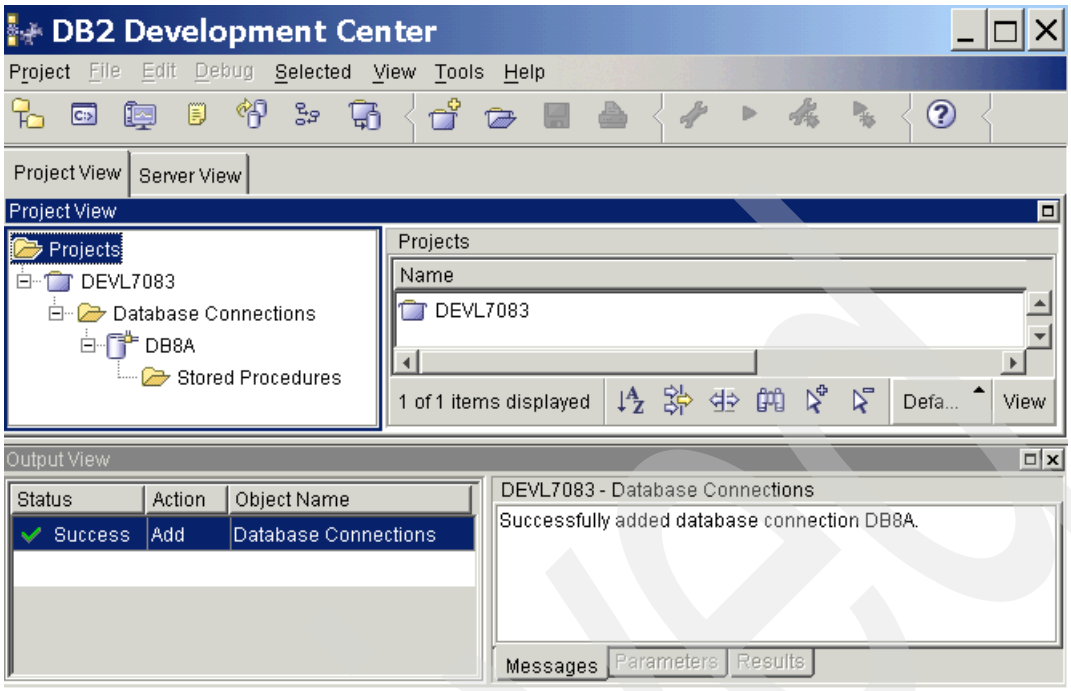


Figure 29-9 Development Center opened in Project View

Create an object

Clicking the **Create Object** button launches the New Object panel. We selected **SQL** for the type of stored procedure we want to create. We created a very simple SQL stored procedure to test our set up using the panel in Figure 29-10.

Figure 29-10 Create new SQL stored procedure

Click **OK,** and the Create SQL Stored Procedure wizard is launched. Five panels make up the wizard:

- Name

  We entered `DEVL7083.SQLTEST` for our name.

  Note: Development Center 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.

- Definition

We took defaults for the definition. A default SQL statement for testing setup is included in the SQL Statement on the Definition panel. The statement is SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES.

- Parameter

  We took defaults, no parameters were defined.

- Options

  We specified the COLLID of DEVL7083. Since we updated our default WLM environment for executing SQL stored procedures to point to our DB8A WLM AE, DB8ADS1, we did not have to make any updates to the z/OS Options panel, which is accessed after clicking the **Advanced** button.

- Summary

  This panel (shown in Figure 29-11) summarizes our SQL stored procedure. Optionally, we can view the SQL procedure definition by clicking the **Show SQL** button.

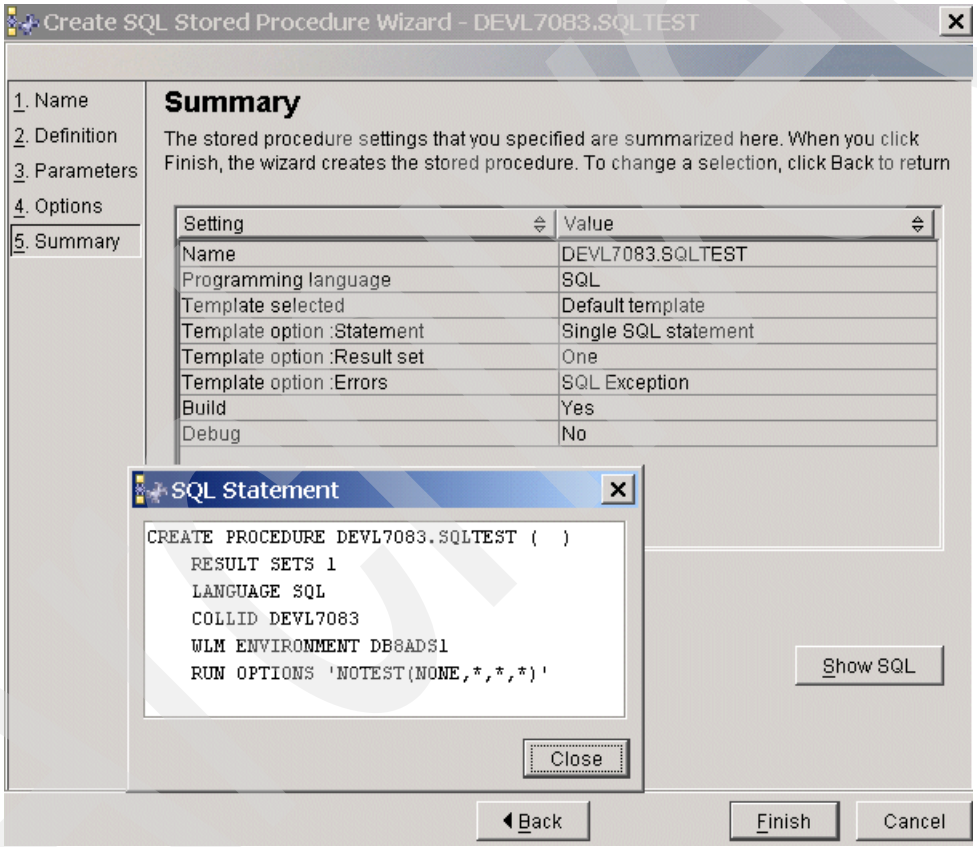


Figure 29-11 SQL stored procedure summary info including procedure definition

Once you are familiar with Development Center and you can check the **Do not show this window again** check box, and the launchpad will no longer be launched when starting Development Center.

Configure environment settings

Development Center has default settings for processing that are general to all platforms and some that are platform specific. With Development Center started, view or change these settings from menu option **Project-> Environment Settings**. The default settings are at a global level and set for all DB2 server connections. The build options for z/OS can be

overridden at build time in the **Options->Advanced options** buttons of the wizards, or after the stored procedure has been built in the properties for the stored procedure. The settings where defaults exist and can be changed follow:

- Debugger
  - Set Java Debugger listening port (distributed DB2).
  - Set SQL timeout value (DB2 V8 on Windows, UNIX, AIX, and z/OS).
- Editor
  - Specify options for editing source code for selected language.
- Output
  - Specify settings for results and refreshing of objects.
- Process
  - Specify settings for building, running and tracing.
- User-assistance features
  - Specify settings for field help and diagnostics.
- Build options
  - Specify settings for z/OS and OS/390 SQL and Java stored procedures.

The settings that are z/OS and OS/390 specific are:

- Click **Debugger ->Set SQL timeout value** (applies to DB2 for z/OS V8)
- Build options

  Two settings files exist: DB2DC.settings and YourName.settings. Changes to defaults are saved in YourName.settings file
  - SQL Language

    Figure 29-12 describes the default settings for SQL stored procedures on z/OS that are configured in the Environment Settings.

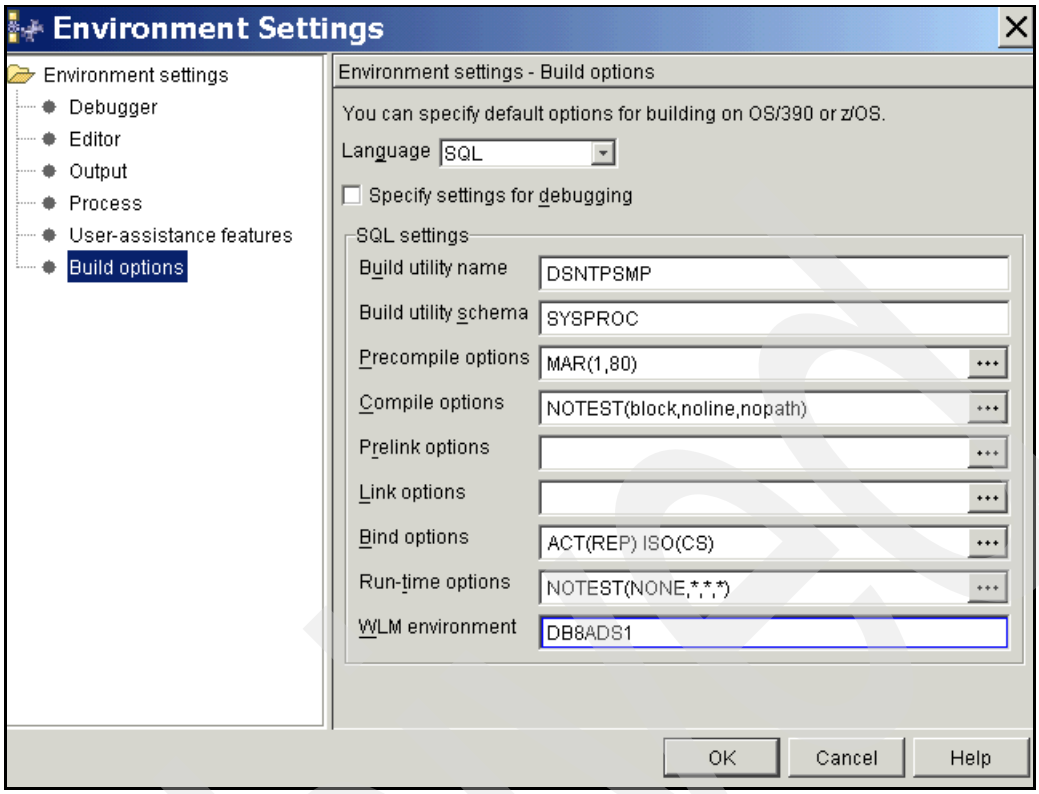


Figure 29-12 Default environment settings for SQL stored procedures

The Build Options panel is shown on this screen. There may be requirements for different JCL to run the SQL stored procedure build process. That is different source, DBRM, etc. libraries may be desired for test and production systems. See “Creating multiple versions of DSNTPSMP or DSNTJSPP” on page 515.

– Java Language

  Fewer fields are enabled on the Java Language setting panel. The default settings are shown in Figure 29-13. Multiple schemas may be required to support different resource requirements for the DSNTJSPP.properties file used by the build process of DSNTJSPP. See “Creating multiple versions of DSNTPSMP or DSNTJSPP” on page 515.

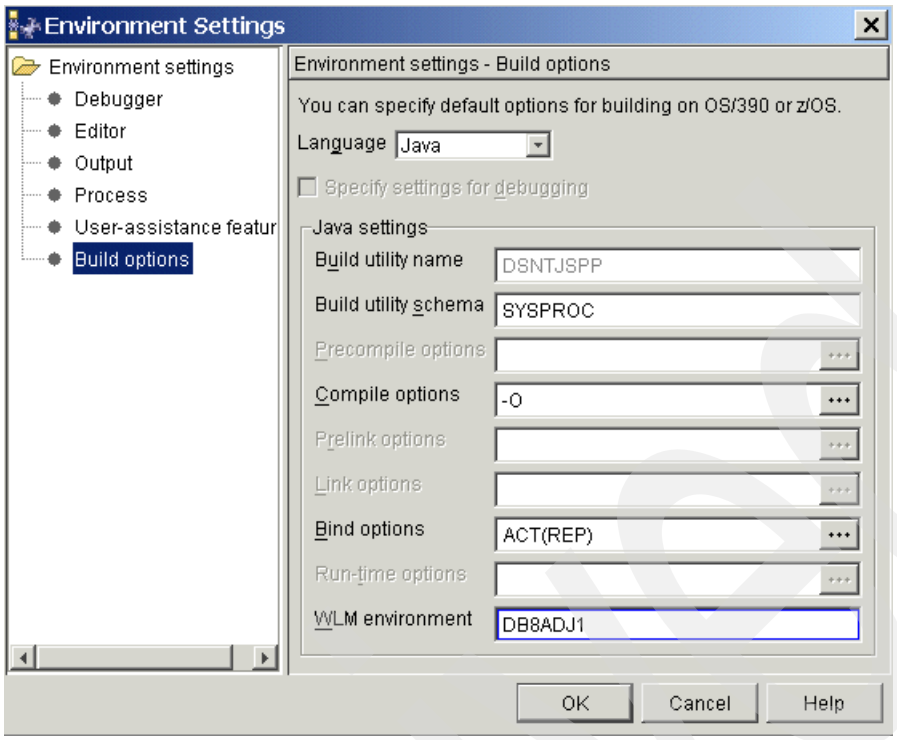


Figure 29-13 Default settings for Java stored procedures

29.4.2 Using SQL Assist

SQL Assist is used by many DB2 UDB client development tools to assist in constructing SQL DML statements. These statements can be very simple, accessing a single table, to very complex joining multiple tables using multiple clauses. SQL Assist is available when creating a new SQL or Java stored procedure using the new object wizard on the SQL Statement panel.

With SQL Assist and some knowledge of SQL, you can create SQL SELECT statements. In some environments, you can also use SQL Assist to create INSERT, UPDATE, or DELETE statements.

The SQL Assist panels look the same for both SQL and Java stored procedures. SQL Assist is launched from the Project view clicking **Database connections-><DB2 Server>->Stored procedure folder->New->Stored procedure using wizard-><SQL> or <Java>->Definition panel->Statement field->**... where **<DB2 Server>** is the CA alias of the remote DB2 server that is being accessed.

The SQL Assist main window is comprised of three areas:

- Outline view
- Details area
- SQL Code view

29.4.3 Arranging the development views

You can customize your development environment by docking or moving and arranging the views, such as the Project View, Output View, Server View, and Editor View. The Development Center remembers the last size, position, and visibility of each view for subsequent invocations. You can do the following:

- Open a view.
- Move a view out of the main window.
- Move a view into the main window.
- Reset views.
- Close a view.

Figure 29-14, shows selecting the **Reset Views** option, which resets the views to the default view layout.

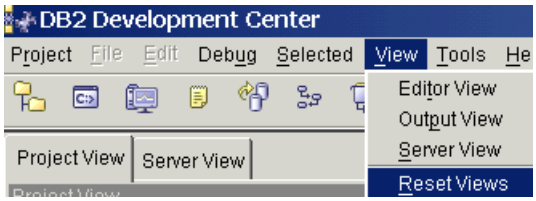


Figure 29-14 Resetting views in Development Center

29.4.4 Create a new stored procedure

Both SQL and Java stored procedures can initially be created using new object wizard or using the Import wizard to import existing source. Additionally, SQL stored procedures can be created using a minimal template to just start coding your procedure, or alternatively copy and paste the source from another file replacing the minimal template.

29.4.5 Creating an SQL stored procedure on z/OS

The following three options are available for creating SQL stored procedures using the Development Center:

- Create a stored procedure from the Editor View panel.
- Import the source of a stored procedure stored from a file.
- Create a stored procedure from the wizard.

With DC started, and project DEVL7083 opened with a database connection to DB8A, we created the following.

Create a stored procedure from the Editor View panel

We did not create any SQL stored procedures using this option, but mention it here for those developers who want to code directly into an editor without using any wizards. In Project View, positioned on the DB8A Stored Procedures folder, right-click **New ->SQL Stored Procedure,** which opens up the Editor View with a small template for coding an SQL stored procedure. When you are finished, close out the Editor View by clicking the **X** to close the window in the upper right hand corner. With the editor closed, we are positioned on the stored procedure in the Project View where we can click the **wrench** icon from the tool bar, or select the stored procedure and right-click **build** to start the build process. Alternatively, clicking the editor **wrench** will prompt you to save your changes and start the build process. This option is only available when creating SQL stored procedures.

Import the source of a stored procedure from a file

We used the Import wizard to copy the SQL stored procedure, EMPDTLSS from DB2G to our DB8A and built the SQL stored procedure for debugging. See our example of the Import wizard for an SQL stored procedure in section “Using the Import wizard” on page 470.

Create a new stored procedure from the wizard

We created a sample stored procedure from the wizard to test our system configuration. See “Create an object” on page 529.

29.4.6 Creating a Java stored procedure on z/OS

The following two options are available for creating Java stored procedures using the Development Center:

- Import the source or an existing stored procedure
- Create a stored procedure from the wizard

With DC started, and project DEVL7083 opened with a database connection to DB8A, we created the following

Import the source or an existing stored procedure

We imported the source for EmpDtlsJ into Development Center from 17.8.1, “Sample Java stored procedure code: EmpDtlsJ using JDBC” on page 268. We also imported the SQLJ stored procedure, using similar steps to the following.

The Import wizard includes the following six panels to be completed. The wizard parses the imported source to complete the panels when possible:

- Source File

  On this panel, we clicked the **Browse** button 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 Development Center panel, 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. DC selects that static method.

- Parameters

  On this panel, one input parameter is listed, and six output parameters are listed.

- Stored Procedure Name

  We enteedr the name EmpDtlsJ on this panel.

- Options

  On the options panel we entered `COLLID` of DEVL7083. The COLLID we choose needs to include the DSNJDBCx drivers. Next, we selected the **Advanced** button on this page, which launches the z/OS Options panel. We entered `DB8ADJ1` for the WLM environment name and clicked **OK**.

- Summary

  The summary panel summarizes the above settings. Optionally, we can view the DDL for the procedure definition for the DB2 catalog by clicking the **Show SQL** button.

We clicked **Finish** to build the stored procedure.

Once the build is completed, you can open the stored procedure in the editor.

Create a stored procedure from the wizard

We created a Java SQLJ stored procedure using the wizard in this section.

Creating a Java stored procedure from the new stored procedure wizard has comparable panels to those used when creating SQL stored procedures. In Project View positioned on the DB8A Stored Procedures folder right-click **New -> Stored Procedure** using Wizard. This launches the New Object panel where we selected **Stored Procedure** in the left-hand panel and **Java** in the right-hand panel, then clicked **OK**.

Five panels are returned to complete as needed to create the required Java stored procedures:

- Name

  We entered `DEVL7083.JAVATEST` for our name.

- Definition

  The definition panel includes seven optional fields, each has a default:

  - Statement

    On the **SQL Statement** panel, you can determine how many SQL statements are generated. Only one option can be chosen:

    - Generate one SQL statement.
    - Generate multiple SQL statements.
    - Generate no SQL statements.

    The default selection is Generate one SQL statement.

  - Result Set

    - One
    - None
    - Multiple

  - Errors

    - SQLSTATE
    - SQLCODE
    - SQLSTATE and SQLCODE
    - SQL Exception
    - SQL Message
    - SQLSTATE and SQL Message
    - SQLCODE and SQL Message
    - SQLSTATE, SQLCODE and SQL Message

  - Header fragments

    - Include any file with header information you want included in this stored procedure. thxThis 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.

  We took defaults for the definition. A default SQL Statement for testing setup is included in the SQL statement on the Definition panel. The statement is SELECT SCHEMA, NAME

FROM SYSIBM.SYSROUTINES. We also included a file for each fragment entry. See 29.5.1, “Using code fragments” on page 538 for more information.

- Parameter

  We took defaults, no parameters were defined.

- Options

  The Jar ID is automatically filled in with a random value. We override it with `DEVL7083.JAVATEST` our schemaname.procname.

  The Collection ID that is specified must include the JDBC drivers on z/OS. The JDBC drivers are bound into DSNJDBC using <hlq>.SDSNSAMP(DSNTJJCL). The value that is specified here is included on the BIND PACKAGE(collid).

  We built this stored procedure using SQLJ and checked **Database access->Static SQL** using SQLJ check box.

  The **Verbose build** check box should be used initially to verify the set up on z/OS. This option returns information to the Development Center Output View, Messages panel regarding the build process being performed by DSNTJSPP and DSNTBIND (SQLJ stored procedures only) on z/OS. Incorrect or missing settings in the properties files used by DSNTJSPP during the build process are the types of errors returned to the Messages panel when this option is checked.

  Since we updated our default WLM environment for executing Java stored procedures to point to our DB8A WLM AE, DB8ADJ1, we did not have to make any updates z/OS Options through the Advanced button.

- Summary

  This panel summarizes our Java stored procedure. The generated code for this example is shown in Example 29-17.

Example 29-17 Example of generated SQLJ code using fragments

```
/**
 * SQLJ Stored Procedure DEVL7083.JAVATEST
 */
/**
 * Header fragment inserted from SP_JAVA_HDR.FRAGMENT
 */
package PKG3113011336980;
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 ( String, String );
public class JAVATEST
{
    /**
     * Data  fragment inserted from SP_JAVA_DATA.FRAGMENT
     */

    public static void jAVATEST ( ResultSet[] rs1 ) throws SQLException, 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();
        }
        catch (SQLException e)
        {

            // Close open resources
            try
            {
                if (cursor1 != null) cursor1.close();
                if (ctx != null) ctx.close();
            } catch (SQLException e2) { /* ignore */ };
            throw e;
        }
    }

    /**
     * Methods fragment inserted from SP_JAVA_MTHD.FRAGMENT
     */
}
```

29.5 Advanced Development Center topics

This section includes information on the following advanced topics:

- Using code fragments
- Generating multiple SQL statements with a single result set
- Generating multiple SQL statements with multiple result sets
- Using DC to copy from one server, and paste and build on another server
- Deploying SQL or Java stored procedures without recompiling

29.5.1 Using code fragments

Development Center wizards for creating a new SQL or Java stored procedure include the option to include code fragments on the Definition panel of the wizard.

For SQL stored procedures, these fragments relate to:

- 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 exceptions 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:

- 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.

29.5.2 Generating multiple SQL statements with a single result set

The SQL wizard can generate multiple SQL statements with a single result set. When creating new SQL or Java stored procedures using the new wizard, the **Definition panel ->Statement panel** allows the option for Development Center to generate multiple SQL statements. On the SQL Statement panel, click the **Generate Multiple SQL statements** radio button. Selecting the **Add** button in the right-hand side of the panel generates another statement to replace with your SQL statement.

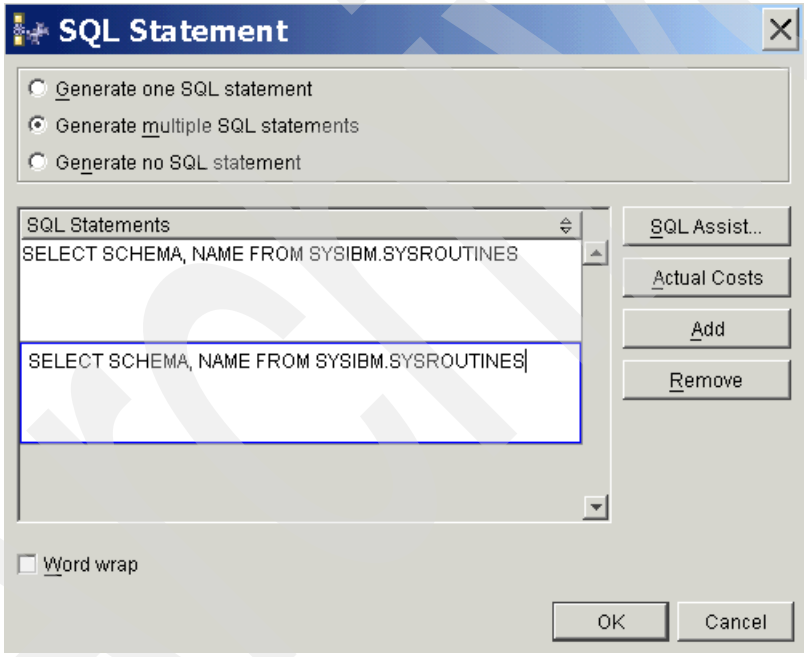


Figure 29-15 Generate multiple SQL statements

This causes a CASE statement to be generated in the code, as shown in Example 29-18. Our example generated the following.

Example 29-18 Java stored procedure with multiple SQL statements and a single result

```
/**
 * JDBC Stored Procedure DEVL7083.JAVATEST2
 */
/**
 * Header fragment inserted from SP_JAVA_HDR.FRAGMENT
 */
package PKG31123063024400;
```

```
import java.sql.*;                    // JDBC classes
/**
 * Imports fragment inserted from SP_JAVA_IMPORT.FRAGMENT
 */

public class JAVATEST2
{
    /**
     * Data fragment inserted from SP_JAVA.DATA.FRAGMENT
     */

    public static void jAVATEST2 ( 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 SCHEMA, NAME FROM SYSIBM.SYSROUTINES";
                stmt = con.prepareStatement( sql );
                break;
            case 1:
                sql = "SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES";
                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
     */
}
```

29.5.3 Generating multiple SQL statements with multiple result sets

The SQL wizard can generate multiple SQL statements with multiple result sets. The following example repeats the definition in Example 29-18 but this time we select multiple result sets instead of a single result set. This is done from the **Definition panel -> Result set -> Multiple**.

The generated code using SQL for the language is shown in Example 29-19.

Example 29-19 SQL stored procedure with multiple SQL statements and a multiple result sets

```
CREATE PROCEDURE DEVL7083.SQLTEST2 ( IN whichQuery INTEGER )
    RESULT SETS 2
    LANGUAGE SQL
```

```
    COLLID DEVL7083
    WLM ENVIRONMENT DB8ADS1
    RUN OPTIONS 'NOTEST(NONE,*,*,*)'
-------------------------------------------------------------------------
-- SQL Stored Procedure
-------------------------------------------------------------------------
P1: BEGIN
    -- Declare cursors
    DECLARE cursor1 CURSOR WITH RETURN FOR
        SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES;
    DECLARE cursor2 CURSOR WITH RETURN FOR
        SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES;

    -- Cursor left open for client application
    OPEN cursor1;
    -- Cursor left open for client application
    OPEN cursor2;
END P1
```

29.5.4 Using DC to copy from one server and paste/build on another server

Development Center can be used to copy an SQL or Java stored procedure from one server and then paste, modify as needed, and 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 and paste requires having an open project with database connections to both the source and target server.

In our case (shown in Figure 29-16) we have DC started with connections to both DB8A and DB2G. From Project View, we selected the **Stored Procedure** folder for our source database connection (DB8A), then the stored procedure **DEVL7083.JAVATEST** that we wanted to copy. We then right-clicked **Copy.**

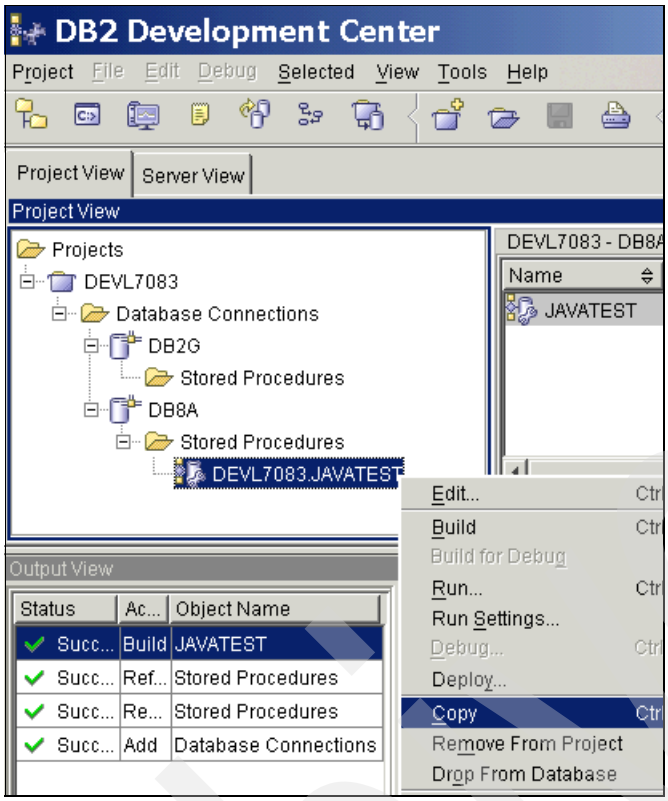


Figure 29-16 Using DC to copy a Java or SQL stored procedure to another server

Next, we selected the **Stored Procedure** folder under the database connection for our target server (DB2G) and right-clicked **Paste**. The only property change we need to make is for the WLM AE. We made this change before we built the stored procedure by selecting **DEVL7083.JAVATEST->Properties->Options** and changed the WLM Environment from DB8ADJ1 to DB2GDEJ1. Next, we clicked the **wrench icon** from the tool bar to build the stored procedure on DB2G. We used the default COLLID of DSNJDBC for this stored procedure on both the DB8A and DB2G servers. The output of this is shown in Figure 29-17.

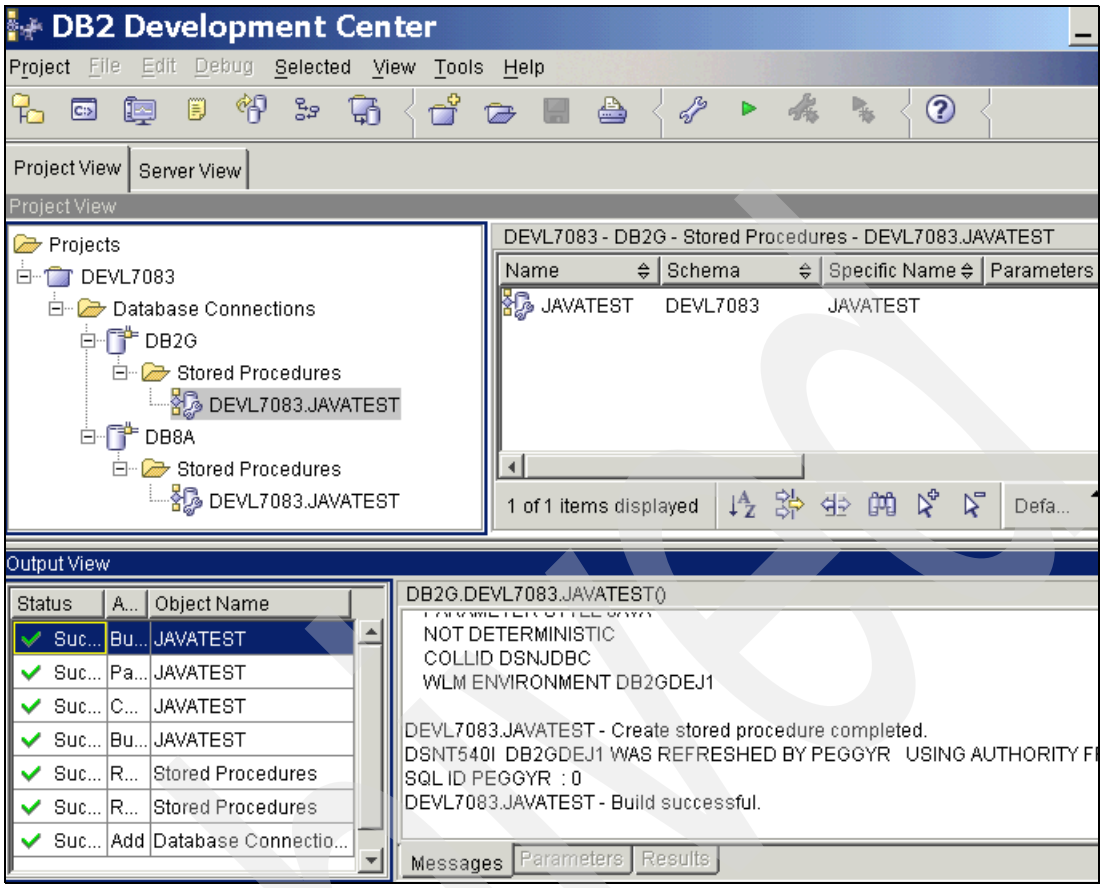


Figure 29-17 DC copy, paste, modify stored procedure and build output

29.5.5 Deploying SQL or Java stored procedures without recompiling

The Development Center includes a Deployment Tool and a Deployment Wizard to deploy SQL or Java stored procedures between source and target servers on the same platform. Using the Deployment Tool or the Deployment Wizard builds the stored procedure on the target server.

Some customers want to migrate compiled code and not rebuild the stored procedure on the target server.

SQL stored procedures

Deploying SQL stored procedures without rebuilding requires performing the following steps on the target server:

- Registering the stored procedure

  Before registering the DDL, you may need to update the COLLID, WLM AE name, and the SCHEMA

- Copy the DBRMs from the source server and bind the DBRMs in the target server
- Copy the compiled SQL module into the production data set
- Grant execute authorization on the stored procedure

See 16.3.1, “Compile just once” on page 231 for more details.

Java stored procedures

Development Center installs the compiled Java class bytes in the DB2 catalog as a BLOB. We modified the sample code from Example 18-13 on page 289, and created a stored procedure to extract the Java *.class, *.ctx (SQLJ only) and *.ser (SQLJ only) files. We performed the following steps to migrate our DEVL7083.SQLJTEST stored procedure from DB2G to another DB2 for OS/390 V7 server, DB7P:

1. Create and compile the ExtractJarSp.java code on USS.
2. Register the DDL for the ExtractJarSp in the DB2 catalog.
3. Update the _CEE_ENVFILE pointed to by our JAVAENV DD for the WLM proc where the ExtractJarSp executes:
   - Add a JVMPROPS entry.
   - Update CLASSPATH with the HFS directory where ExtractJarSp was compiled.
4. Run the ExtractJarSp using Development Center.
5. FTP the jar file created by ExtractJarSp from the source server to the target server.
6. Use Development Center to install the jar file in the target server using the DB2-supplied SQLJ.INSTALL_JAR stored procedure.
7. Copy the DBRMs and perform the DBRM BIND.
8. Create the DDL for the stored procedure and register the DDL in the DB2 catalog on the target DB2 server.
9. Grant execute authorization and run the migrated stored procedure.

This example works on DB2 V7 servers only when no changes are required to the Java code being migrated:

1. Create the ExtractJarSp.java code on USS

   We created the code listed in Example 29-20 on our DB2G, /SC63/sg247083/spjava directory, and compiled it using the javac program. No error handling code has been written for this stored procedure.

Example 29-20 ExtractJarSp code to extract

```
/*   DDL for registering stored procedure
* CREATE PROCEDURE DEVL7083.EXTRACTJARSP (IN SCHEMANAME     CHAR(8),
*                                         IN PROCNAME       CHAR(18),
*                                         IN FILENAME       VARCHAR(200),
*                                         OUT OUTPUTMESSAGE  VARCHAR(250))
*   EXTERNAL NAME 'ExtractJarSp.GetJarFile'
*   LANGUAGE JAVA
*   PARAMETER STYLE JAVA
*   COLLID DSNJDBC
*   PROGRAM TYPE SUB
*   WLM ENVIRONMENT DB2GDEJ1 */

import java.sql.*;
import java.io.*;

public class ExtractJarSp {

 public static void GetJarFile(String Schema, String Procname, String Filename, String[]
outmsg) {
  String owner;
    Blob jarBlob = null;
    String sql = null;
    System.out.println("Schema Name is " + Schema);
```

```
    System.out.println("Procname      is " + Procname);
    System.out.println("Filename    is " + Filename);
    String jarData;
    String jarID;
    String sqltxt;
    InputStream inpStream = null;
    int nread;
    byte[] byteArray = new  byte[1024];

    try {
     FileOutputStream outFile = new FileOutputStream(Filename);
     Connection con =
      DriverManager.getConnection("jdbc:default:connection");
    Statement stmt = con.createStatement();
                    sqltxt = "SELECT JAR_DATA FROM SYSIBM.SYSJAROBJECTS J INNER JOIN
SYSIBM.SYSROUTINES R  ON   J.JARSCHEMA = R.JARSCHEMA AND  J.JAR_ID = R.JAR_ID  WHERE SCHEMA
= "
                           + "'" + Schema + "'"
                           + "and  NAME   = '" + Procname + "'" ;
     ResultSet rs =
      stmt.executeQuery(sqltxt);
     if (rs.next())
     {
      jarBlob = rs.getBlob("JAR_DATA") ;
   if (jarBlob != null)
      inpStream = jarBlob.getBinaryStream() ;
     }

   if (inpStream != null)
               { 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) {
     outmsg[0]=
      "SQLException raised, SQLState = "
       + e.getSQLState()
       + " SQLCODE = "
       + e.getErrorCode()
       + " :"
       + e.getMessage();
   System.out.println(outmsg[0]);
    } catch (Exception e) {
      e.printStackTrace();
     System.out.println("Error found" + e.toString());

    }
 }
}
```

2. Compile the ExtractJarSp from our USS command prompt as shown in Example 29-21.

Example 29-21 Compile ExtractJarSp.java

```
PEGGYR @ SC63:/u/peggyr>cd /SC63/sg247083/spjava
PEGGYR @ SC63:/SC63/sg247083/spjava>javac ExtractJarSp.java
PEGGYR @ SC63:/SC63/sg247083/spjava>ls ExtractJarSp.*
```

```
ExtractJarSp.class  ExtractJarSp.java
```

3. Register the DDL for the ExtractJarSp in the DB2 catalog.

   This registration occurs on the source DB2 server, which is our DB2G system. You will need to modify the SCHEMA and WLM ENVIRONMENT name as shown in our Example 29-22 before registering the DDL for your source DB2 server.

Example 29-22 CREATE PROCEDURE DDL for ExtractJarSp

```
CREATE PROCEDURE DEVL7083.EXTRACTJARSP (IN SCHEMANAME     CHAR(8),
                                        IN PROCNAME       CHAR(18),
                                        IN FILENAME       VARCHAR(200),
                                        OUT OUTPUTMESSAGE VARCHAR(250))
   EXTERNAL NAME 'ExtractJarSp.GetJarFile'
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    COLLID DSNJDBC
    PROGRAM TYPE SUB
    WLM ENVIRONMENT DB2GDEJ1
```

4. Update the _CEE_ENVFILE pointed to by our JAVAENV DD for the WLM proc where the ExtractJarSp executes.

   Our JAVAENV DD size exceeded 245 bytes so we used the _CEE_ENVFILE environment variable to point to the HFS file containing most of the environment variables used:

   – We added a CLASSPATH environment variable to point to the HFS directory where we compiled our ExtractJarSp file.

   – We then added a JVMPROPS entry to point to an HFS data set, jvmprops.properties, with our increased Java memory settings needed to execute our ExtractJarSp.

   These two entries are shown in example Example 29-23 and Example 29-24.

Example 29-23 JAVAENV file updates for CLASSPATH and JVMPROPS environment variables

```
CLASSPATH=/SC63/sg247083/spjava
..
JVMPROPS=/SC63/sg247083/DB2GU/jvmprops.properties
```

Example 29-24 JVMPROPS file /SC63/sg247083/DB2GU/jvmprops.properties

```
-Xms64M
-Xmx128M
```

5. Run the ExtractJarSp using Development Center.

   The ExtractJarSp can be run using Development Center on the workstation, or from a UNIX command prompt on the source DB2 server. We used Development Center in Server View. We selected the **ExtractJarSp** and right-clicked **Run**, then entered the following three parameters:

   – SCHEMANAME - The schema to be migrated
   – PROCNAME - The procname to be migrated
   – FILENAME - The file containing the extracted Jar on USS

   This is shown in Figure 29-18.

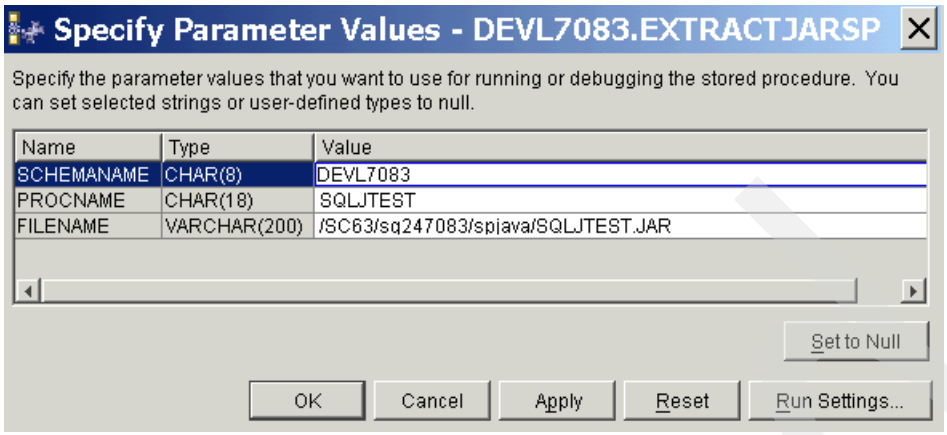


Figure 29-18 Run ExtractJarSp on DB2G

6. FTP the jar file created by ExtractJarSp from the source server to the target server

   We used the FTP to copy the Jar, /SC63/sg247083/spjava/SQLJTEST.JAR, created by ExtractJarSp on the source server to our target server into the HFS file, /u/peggyr/SQLJTEST.JAR. This information will be used in the URL parameter when we call SQLJ.INSTALL_JAR.

7. Use Development Center to install the jar file in the target server using the DB2-supplied SQLJ.INSTALL_JAR stored procedure.

   We used Development Center to install the jar file in the target server using the DB2-supplied SQLJ.INSTALL_JAR. See Figure 29-19.

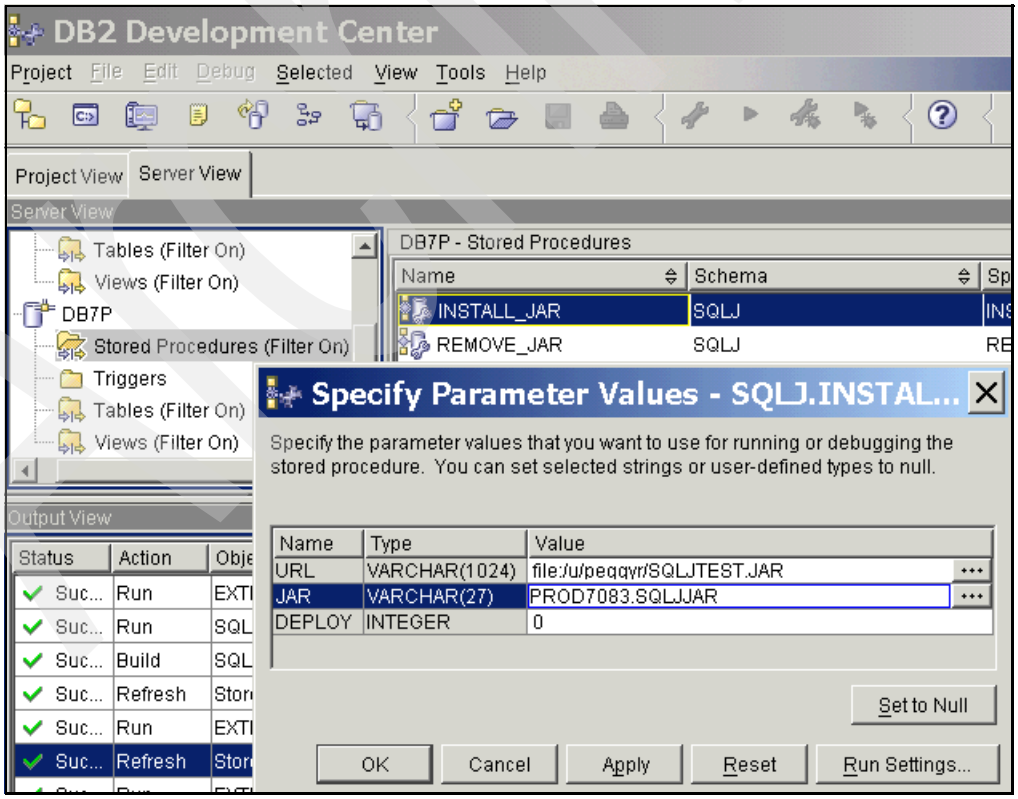


Figure 29-19 Install SQLJTEST.JAR into DB7P DB2 server

The three parameters passed to the INSTALL_JAR stored procedure are described in Table 29-11.

Table 29-11 DB2-supplied INSTALL_JAR stored procedure run on target server

Parameter	Definition	Value entered
URL	A VARCHAR(128) input parameter that identifies the HFS full path name for the JAR file that is to be installed in the DB2 catalog. The format is file://path-name or file:/path-name.	file:/u/peggyr/SQLJTEST.JAR
JAR	A VARCHAR(27) input parameter that contains the DB2 name of the JAR, in the form schema.jar-id or jar-id. This is the name that you use when you refer to the JAR in SQL statements. If you omit schema, DB2 uses the default schema name.	PROD7083.SQLJJAR this value has to be specified as the first 2 parameters of the CREATE PROCEDURE DDL External Name
DEPLOY	An INTEGER input parameter that indicates whether additional actions should be performed after the JAR file is installed. Additional actions are not supported, so this value should always be 0.	0

8. Copy the DBRMs and perform the DBRM BIND

   Copy the DBRMs from DB2G (source server) to DB7P (target server)

   We used the SQL statement in Example 29-25 to determine the DBRMLIB and the DBRM name that needed to be selected from the source.

Example 29-25 SQL statement to determine DBRMs to migrate

```
SELECT "DBRMLIB" , "POBJECT_LIB"
FROM SYSIBM.SYSJAVAOPTS J INNER JOIN
     SYSIBM.SYSROUTINES R
ON   J."JARSCHEMA" = R."JARSCHEMA"
AND  J."JAR_ID"    = R."JAR_ID"
WHERE "SCHEMA" = 'DEVL7083'
AND   "NAME"   = 'SQLJTEST'
```

The output of this SQL statement is shown in Example 29-26.

Example 29-26 DBRMLIB and object to be migrated

```
DBRMLIB                    POBJECT_LIB
DB2V710G.DBRMLIB.DATA      SPB243
```

There are four DBRMs for each POBJECT_LIB name to support the four isolation levels, CS, RR, RS, and UR for our Java stored procedure. Therefore, we need to copy and bind: SPB2431, SPB2432, SPB2433, and SPB2434 from our source server to our target server. Our DB7P target server was on a different sysplex than our DB2G source server. We used FTP to copy these four DBRMs from the source DBRMlib to the target DBRMlib. An example of the bind job on the target server is shown in Example 29-27.

Example 29-27 Bind package sysin for DB7P server

```
//BINDPKG EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB  DD DISP=SHR,DSN=DB7PU.DBRMLIB.DATA
```

```
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *
  DSN SYSTEM(DB7P)

   BIND PACKAGE (DSNJDBC) MEMBER(SPB2431) ISOLATION(UR)   -
     ACTION(REPLACE)  VALIDATE(BIND)
   BIND PACKAGE (DSNJDBC) MEMBER(SPB2432) ISOLATION(CS)   -
     ACTION(REPLACE)  VALIDATE(BIND)
   BIND PACKAGE (DSNJDBC) MEMBER(SPB2433) ISOLATION(RR)   -
     ACTION(REPLACE)  VALIDATE(BIND)
   BIND PACKAGE (DSNJDBC) MEMBER(SPB2434) ISOLATION(RS)   -
     ACTION(REPLACE)  VALIDATE(BIND)
END
//
```

9. Create the DDL for the stored procedure and register the DDL in the DB2 catalog on the target DB2 server.

   We used Development Center to generate the DDL used in our test environment, then modified the WLM AE, SCHEMA and the first two parameters of the EXTERNAL NAME for our production system. We obtained the DDL from DB2G by starting Development Center in Project View, selecting the **DEVL7083.SQLJTEST** stored procedure on DB2G that is being migrated. Next, we right-clicked **Show SQL**. The DDL was then copied to our workstation. We changed the WLM AE to DB7PJAVA the SCHEMA to PROD7083 and the first two parts, (SCHEMA and JARID) of the EXTERNAL NAME to PROD7083.SQLJJAR. This is summarized in Table 29-12.

Table 29-12 Source and target DB2 server CREATE PROCEDURE DDL

Source - DB2G	CREATE PROCEDURE DEVL7083.SQLJTEST ( ) RESULT SETS 1 EXTERNAL NAME 'PEGGYR.SQL40018072519570:PKG40018072527490.SQLJTEST.sQLJTEST' LANGUAGE JAVA PARAMETER STYLE JAVA NOT DETERMINISTIC COLLID DSNJDBC WLM ENVIRONMENT DB2GDEJ1
Target - DB7P	CREATE PROCEDURE **PROD7083.SQLJTEST** ( ) RESULT SETS 1 EXTERNAL NAME '**PROD7083.SQLJJAR**:PKG40018072527490.SQLJTEST.sQLJTEST' LANGUAGE JAVA PARAMETER STYLE JAVA NOT DETERMINISTIC COLLID DSNJDBC WLM ENVIRONMENT **DB7PJAVA**

   Once we had our production DDL created, we installed that DDL on our target server using the DB2 DEMO workstation GUI tool. This tool is a no charge download from:

   http://www.ibm.com/developerworks/db2/library/demos/db2demo/index.html

   Alternatively, SPUFI or host batch programs such as <hlq>.SDSNSAMP(DSNTIAD) or (DSNTEP2) can also be used. The DDL can also be created by obtaining the appropriate columns from SYSIBM.SYSROUTINES and SYSIBM.SYSPARMS from the source server.

10. Grant execute authorization and run the migrated stored procedure.

Our last step is to GRANT EXECUTE TO PUBLIC for our PROD7083.SQLJTEST stored procedure on DB7P, and use Development Center to test the execution and successful migration.

29.6 Future Development Center enhancements

In this section we describe some of the Development Center enhancements to be delivered in the DB2 Stinger release. Stinger is the code name for the new release of DB2 for Multiplatform of which we keep hearing about. These enhancements include support of the Universal JDBC driver. This support will allow building using either the Legacy JDBC driver or the Universal JDBC driver. Depending which driver is being used, an appropriate WLM environment needs to be selected for executing the Java stored procedure that is created. The difference in the two WLM environments is in the JAVAENV DD. The Legacy JDBC driver requires using DB2_HOME and the Universal JDBC driver requires using JCC_HOME.

Important: The DB2 Stinger Development Center planned enhancements are subject to change, this is not a commitment for inclusion in this product release.

The default JDBC driver that is used by Development Center is the Universal JDBC driver. Figure 29-20 is an early example of the Java Environment Settings in Stinger.

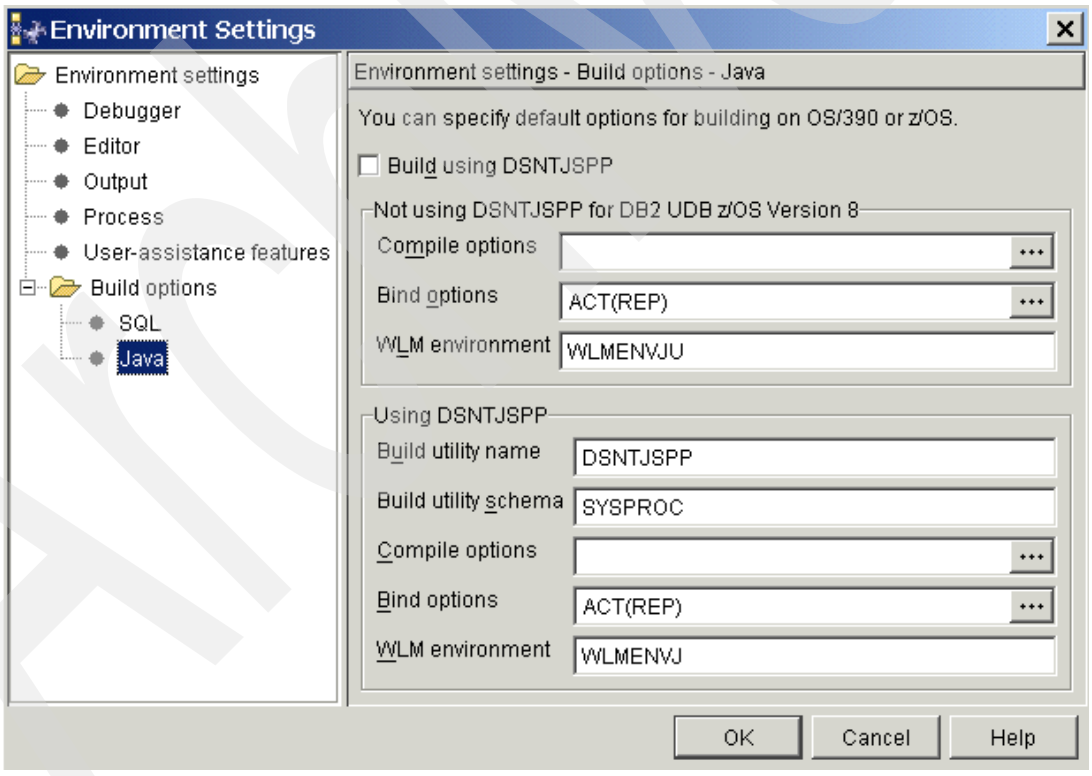


Figure 29-20 Java environment settings

The Development Center support for the Universal JDBC driver can be used for migrating existing Java stored procedures using the Legacy JDBC driver to the new Universal JDBC driver on the same system, or in migration to a new DB2 server. These enhancements apply when connecting to a DB2 for z/OS V8. The steps needed to change drivers follow:

- Create a new or open an existing project.

- Add a database connection to a DB2 for z/OS V8 server.
- Select an existing Java stored procedure from Server View, and add to the project if not already in the open project.
- Select the Java stored procedure to migrate to the Universal JDBC driver, right-click **Properties -> Build -> uncheck Build with DSNTJSPP**. Also, change the WLM environment to a WLM environment where the JAVAENV DD statement points to the JCC_HOME environment variable.
- Close out the properties.
- Rebuild the Java stored procedure.

The reverse procedure can be performed to rebuild the Java stored procedure using the Legacy JDBC driver. This process may be used during migration and testing for the Java stored procedures between the two JDBC drivers.

Other enhancements for Development Center include:

- Increasing the size of JDBC and SQLJ stored procedures from 32 KB to 10 MB when using the Universal JDBC driver
- Support for creating Java stored procedures on iSeries

30

Using WSAD to debug Java stored procedures converted to Java applications

Currently, no IBM debug product exists for debugging interpreted Java stored procedures on z/OS. One alternative is to convert the Java stored procedure to a Java application, and use the debugger included in the following WebSphere Studio products:

- WebSphere Studio Site Developer (WSSD)
- WebSphere Studio Application Developer (WSAD)
- WebSphere Studio Application Developer Integration Editor (WSADIE)
- WebSphere Studio Enterprise Developer (WSED)

Additionally, the DB2 Application Development Client (ADC) for DB2 UDB V8.1 or DB2 V7.2 is required.

In this chapter we describe a possible solution using WSAD to debug Java stored procedures converted to Java programs. Our case study uses WebSphere Studio Application Developer V5.1. This study assumes that WSAD and the DB2 ADC are installed on the workstation, but no additional configuration prerequisites have been implemented. Little knowledge of WSAD is required to use the debugger for Java applications. We consider both JDBC and SQLJ stored procedures.

This chapter contains the following:

- Debugging JDBC SPs converted to JDBC applications
- Debugging SQLJ SPs converted to SQLJ applications

30.1 Debugging JDBC SPs converted to JDBC applications

This section describes using WSAD to debug a JDBC application that is converted from a JDBC stored procedure on DB2 for z/OS.

Launch WSAD by selecting **Start-> Programs-> IBM WebSphere Studio-> Application Developer 5.1**. The first window that is returned is the WebSphere Studio window that sets a working directory for this project, called a *workspace*. Optionally, if you have previously used WSAD, you may have set a *default workspace* for all projects, in which case this window will not appear, and your *default workspace* will be opened. See Figure 30-1.

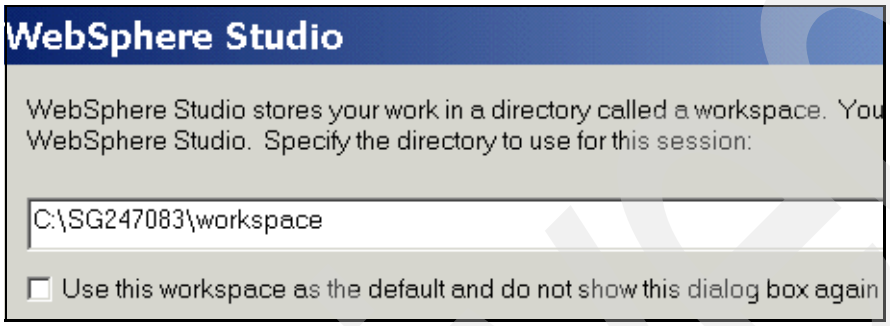


Figure 30-1 WebSphere Studio workspace specification

Since we created a new workspace c:\sg247083\workspace, no other projects or objects exist in this working directory when WSAD is first started. The next window that appears is the WSAD Welcome window. The Welcome window is opened in the J2EE Perspective, which is our default Perspective chosen upon installation of WSAD. The default Perspective can be changed after starting WSAD by selecting the **title bar Window -> Preferences -> Workbench -> Perspectives**. Close out the Welcome window in Figure 30-2 by clicking the **X** in the Welcome window title bar.

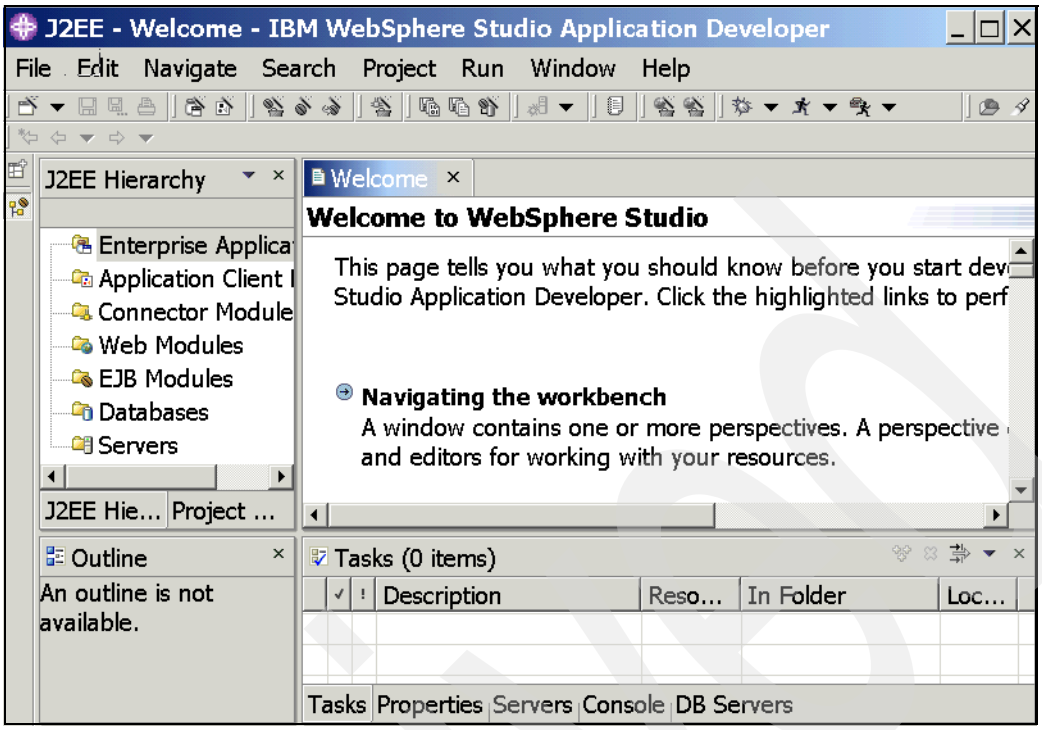


Figure 30-2 WSAD - Welcome window

We want to create a Java project to run our test cases in, and do this by selecting **File -> New -> Other** as shown in Figure 30-3.

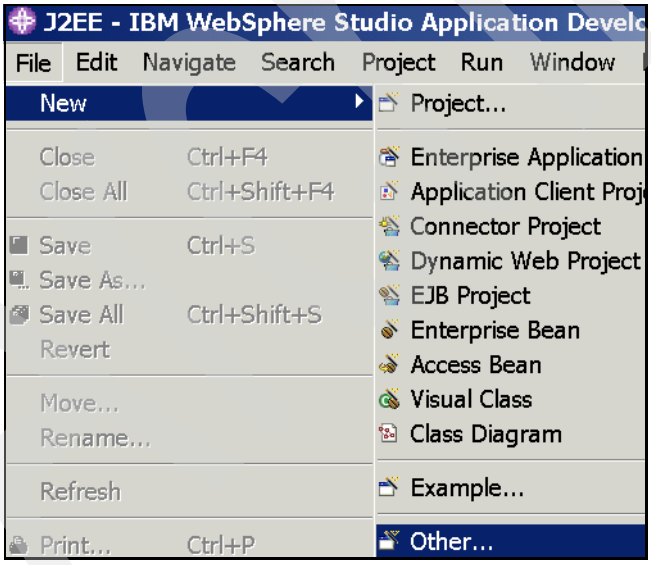


Figure 30-3 Creating a New Project

From the New Project window, select **Java** in the left hand window, and **Java Project** in the right hand window as shown in Figure 30-4, and click **Next.**

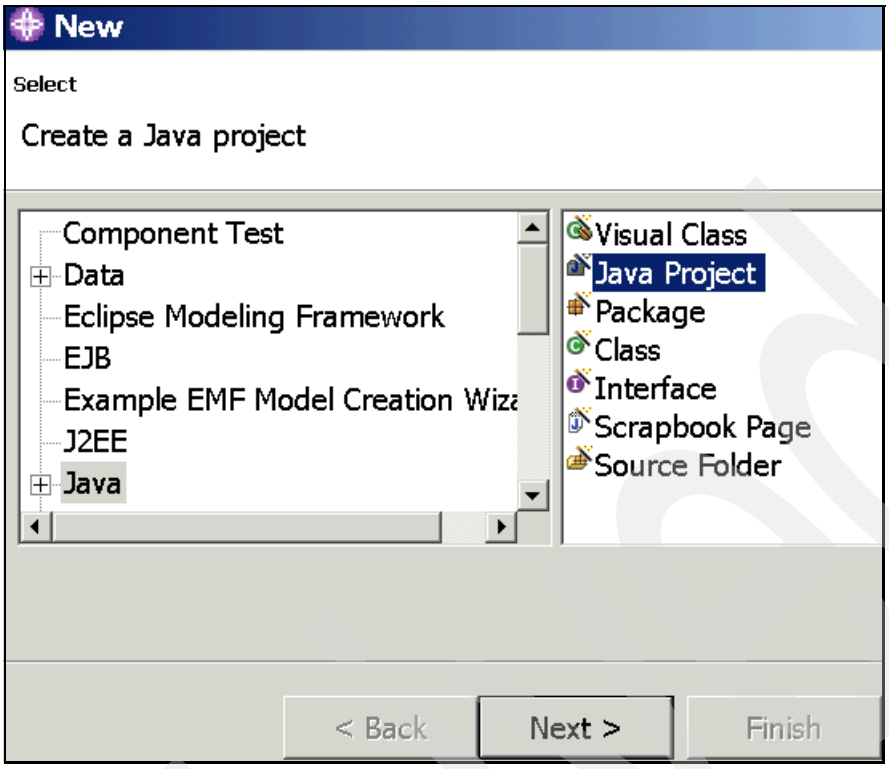


Figure 30-4 Create a new Java Project

On the New Java Project window, enter the Project name of `JDBCSPDEBUG`. Leave the **Project contents** check box **Use defaults** checked as shown in Figure 30-5. Click **Next** to continue.

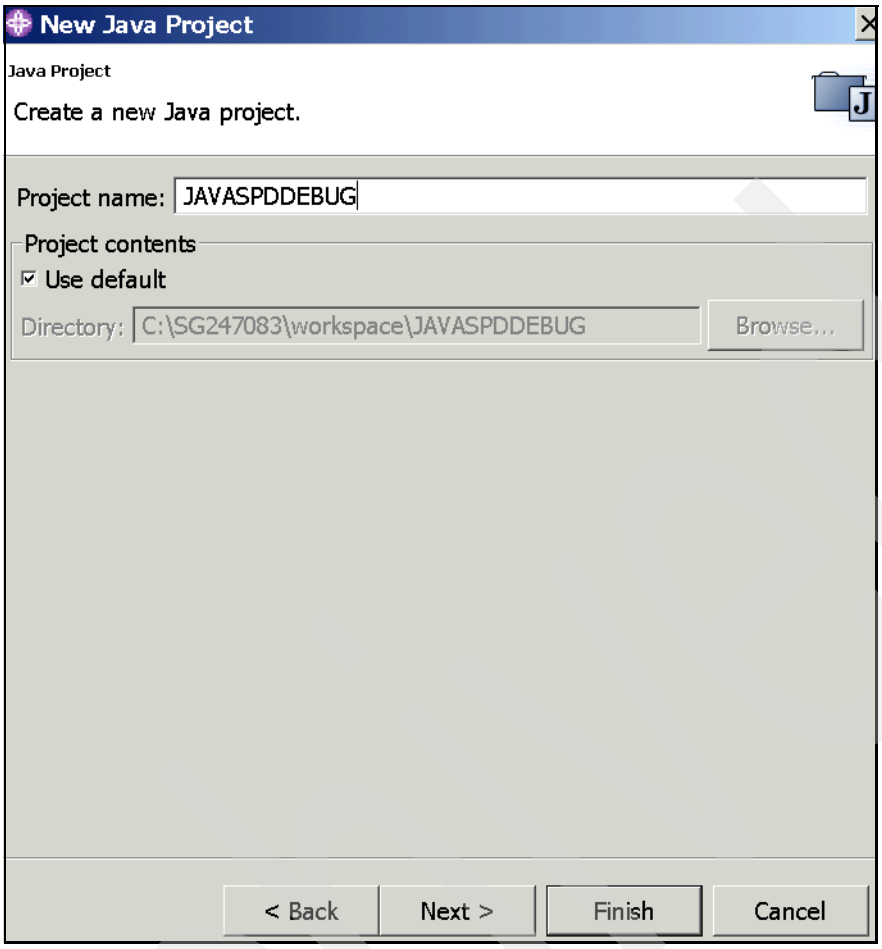


Figure 30-5 Java Project definition

The Java Settings window is returned as shown in Figure 30-6. Click the **Libraries** tab.

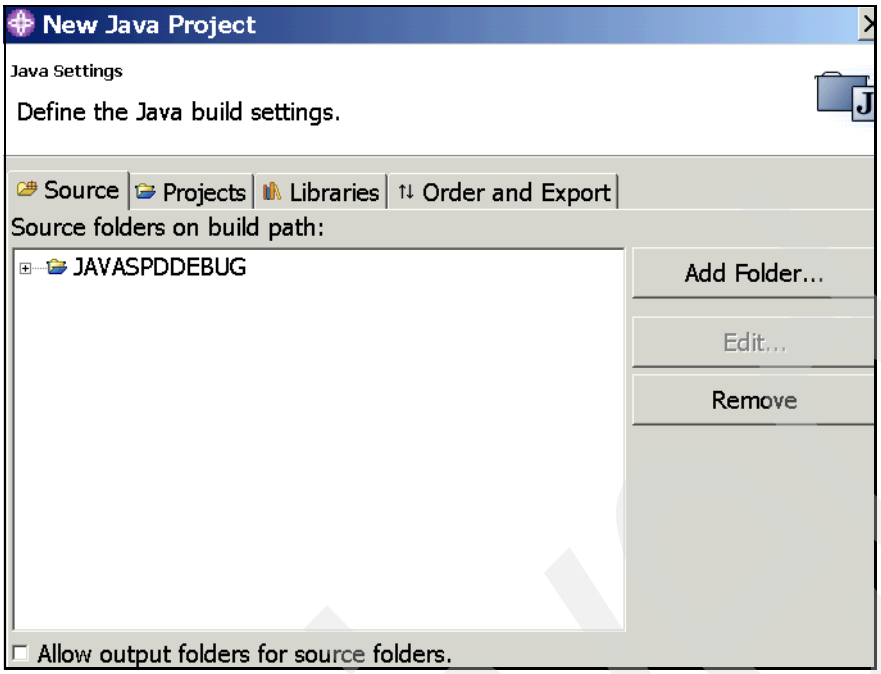


Figure 30-6 Define the Java build settings - Source

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 the **Add External JARS...** button on the right hand side of the window as shown in Figure 30-7.

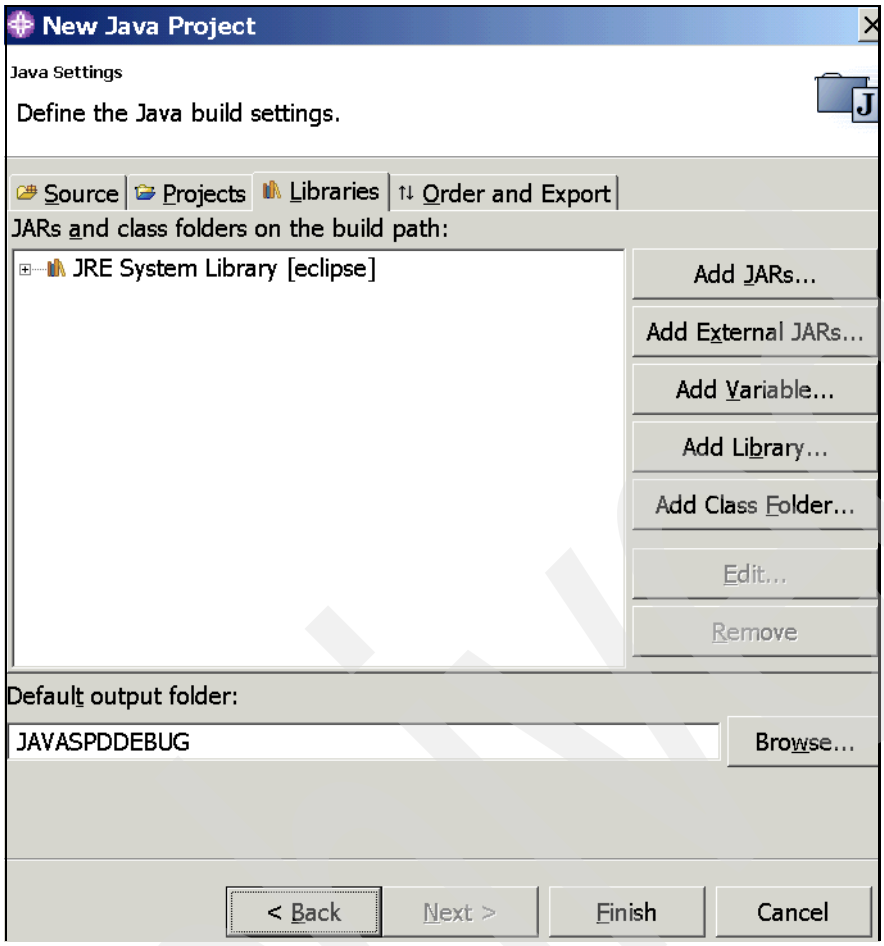


Figure 30-7 Define the Java build settings - Libraries

Browse the workstation folder that includes:

- db2java.zip
- db2jcc_license_cisuz.jar
- db2jcc.jar

By default, these are stored in the ..\SQLLIB\java directory where the DB2 ADC is installed.

The Libraries window now includes the files shown in Figure 30-8. Click the **Finish** button at the bottom of the window, and the template for our Java application will be generated.

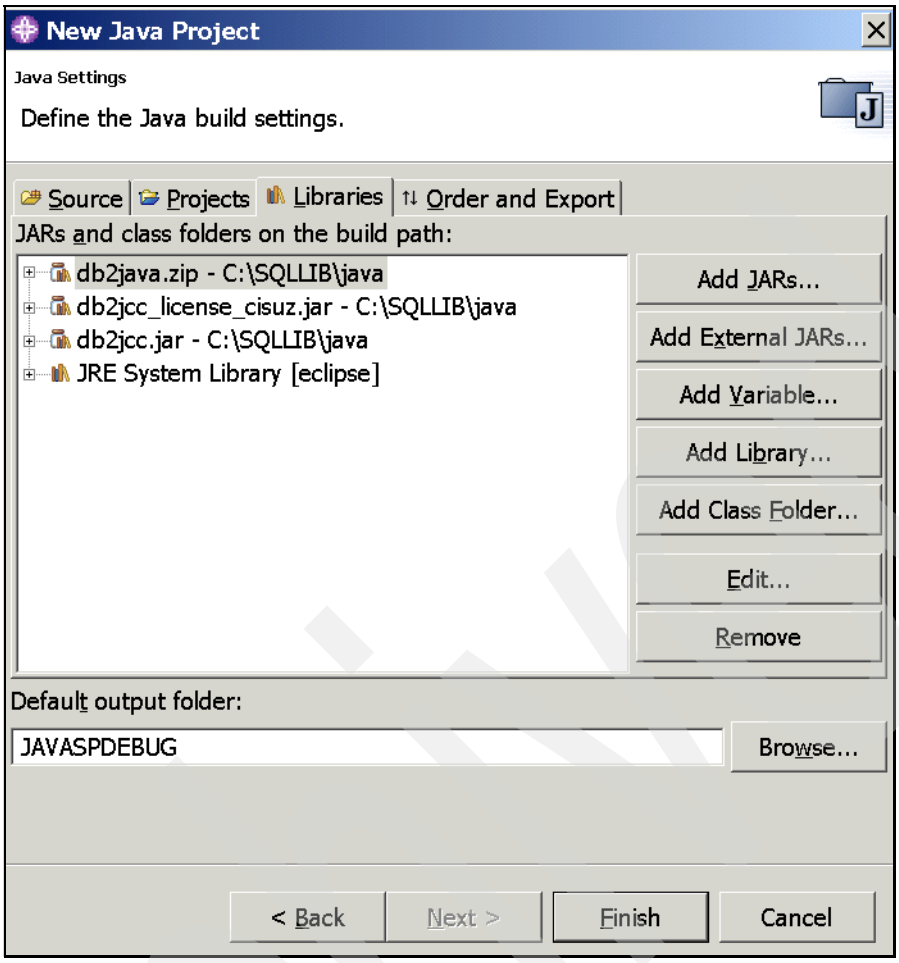


Figure 30-8 Java build settings, selected external jars

Since our WSAD default Perspective was the J2EE Perspective, and we are creating a Java Project that is associated with the Java Perspective, the window in Figure 30-9 appears and asks if we want to switch to the Java Perspective. Click **Yes** in the window as shown.

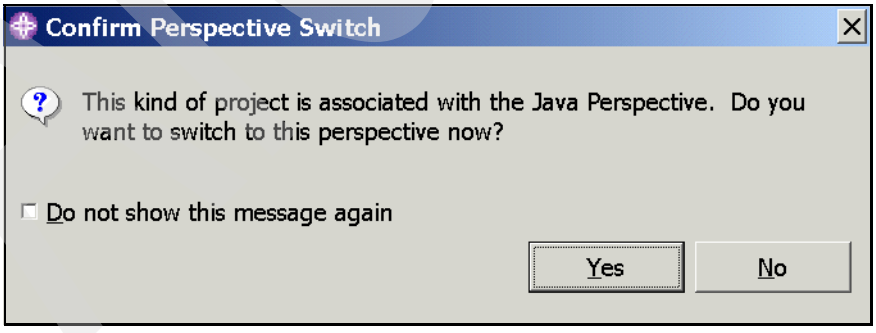


Figure 30-9 Confirm Perspective Switch

The Java Perspective is now returned, which is identified by the title bar Java - IBM WebSphere Studio Application Developer. From the Package Explorer view in the upper left hand portion of the window, select the **JAVASPDEBUG** folder and right-click and select **Import** as shown in Figure 30-10.

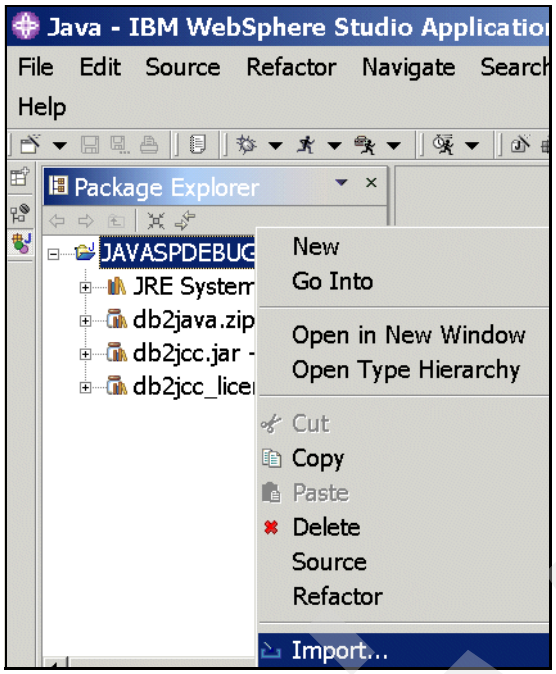


Figure 30-10 Java Perspective, Package Explorer view

From the Import window, select **File system.** Click **Next** as shown in Figure 30-11.

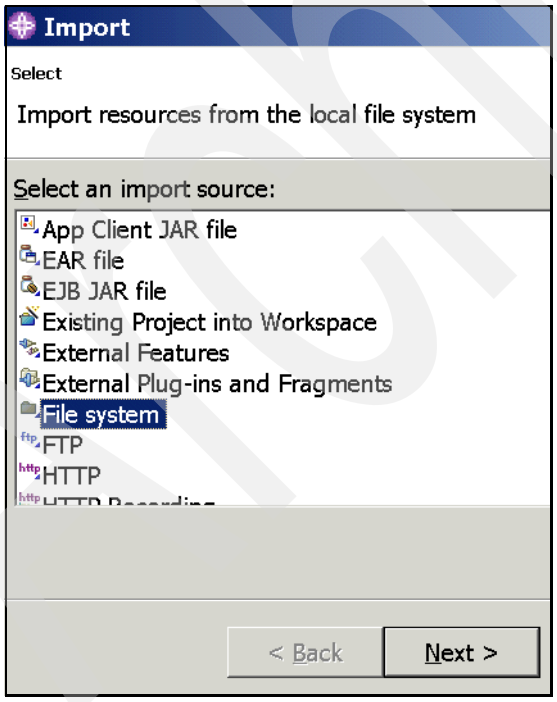


Figure 30-11 Import resources from the local file system

The Import window is where we locate the JDBC Java source to import. Enter the From directory into the input field, or browse to the directory containing the source. Our source was previously saved in the C:\sg247308 directory. In the right hand portion of the window, check **EmpRseAJ.java**. Leave the defaults that are automatically filled in on the window, and click the **Finish** button as shown in Figure 30-12.

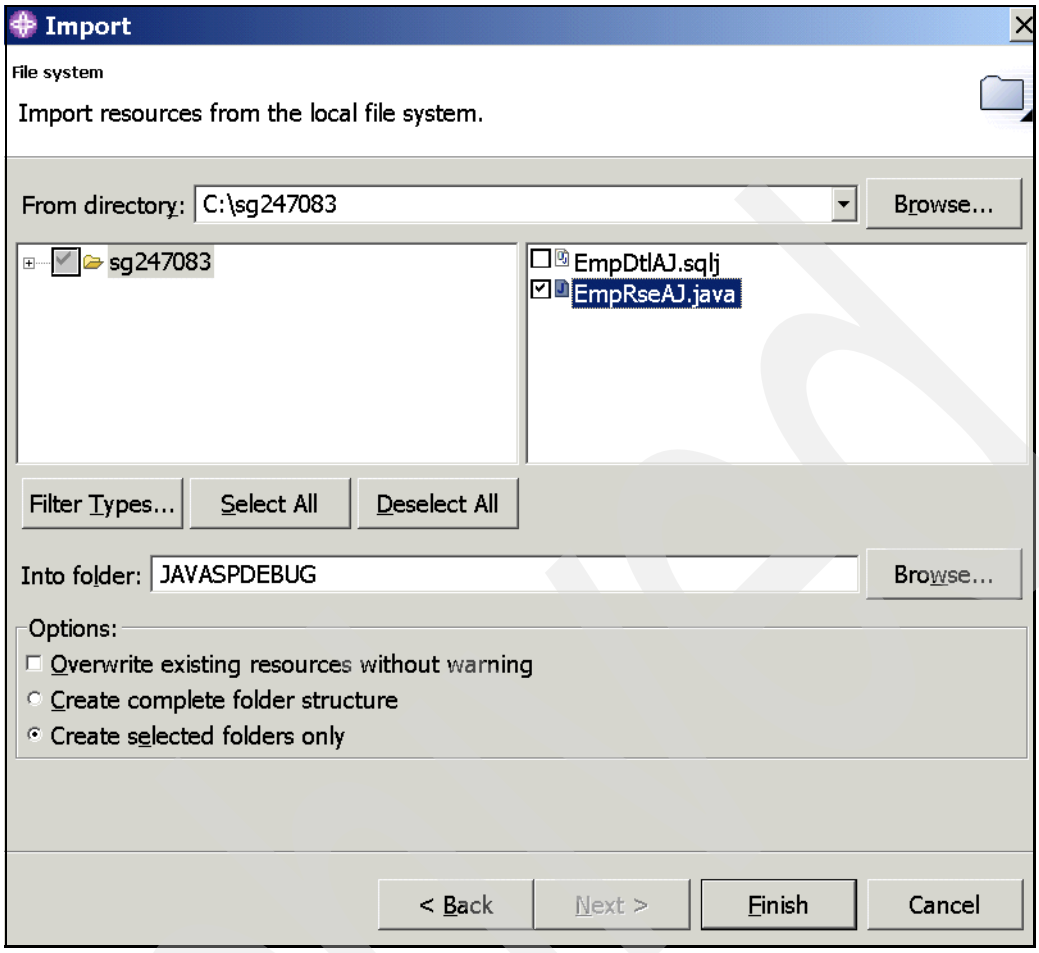


Figure 30-12 Import File system

The Package Explorer view of WSAD appears and includes our Java source. It is located in the default package folder under the JAVASPDEBUG object. Expand the folder and select the EmpRSEAJ.java file.

The Java driver being used on z/OS and the Application Development Client (V7.2 or V8.1) on the client determine the coding of the Class.forName class and the conndb2 method.

If the JDBC 2.0 driver is being used on z/OS, the Java stored procedure source code, which is converted to a Java application will need to be changed as follows:

1. Double click the file, and locate the DB2 server connection information in the Java source:

   ```
   Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver")
   ```

2. Replace it with the following:

   ```
   Class.forName("COM.ibm.db2.jdbc.app.DB2Driver")
   ```

3. Next, find the following line:

   ```
   conndb2 = DriverManager.getConnection("jdbc:db2os390sqlj:DB2G")
   ```

4. Replace it with the following:

   ```
   conndb2 = DriverManager.getConnection("jdbc:db2:DB2G","userid","password")
   ```

   where:

   – DB2G is replaced with your DB2 location alias name

– Userid is replaced with your TSO user ID.
– Password is replaced with your TSO password.

If the DB2 V8.1 Application Development Client is being used, then only the user ID and password have to be updated in the conndb2 statement.

While we have the Java source in edit mode, we are going to set a breakpoint. Locate the line:

```
System.out.println("The Dept " + workDept)
```

In the prefix area on the left side of the editor, right-click and select **Add Breakpoint** as shown in Figure 30-13.

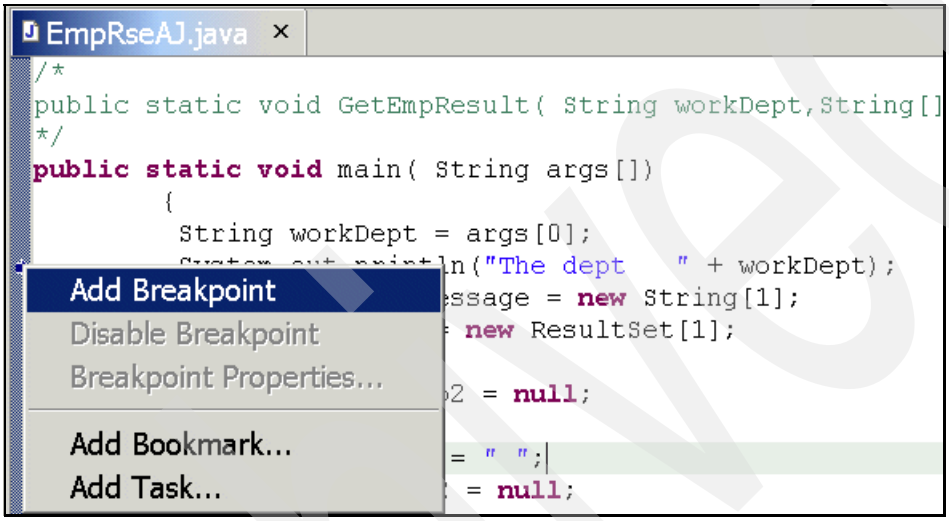


Figure 30-13 Add Breakpoint for Java applications

Close out the EmpRseAJ.java window by clicking the **X** in the upper right hand portion of the window. Reply with `yes` when prompted with `Do you want to save your changes`. Saving the file automatically recompiles the source and creates the Java class file.

From the Java Perspective, Package Explorer view, expand the default package folder. With the **EmpRseAJ.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 30-14.

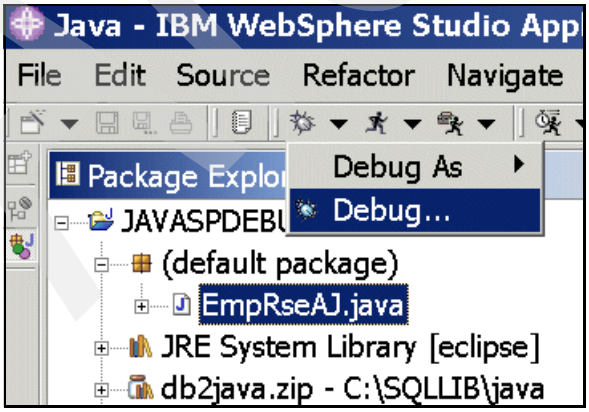


Figure 30-14 Select Debug definition option

This next set of windows is where we configure the debug options. From the Debug window, select **Java Application** and click **New** as shown in Figure 30-15.

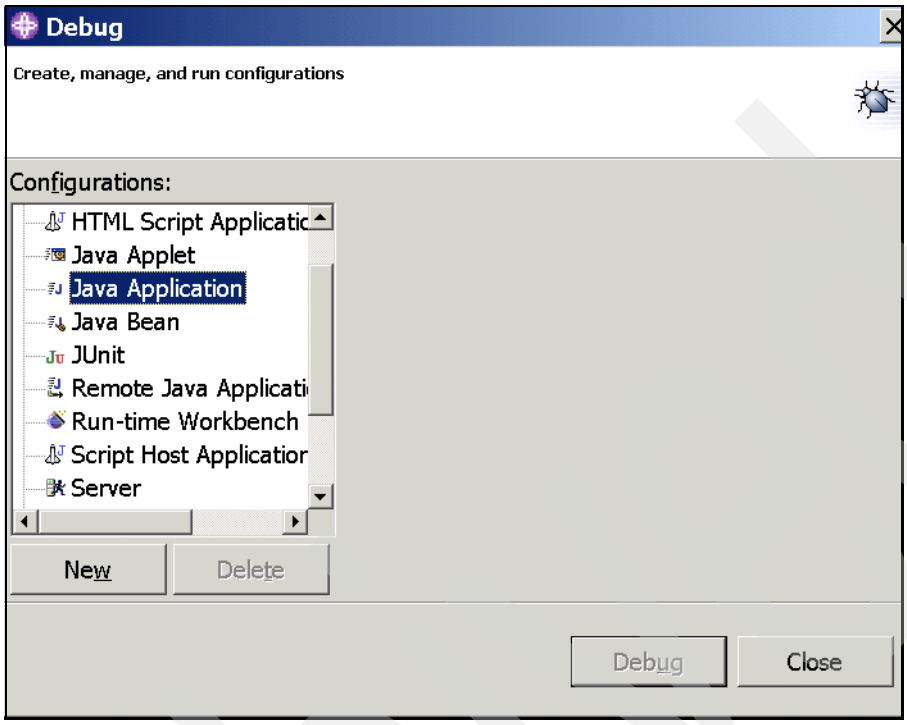


Figure 30-15 Configure Java Application for debug

In the Debug Main window, select **EmpRseAJ** from the Configurations portion on the left. On the right side of the window, enter the information in Table 30-1.

Table 30-1 Debug settings for Java application

Field	Value
Name	Java_Debug
Project	JAVASPDEBUG
Main class	EmpRseAJ.

Click the **Arguments** tab next to the **Main** tab that we are currently positioned on as shown in Figure 30-16.

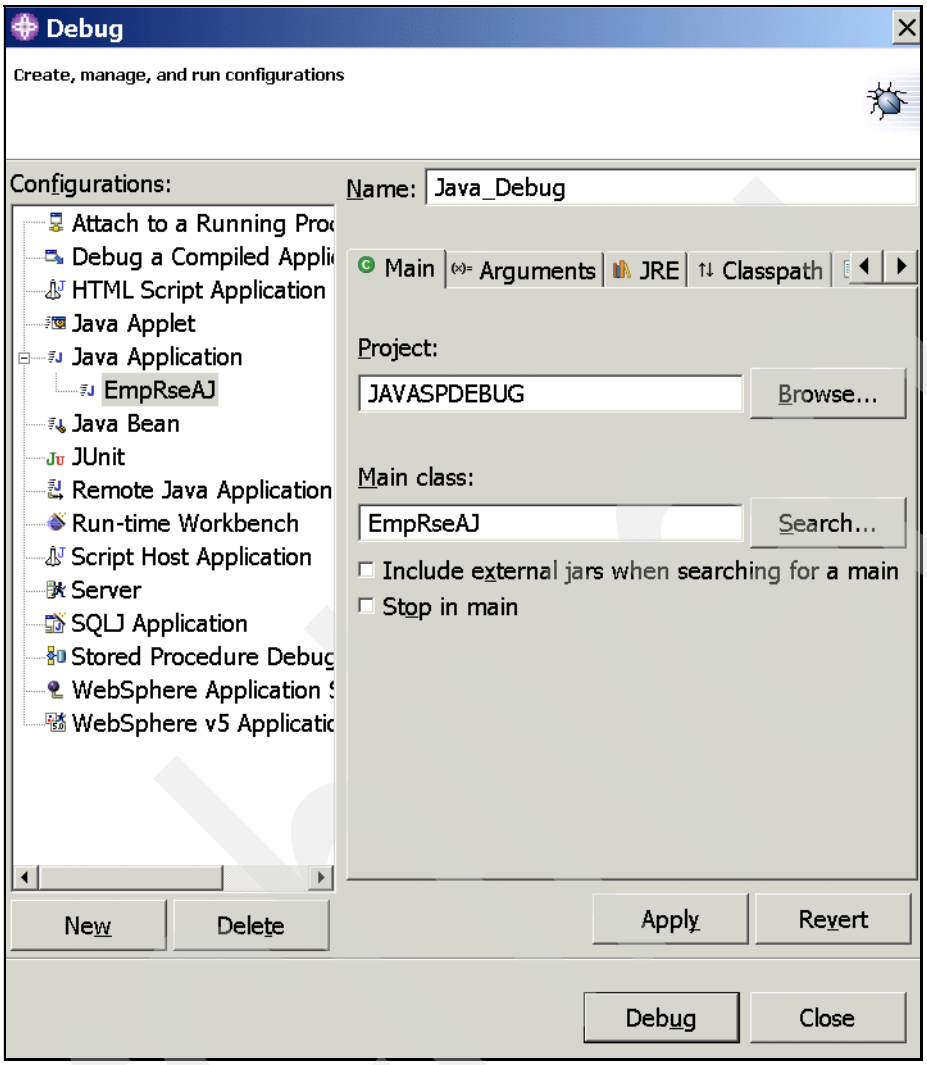


Figure 30-16 Java application Debug Main window definition

Our stored procedure requires input parameters to execute. These were removed as described in 17.7, “Debugging JDBC and SQLJ” on page 266. The Arguments window is where we specify these parameters. Enter `D11` for Department 11. 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 30-17.

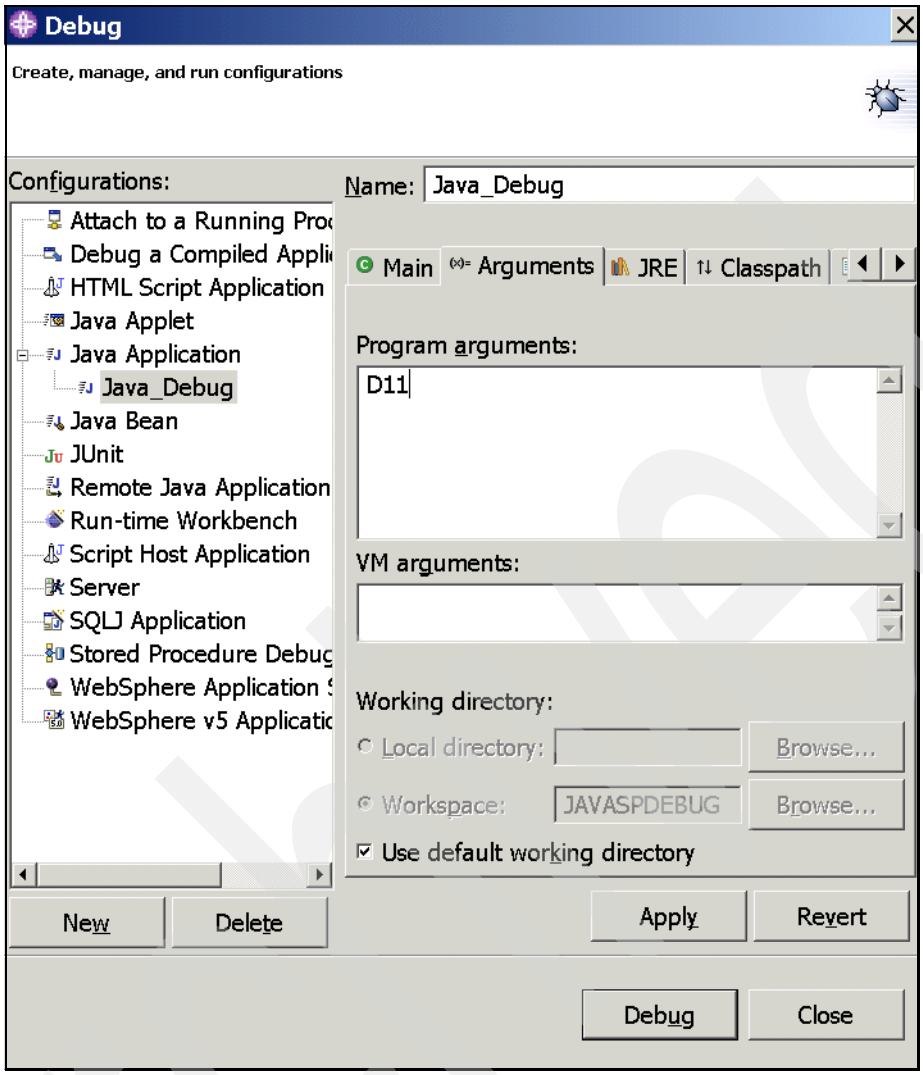


Figure 30-17 Java application Debug Arguments window definition

The debug Perspective opens, displaying the following default window settings:

Debug window is in the upper left portion of the window, variables is in the upper right portion of the window; our EmpJRseAJ.java source is in the middle portion of the window; an Outline view is in the middle right hand portion of the window; and a Console is in the bottom portion of the window.

Our Java source has been stopped at the breakpoint we set previously. In the upper right portion of the Debug window, icons to Run, Suspend, Stop, Disconnect, Remove all Terminated Launches, Step with Filters/Step Debug, Step Into, Step Over, Step Return, Listen for Debug Engines, and Enable/Disable Step by Step Debugging options can now be used to do debugging of our Java application. Breakpoints can be added or removed from the Java source in the Debug Perspective.

The Variables window in the upper right can be changed to show the Breakpoint, Expressions, Registers, Storage, Storage Mapping, Monitors, and Modules Display views as shown in Figure 30-18.

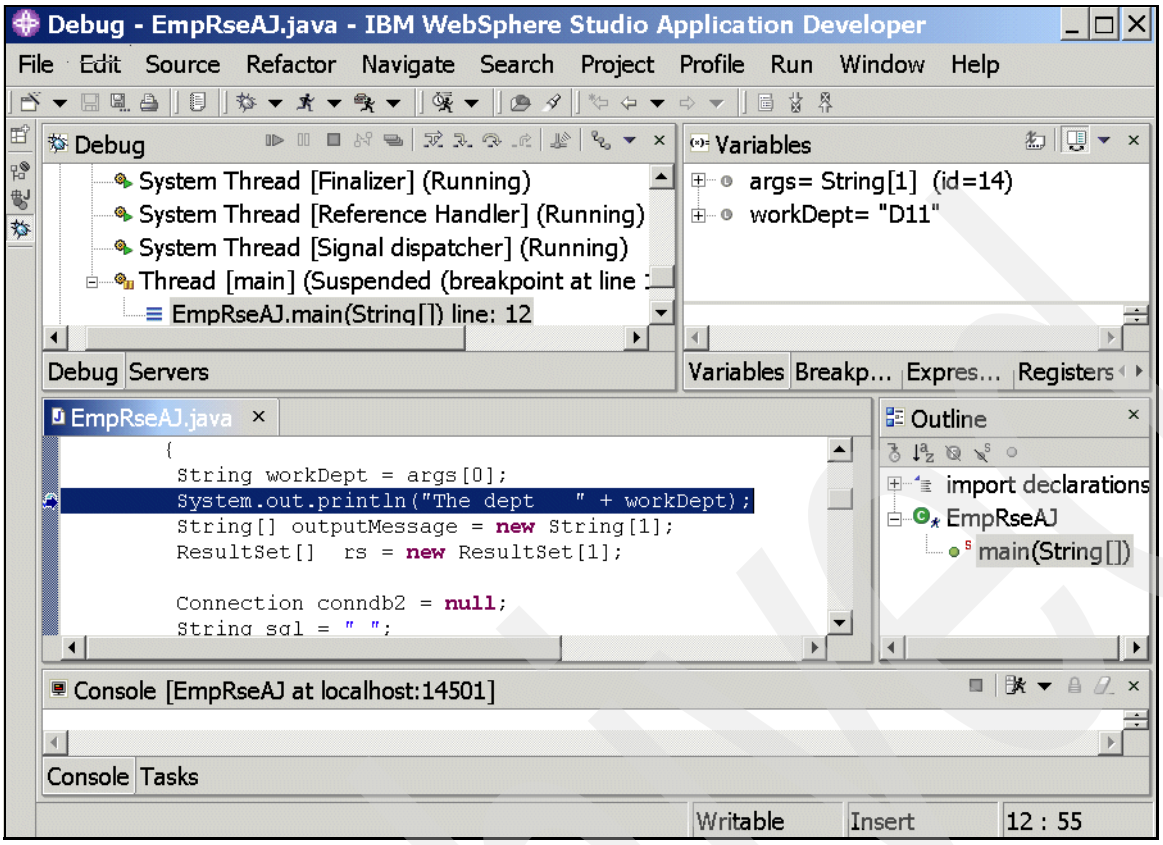


Figure 30-18 Debug Perspective started

30.2 Debugging SQLJ SPs converted to SQLJ applications

This section describes using WSAD to debug a SQLJ application that was converted from a SQLJ stored procedure on DB2 for z/OS.

If WSAD is not already started, launch WSAD as we did in 30.1, “Debugging JDBC SPs converted to JDBC applications” on page 554. Select the same workspace for the working directory.

Create a new Java Project and name it `SQLJSPDEBUG`, adding the same external jars from the Libraries window as were added previously and as shown in Figure 30-19. Click **Finish.**

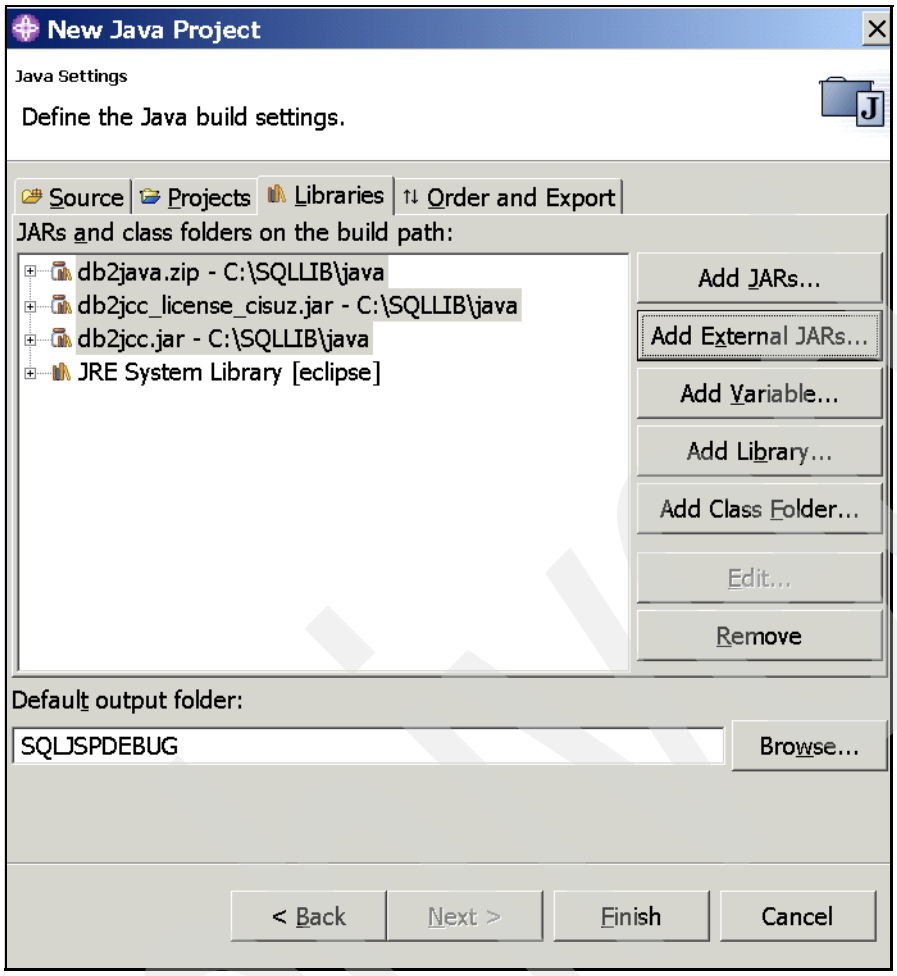


Figure 30-19 Create SQLJSPDEBUG project

The Java Perspective, Package Explorer view now has two Java projects: SQLJSPDEBUG in addition to JAVASPDEBUG. We will import our SQLJ source like we imported the Java source. Select **SQLJSPDEBUG**, then right-click **Import** as shown in Figure 30-20.

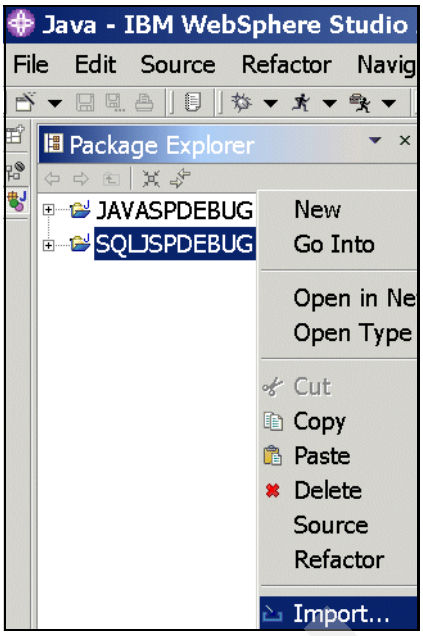


Figure 30-20 Import SQLJ application

On the Import window, select **File system** and click **Next** as shown in Figure 30-21.

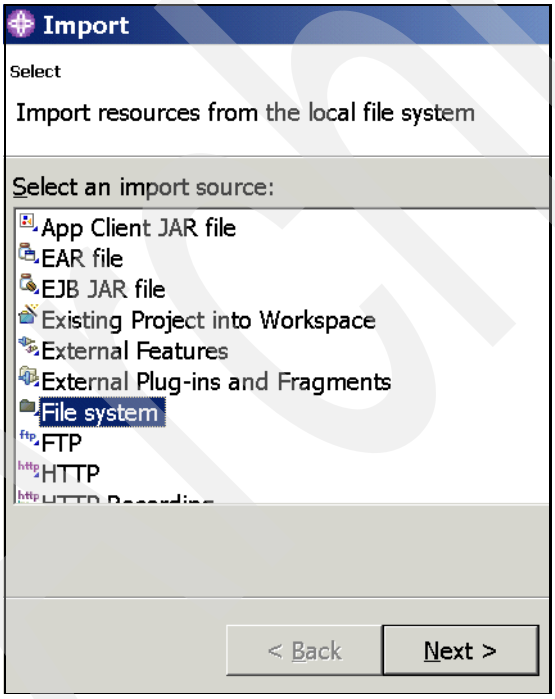


Figure 30-21 Import File system

The Import window is where we locate and import the SQLJ source. Enter the From directory into the input field, or browse to the workstation directory containing the source. Our source was previously saved in the C:\sg247308 directory. In the right hand portion of the window, check **EmpDtlAJ.sqlj.** Leave the defaults that are automatically filled in on the window as shown in Figure 30-22, and click the **Finish** button.

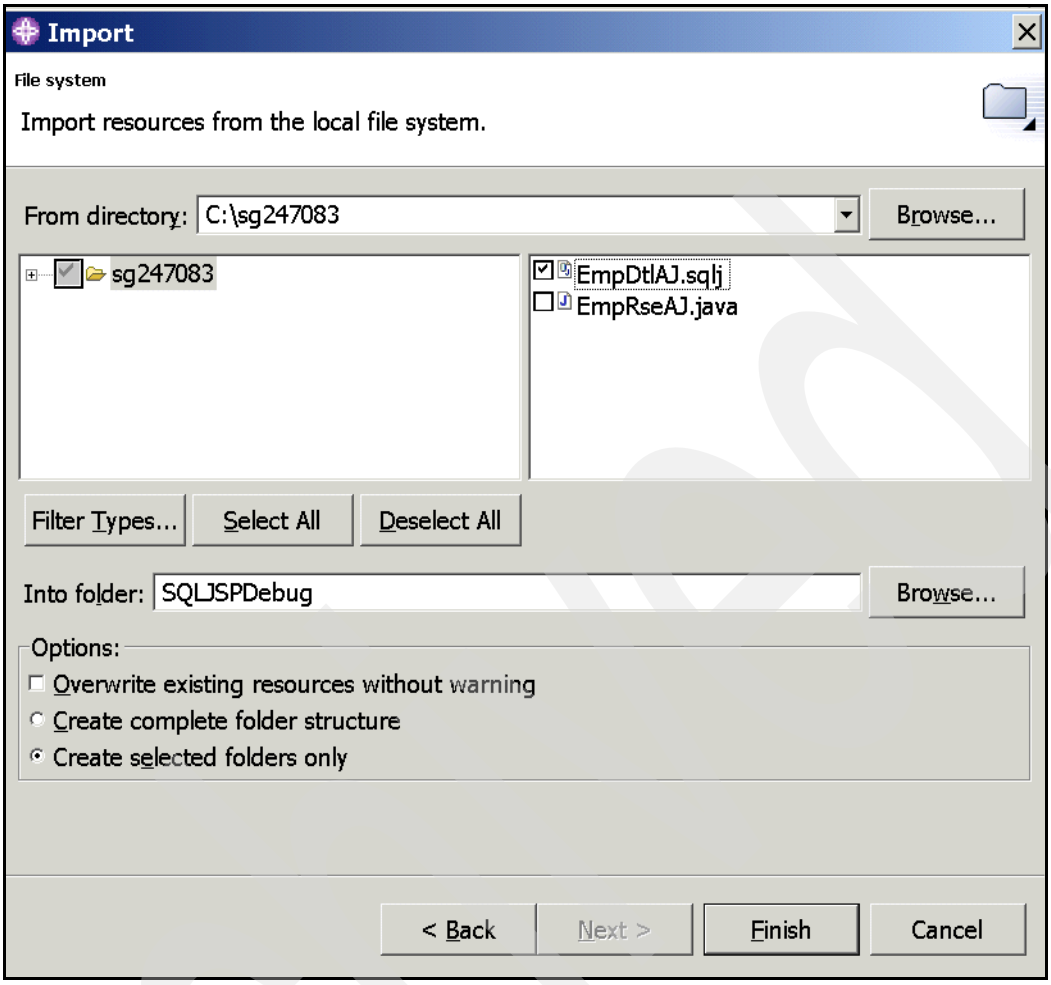


Figure 30-22 Import Resources from the local file system

Next, we added the SQLJ support. In the Java Perspective, Package Explorer, select project **SQLJSPDEBUG**. Right-click **Add SQLJ Support** as shown in Figure 30-23.

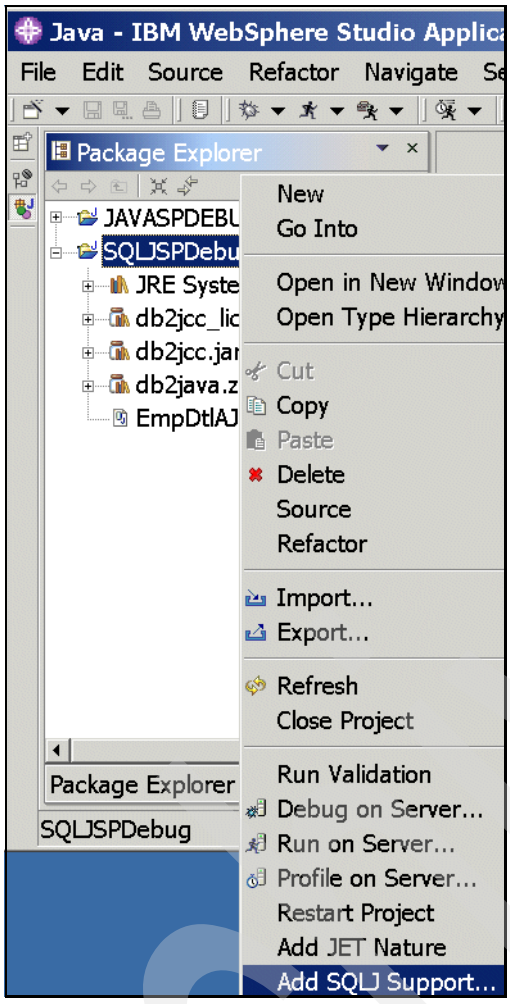


Figure 30-23 Add SQLJ Support

A list of available projects is returned. Select **SQLJSPDEBUG** and click **Finish** as shown in Figure 30-24.

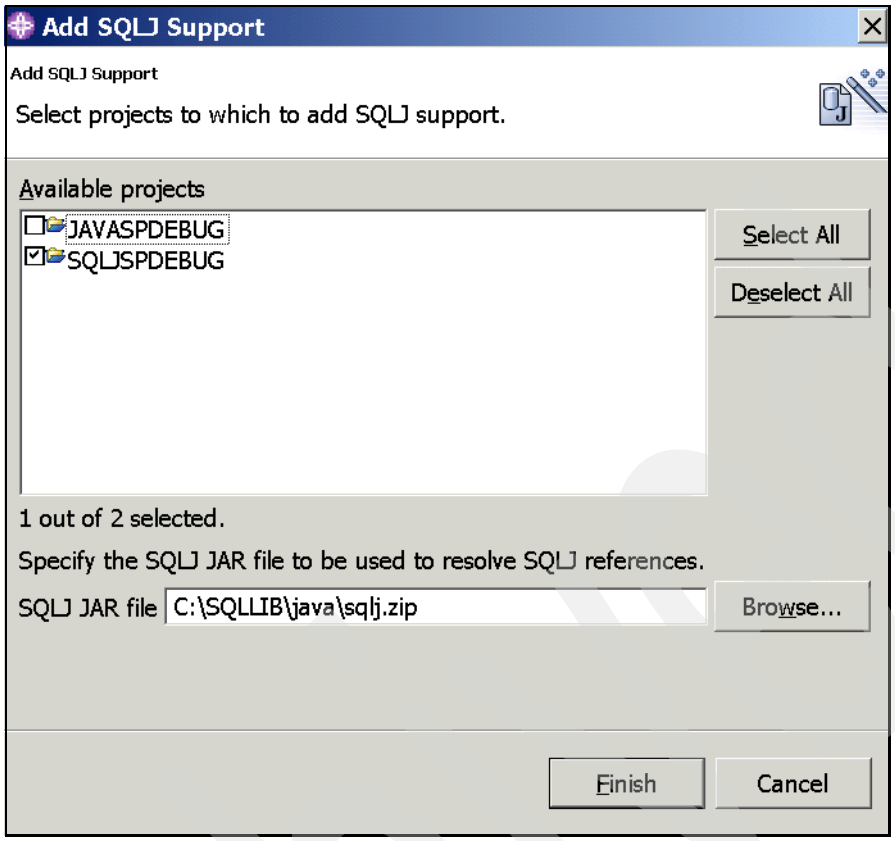


Figure 30-24 Select projects for SQLJ support

The SQLJSPDEBUG project now has the following objects added as shown in Figure 30-25:

- SQLJAnt Scripts
- sqlj.zip
- EmpDtlAJ.sqlj
- EmpDtlAJ_SJProfile0.ser

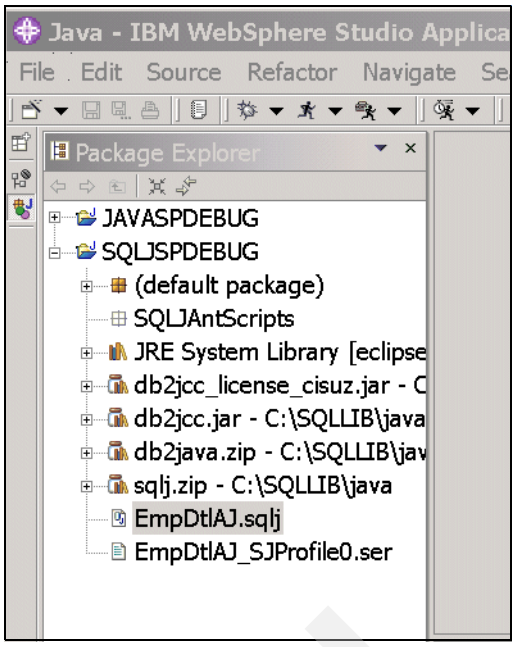


Figure 30-25 SQLJ support added to project

Next, we added our user ID and password to the DB2 for z/OS connection. Double click the **EmpDtlAJ.sqlj** file to open the editor. Since our SQLJ application uses db2jcc.jar, we only need to update the conndb2 method with a valid user ID and password for our z/OS DB2 server:

1. Locate the following line:

```
conndb2 = DriverManager.getConnection("jdbc:db2os390sqlj:DB2G");
```

2. Replace it with the following:

```
conndb2 = DriverManager.getConnection("jdbc:db2:DB2G","userid","password");
```

   Where:

   – DB2G is replaced with your DB2 location alias name
   – User ID is replaced with your TSO userid
   – Password is replaced with your TSO password.

Before closing the editor, we set a breakpoint in the prefix area of the #sql iterative for the SELECT statement. Click the **X** in the upper right corner to close the editor as shown in Figure 30-26. Reply yes when prompted to save the updates.

Figure 30-26 SQLJ application source

Next we configure a debug session for the SQLJ application. From the Java Perspective, Package Explorer, select **EmpDtlAJ.sqlj,** then from the menu, then click the **bug** icon to enable the pop-down list and select **Debug** as shown in Figure 30-27.

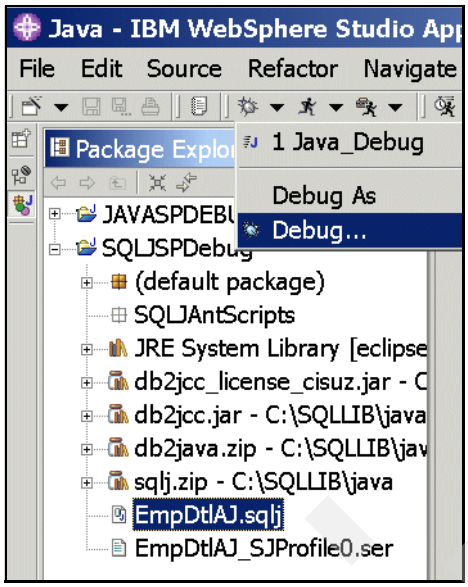


Figure 30-27 Launch the Debug configuration

Locate the **SQLJ Application** in Configurations in the left hand portion of the window, click **New** as shown in Figure 30-28.

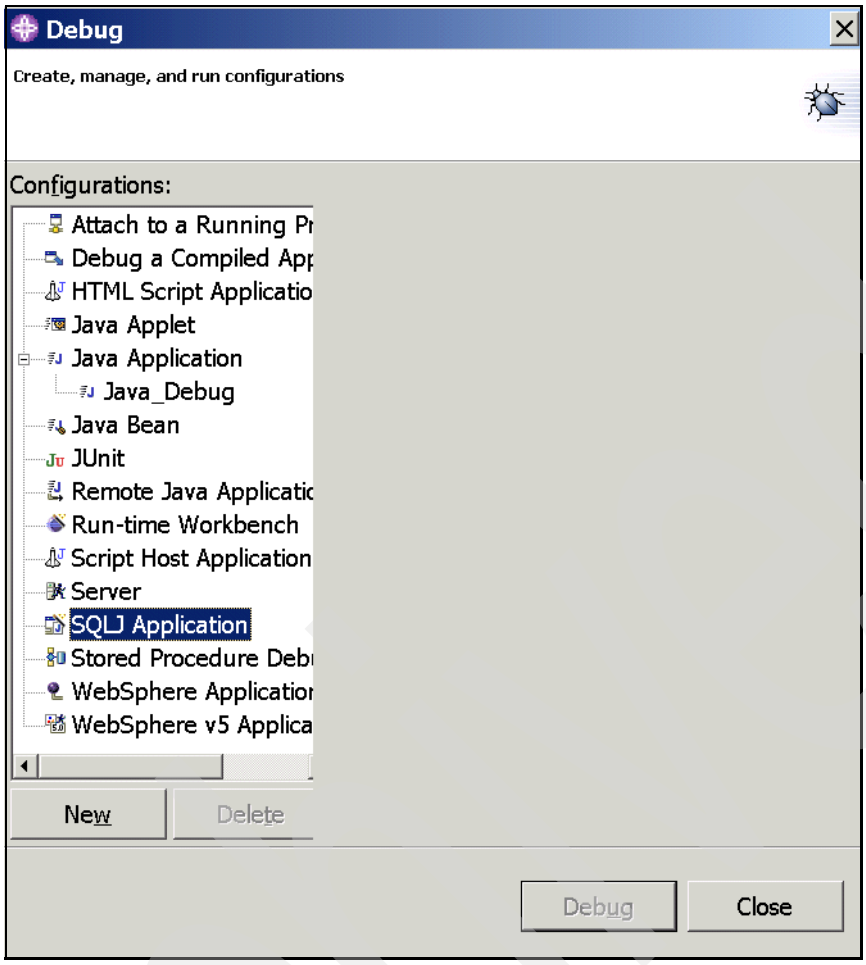


Figure 30-28 Create a New SQLJ Application debug configuration

In the Debug Main window, select **EmpDtlAJ** from the Configurations portion on the left. On the right side of the window, enter the information in Table 30-2.

Table 30-2 Debug settings for SQLJ applications

Field	Value
Name	SQLJ_Debug
Project	SQLJSPDebug
Main class	EmpDtlAJ

Then click the **Arguments** tab as shown in Figure 30-29.

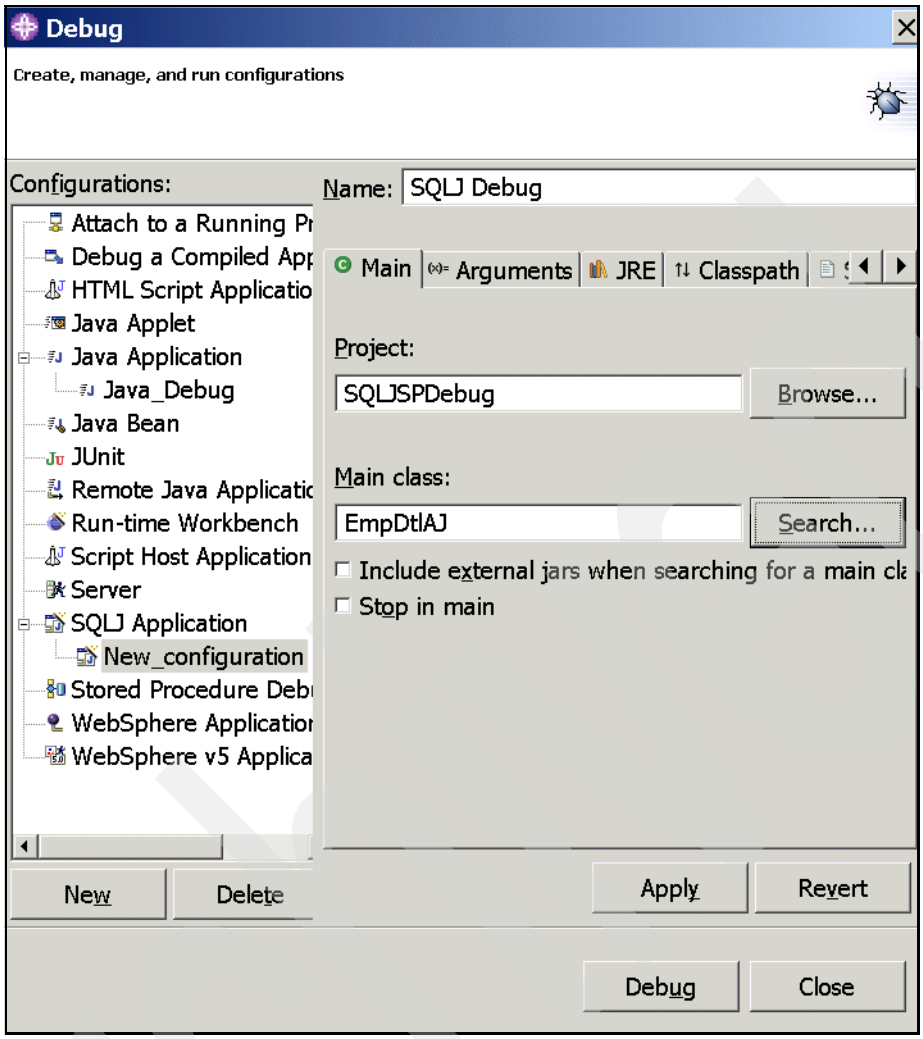


Figure 30-29 Define new SQLJ Debug configuration

Our stored procedure requires input parameters to execute. These were removed as described in 17.7, “Debugging JDBC and SQLJ” on page 266. The **Arguments** window is where we specify these parameters. Enter 000010 selecting an employee number in the EMP table. Clicking **Debug** starts the debug session as shown in Figure 30-30.

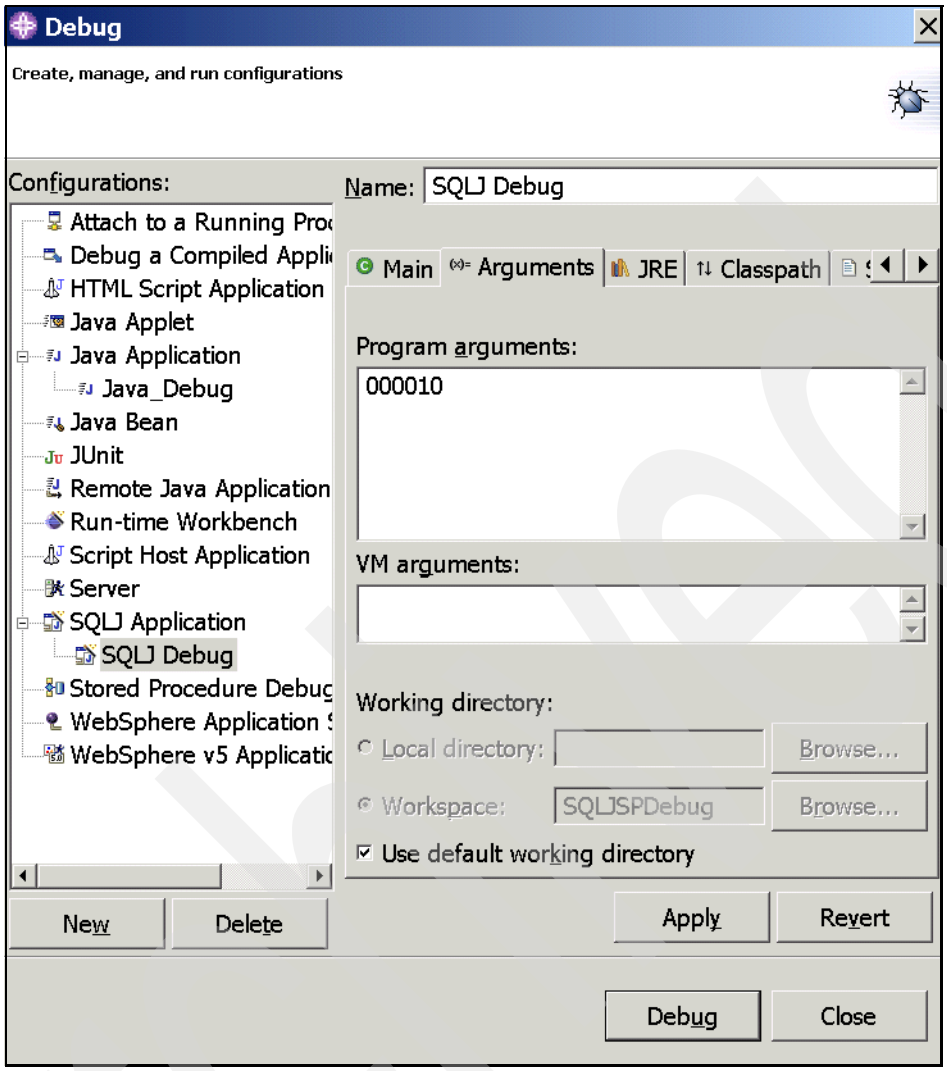


Figure 30-30 Define program arguments for the SQLJ debug session

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 30-31.

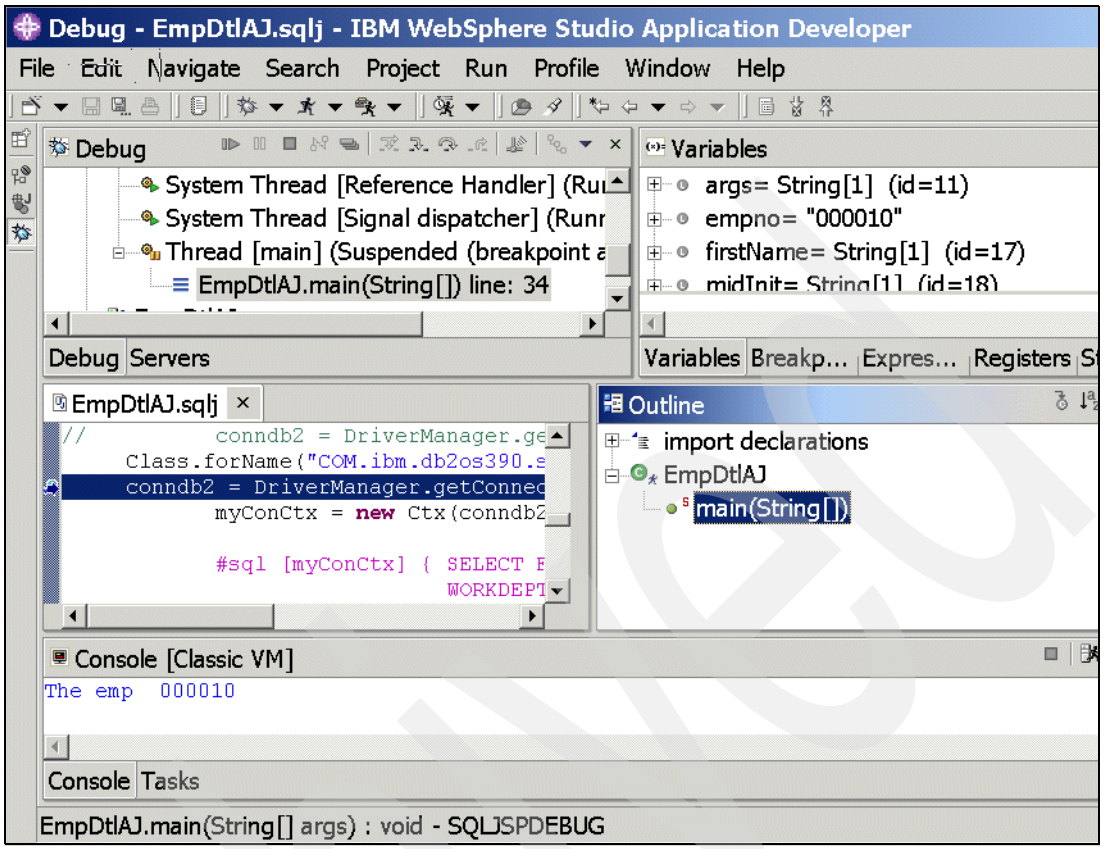


Figure 30-31 Debug Perspective launched for SQLJ application

Once debugging is completed, close out the Debug Perspective.

Part 8

Appendixes

This part includes the Appendixes defined during this project:

A

Frequently asked questions

Stored procedures have been available since DB2 for MVS/ESA Version 4. Since their initial implementation, and up through the announced changes in Version 8, you have asked us a number of questions about developing and managing stored procedures.

In this appendix we provide a list of the most frequently asked questions, and provide a short answer for each followed by a reference to a manual or redbook where you can access more detailed information. For those topics that are covered in more detail in this redbook, we provide a link to the appropriate chapter. The intention is not to provide a comprehensive answer but simply a place where you can start. The questions cover a wide range of topics, from application development to tuning WLM, and we have arranged them into categories to meet your individual needs.

See the informational APAR II11771 for current hints and tips on stored procedures.

This appendix contains the following:

- Application development FAQs
- Performance FAQs
- Administration FAQs
- Miscellaneous FAQs

A.1 Application development FAQs

Table A-1 contains a list of frequently asked questions that deal with application development topics for stored procedures.

Table A-1 Application development frequently asked questions

No.	Question	Answer
1	Can the RAISE_ERROR function be used in a COBOL stored procedure to return an error?	Yes, it can be used, but it will only raise the error on the statement in the COBOL stored procedure. It will not affect the SQLCODE for the SQL CALL statement used to invoke the stored procedure. For more information on the RAISE_ERROR function, see "Chapter 3" of DB2 UDB for z/OS Version 8 SQL Reference, SC18-7426. When you use parameter style DB2SQL, additional options are available to you. For details, see 10.2.6, "Handling PARAMETER STYLE DB2SQL" on page 100 in this redbook.
2	Are triggers fired by dynamic SQL statements?	Yes. Triggers are fired by any SQL statement that performs the triggering event specified in the CREATE TRIGGER statement. For more information on when triggers are fired, see "Chapter 12' of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide*, SC18-7415. For more details on how triggers interact with stored procedures, see Chapter 27, "Using triggers and UDFs" on page 451 of this redbook.
3	Can a call-attach program be used in lieu of writing a stored procedure? We have a need to modify a large quantity of non-DB2 programs that will all need to call a common routine for DB2 based information. We would rather not have to transform all these programs to DB2 in order to call a stored procedure.	Yes, a program can be written to attach to DB2 using the call attachment facility (CAF). For more information on using CAF, see "Chapter 30" of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415
4	Where can I find a sample of a simple stored procedure written in language SQL?	Sample SQL procedures are provided with the additional materials for this redbook. Refer to Appendix D, "Additional material" on page 651 of this redbook for instructions on accessing the SQL language sample procedures that are available on the IBM redbooks Web site.
5	Do z/OS stored procedures support the SQLDA interface? We would like to take maximum advantage of code reuse by utilizing existing routines as stored procedures. One limitation is that current programs exchange large numbers of variables as host structures instead of variables. These structures cannot be converted to result sets. The CREATE PROCEDURE maximum statement length in SPUFI prevents us from defining the current number of variables exchanged. The COBOL compiler also has a limit on the number of variables used on the PROCEDURE DIVISION statement in the stored procedure.	The SQL CALL statement on z/OS does support the USING DESCRIPTOR syntax for specifying parameters when invoking a stored procedure. If the SPUFI limit is not as big as the 32KB limit for an SQL statement, then consider using a different method to get to the 32KB statement length. The limit for an SQL statement in V8 is raised from 32KB to 1MB. You may want to contact the COBOL team to see whether there is a way to increase the compiler limit.

No.	Question	Answer
6	What is the best way to promote SQL stored procedures through a development life cycle? After successfully testing an SQL stored procedure on one DB2 subsystem, we could recreate the SQL stored procedure using Stored Procedure Builder on another DB2 subsystem, but this is not a true promotion of tested code.	There are two alternatives that you can use: ► If a recompile is required, which can happen if the source and target systems are running incompatible levels of Language Environment, or if you are promoting between two completely different operating systems that require different compilers, the best method is to use the DB2 Development Center client program to copy and paste the program and rebuild it to the target system. ► If a re-compilation is not required or not allowed, then you can copy the load module, copy the DBRM, bind the package, and issue a CREATE PROCEDURE statement, which you can extract from the source and execute as an SQL statement. This will not save the source or the build options on the target system, but they would not be needed if builds are never done against the production system. Some of the popular change management tools are adding support for stored procedures, so if you use that type of tool for other programs, you are encouraged to contact the vendor about SQL procedures support. See Chapter 16, “Code level management” on page 223 for details.
7	Is there a limit to the size of a COBOL stored procedure?	There are no restrictions on module size, other than what is available in the address space.
8	Do all COBOL stored procedures need to be compiled and linked as re-entrant (RENT) and re-usable (REUSE)?	RENT and REUSE are not required, but these options are recommended so that a single copy of the load module can be used for multiple invocations, which will improve performance.
9	Where can I find detailed documentation about how to use the DB2 Stored Procedure Builder?	For details on the DB2 Stored Procedure Builder, see “Chapter 2” of the IBM Redbook *Cross-Platform DB2 Stored Procedures: Building and Debugging*, SG24-5485-01. The DB2 Development Center extends the capabilities of the DB2 Stored Procedure Builder. For details on the DB2 Development Center, see Chapter 29, “The DB2 Development Center” on page 499 of this redbook. There are also links to information from the Stored Procedure Builder Web site. There are several articles on both the Stored Procedure Builder and the DB2 Development Center on the DB2 Developer Domain.
10	Can I call mainframe stored procedures from SQL Server?	There is some information about accessing mainframe databases using ODBC on the Microsoft Web site. Access the Web site and search on ODBC for more details.

No.	Question	Answer
11	There are two alternatives for accessing CICS from a DB2 stored procedure: EXCI and DSNACICS. Which one should I use in which circumstance, and which one will perform better?	The external CICS interface (EXCI) can be used to link to a CICS program from a DB2 stored procedure. Some knowledge of CICS language syntax is required. DB2-supplied stored procedure DSNACICS accomplishes the same purpose without requiring the developer to understand CICS language syntax. DSNACICS is written to use the DPL services and allocates a PIPE between CICS and a TCB in the stored procedure address space. This pipe stays allocated even when DSNACICS completes. If there is another call to DSNACICS, which is to invoke a CICS transaction in the same CICS applid, then the already allocated PIPE will be used for the second and subsequent invocations of CICS transactions. This allocation of the pipe which "belongs" to a stored procedure TCB can be a reason to put DSNACICS into its own WLM address space, so that any invocation will have a higher chance of being assigned to a TCB which has already established the pipe to the CICS region. On a performance note: The exit DSNACICX is only loaded one time per the address space that is executing DSNACICS, so if you funnel DSNACICS calls to the same WLM address space, you will avoid having the cost of the load issued in more generic stored procedure address space. See Chapter 24, "Accessing CICS and IMS" on page 385 for more details on the two alternatives.
12	It looks like I can access VSAM files directly from a stored procedure or do it through CICS. What method should I use?	The method you use should depend on the resources available in your environment. If you have minimal CICS skills you may wish to access the VSAM files from your stored procedure. If you have significant CICS skills and legacy applications that access VSAM data, then you may wish to use EXCI or DSNACICS to access the data. See Chapter 24, "Accessing CICS and IMS" on page 385 for more details.
13	It looks like I can access IMS databases directly through ODBA or by using an IMS transaction manager. Which should I use?	The IMS Open Database Access interface (ODBA) can be used to access IMS data directly from your stored procedure. DB2-supplied stored procedure DSNAIMS will allow access to IMS transactions and commands from a DB2 stored procedure. As of the publication of this redbook DSNAIMS was not yet available. See Chapter 24, "Accessing CICS and IMS" on page 385 for more details on ODBA and DSNAIMS.
14	There appears to be some overlap in the functionality of a stored procedure and a user defined function when invoked by a trigger. Which one should I use?	This question is really outside the scope of this redbook, but see 27.4, "Stored procedures versus user defined functions" on page 456 for some discussion. Various other redbooks have discussed this earlier. *See the redbook DB2 for z/OS Application Programming Topics*, SG24-6300-00 for more details.
15	I have multiple choices for the language of my stored procedures. Which languages should I use, and what are the factors that should influence my decision?	This decision should be based on various factors and we will list these instead of recommending a particular language: - Existing expertise - Productivity (REXX and SQL are faster to develop, SQL is faster development using DB2 Development Center) - Performance (REXX should not be used for mission critical applications) - Impact on other parameters (e.g. NUMTCB must be 1 for REXX)
16	Can a stored procedure call itself recursively, and are there any limitations?	Yes, a stored procedure can call itself recursively. There is a limit of 16 to the number of levels of recursion.

No.	Question	Answer
17	What are the benefits of using PARAMETER STYLE DB2SQL? It enables some extra stuff in the parm list returned to the calling application. Does the calling program have to be coded to handle this extra information? Is there a description published of exactly what is passed back?	This parameter style provides additional data back to the caller including the ability to set SQLSTATE. The SQLSTATE is not always passed directly but converted. For a discussion of this topic, see 10.2.6, “Handling PARAMETER STYLE DB2SQL” on page 100.
18	How do I debug a DB2 stored procedure that is written in COBOL or other languages?	If you have VisualAge COBOL installed on your workstation, and the Debug Tool installed on your z/OS or OS/390 system, you can use the VisualAge COBOL Edit/Compile/Debug component with the Debug Tool to debug a COBOL stored procedure that runs in a WLM-established stored procedures address space. The Debug Tool works for COBOL MAIN and COBOL SUB program types, which are specified on a CREATE PROCEDURE STATEMENT. For more information on debugging stored procedures, see the *DB2 Application Programming and SQL Guide* for your version of DB2 at the DB2 for z/OS and OS/390 library Web page. If you are debugging a Java stored procedure, see the *DB2 Application Programming and SQL Guide for Java* for your version of DB2 at the DB2 for z/OS and OS/390 library Web page. For further discussion of this topic, see Chapter 14, “Debugging” on page 173.

No.	Question	Answer
19	We have developed a mainframe-based application that invokes DSNTPSMP and creates SQL stored procedures. When invoking DSNTPSMP, we pass it a source data set name and leave the SQL source field as a blank string. This process causes it to read the source from a data set, which works fine for creating the stored procedure. However, when the user starts Stored Procedure Builder, selects a stored procedure, and then selects **Get Source**, the source code is not parsed correctly. If the procedure is built from SPB (W2K), the source appears correctly.	When input from the data set is read, a string is created. That process injects a line formatting character between records. In your case, the new line formatting character used was (x'15') NEXTLINE, which is the standard line formatting character used in the mainframe realm (the same one C uses for \n on the mainframe). DB2 Stored Procedure Builder, prior to FP8, does not recognize all the new line formatting characters that are possible. There are several: linefeed, carriage return, nextline, and so on. The adverse result of this situation is normally observed when the source appears to be formatted as a single line. Unicode translation is being used when retrieving the SQL source from the DB2 catalog. There are multiple ways to correct the problem. See Info APAR II13277 for setting up the Unicode conversion services, or for bypassing the Unicode translation. Alternatively, you may use one of the following service levels: 1. Use SPB at FP8 or “higher”. (FP10A is available and is recommended.) The (mainframe) NEXTLINE new line formatting character is recognized. SPB formats the lines appropriately for display. For this option to be effective you must also be using a DB2 for OS/390 and z/OS V7 server. 2. Use DSNTPSMP 1.15 (APAR PQ45854, see link below). This update was a comprehensive update to DSNTPSMP, which included revisions to the handling of the source passed from a data set named in parameter-10 (your scenario). The injected new line formatting character was changed to (x'15') LINEFEED, aligning with the predominate new line character used outside of the mainframe realm. This update is consistent with the new line formatting character, which SPB provides itself within the SQL procedure source string. For your existing SQL procs, you can rebuild them, which causes the source to be saved again. This action can be performed easily by repeating your builds from a data set source, but this time using parameter-1 FUNCTION_REQUEST='ALTER_REBUILD' (You could also use ALTER or ALTER_REBIND but these require that an ALTER PROCEDURE statement also be passed in parameter-9.)
20	What are IBM DB2 SQL procedures and IBM DB2 Stored Procedure Builder?	IBM DB2 for z/OS and OS/390 offers SQL Procedures language for creating stored procedures. SQL is enhanced with procedural constructs for writing stored procedures. When you use Stored Procedure Builder to create SQL procedures, SQL statements are automatically embedded in the stored procedure and the source code is prepared for use with DB2. By using SQL, you use only one programming language to create your stored procedures. Use the Stored Procedure Builder to easily develop stored procedures. When you add your logic to an SQL procedure, Stored Procedure Builder and DB2 automatically produce and manage the required resources, such as DB2 catalog content, for the stored procedure. For more information, go to the DB2 Stored Procedure Builder Web page. http://www-306.ibm.com/software/data/db2/os390/spb/

No.	Question	Answer
21	How can I retrieve the SQLSTATE after calling an SQL stored procedure?	You can retrieve the SQLSTATE from the SQLCA after the stored procedure has completed. Prior to DB2, Version 8, the SQLSTATE and SQLCODE were usually returned as 00000 and 0, respectively, regardless of the actual exit condition of the procedure. In some cases, valid SQLSTATEs and SQLCODEs were returned. For instance, in Version 6, a ROLLBACK statement that is executed within a stored procedure causes a negative SQLCODE to be returned to the caller. Also, APAR PQ56323 enabled unhandled exceptions (negative SQLCODEs and error SQLSTATEs) to be returned instead of 0. In Version 8, you can choose any SQLSTATE to send back to the program, along with optional message text. Use a SIGNAL or RESIGNAL statement to specify the SQLSTATE to return. The message text can be retrieved with GET DIAGNOSTICS.
22	What is the difference between the SIGNAL and RESIGNAL SQL statements?	Currently, the SIGNAL and RESIGNAL statements have only the following two superficial differences: - RESIGNAL must be included in a handler, whereas SIGNAL can be included anywhere in an SQL stored procedure. - You can specify RESIGNAL; by itself with no other syntax. You do not have to specify an SQLSTATE. (When you specify RESIGNAL without an SQLSTATE, DB2 uses the SQLSTATE that invoked the handler.) For SIGNAL, you must specify an SQLSTATE. RESIGNAL as defined in the SQL standard has more functionality than is in DB2, Version 8.
23	We are having problems specifying schema names when building SQL stored procedures. What do we need to do to make this work?	First, make sure that your WLM environment is defined with NUMTCB=1. You need to apply SPB FixPak 10A and use it with DSNTPSMP level 1.14 or higher. SPB FixPak 10A enables the schema name to include the following alphabetic extender characters: $, # and @. Without this fix, a schema name that contains any of the alphabetic extenders must be specified as a delimited identifier. DSNTPSMP level 1.14 and higher supports schema names that are specified as delimited identifiers. This level was introduced by APAR PQ58235. To search our library of closed APARs on the z/OS and OS/390 platform, go to the IBM eServer Support Database.
24	Running a REXX stored procedure, such as DSNTBIND or DSNTZALI, returns message: 382 *-* ADDRESS LINKMVS "DSNESM71" +++ RC(-78) +++ What does this message mean?	First make sure the REXX stored procedure is running in an environment with NUMTCB=1; without this, unpredictable results are possible. REXX has an error/abend handling routine. The -78 comes means that there is a DB2 04E abend. The '04E'X is 78 decimal. The negative sign indicates that it is a system abend, not a user abend. Check to see which ABEND04E occurred. This information is documented in the TSO/REXX Reference manual.
25	Is there anything I have to do to IMS (like put on maintenance) that would allow me to call IMS from s stored procedure.	You currently can use the ODBA interface to do IMS database calls within your stored procedure. If you want a stored procedure to call an IMS TM region, then you will be able to do that by calling DB2-supplied stored procedure DSNAIMS, when the PTF is available for this. The APAR number is PQ77702
26	In a completely COBOL environment with good COBOL programmers, do you know of any reason we would not use PROGRAM TYPE SUB for everything.	If you know you have good programmers then there is no good reason not to use SUB for everything.

No.	Question	Answer
27	Are there any published comparisons of SQL stored procedures and COBOL stored procedures? Performance difference, language differences, etc.	In general, COBOL stored procedures are the best performers, but this a complex issue that involves ease of development, speed of development and operating costs.
28	Program A calls store procedure 1, stored procedure 1 calls stored procedure 2, and stored procedure 2 calls stored procedure 3. Stored procedure 3 opens a cursor and does not perform a close, with the intent of making the answer available to the caller as a result set. Will program A be able to use the result set created by stored procedure 3?	Result sets are only returned to the program that calls the stored procedure, so program A will not be able to use the answer set created by stored procedure 3. If you need to access that data through DB2 in program A you could create temp tables in the intermediary stored procedures and pass result sets to the next highest level.
29	In our development area we have 20 IMS/TM systems using a single DB2 subsystem, with different levels of development occurring on each IMS. What happens when two different development teams need different levels of the same stored procedure? They would prefer not to code SET CURRENT PACKAGESET because this situation would never happen in production and they do not want to change code before going live. Would you recommend using SET CURRENT PACKAGE PATH or using multiple WLM environments with different steplibs?	It is difficult to provide a general recommendation for the set up since each site is different. See 22.4, “CURRENT PACKAGE PATH special register” on page 360 of this redbook for a discussion of SET CURRENT PACKAGE PATH.

A.2 Performance FAQs

Table A-2 contains a list of frequently asked questions that deal with tuning the performance of stored procedures.

Table A-2 Performance FAQs

No.	Question	Answer
1	We have CICS programs running on the mainframe that contain application logic to manage our DB2 data. When we update a table we also issue a COBOL CALL to another module that writes to a history DB2 table. I understand that we can instead call a stored procedure to write to the history table. Would you recommend we stay with the original application logic or convert to stored procedures for these history database calls.	You need to be careful about converting common routines that run on the mainframe to stored procedures. Stored procedures were designed to save network transmission time for client applications that make multiple SQL calls to DB2 for z/OS data. There is overhead in invoking a stored procedure, because the address space may need to be started and the program may need to be loaded into the address space. An alternative solution is to use a trigger for the update to the history table. See 10.3, "COBOL subprogram interfaces" on page 111 for a comparison of the SQL CALL and COBOL CALL statements. For more information on using triggers, see *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide*, SC18-7415.
2	Is there a significant overhead in specifying a large number for the number of dynamic result sets? If not, can I declare 999 or some huge number?	We are not aware of any performance degradation from this. The actual number of dynamic result sets is more important.
3	Is there a significant overhead in specifying a large parameter area?	We are not aware of any significant overhead. What is more important is to minimize the transmission time by setting all unnecessary variables to null and using a parameter style that supports nulls (GENERAL WITH NULLS or DB2SQL).
4	I seem to have a choice of methods available to access remote objects - MRO, 3-part name, alias, remote package, MQ etc. What are the trade-offs? What should I choose?	Using DRDA protocol, from DB2 for OS/390 V6 onwards, all methods of accessing remote data through DB2 should work equally well. The comparison of CICS, MQ-Series and DB2 is beyond the scope of this redbook.
5	What general recommendations do you have for improving the performance of stored procedures?	1. Use the following recommended options for the CREATE PROCEDURE statement: PROGRAM TYPE SUB STAY RESIDENT YES COMMIT ON RETURN YES for stored procedures that are called in a distributed environment. 2. Use WLM-managed stored procedures. 3. Use the NUMTCB more appropriate for your environment. 4. Make sure the LE libraries are preloaded with LLA, and use the Language Environment (LE) run-time options that are documented in "Chapter 25" of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide*, SC18-7415. This chapter provides recommendations on LE run-time options to minimize storage usage below the 16-MB line. You can find the *DB2 Application Programming and SQL Guide* for your version of DB2 at the DB2 for z/OS and OS/390 library Web page: `http://www-306.ibm.com/software/data/db2/os390/library.html`

No.	Question	Answer
6	How do I set stored procedure priority?	Stored procedure priority is inherited from the caller. The stored procedure always runs at the dispatching priority of whatever called it. So, if you call a stored procedure from CICS, it runs at CICS priority. If you call a stored procedure from batch, it runs at batch priority. If you call a stored procedure from DDF, it runs at DDF priority. For the WLM setup, ensure that the WLM address spaces get the default started task priority so that they can do their system administrative work quickly. The rest of the stored procedure priority is controlled by the way that you set up your service classes for the regular DB2 threads.
7	How can I improve the performance of nested stored procedures? How can I force them to run in the same WLM environment?	For better performance of nested stored procedures, and forcing them to run in the same WLM environment you can use: `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 proc. 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 proc that provide access to specific data sets. There is a wide variety of other possible dependencies for a particular WLM proc. 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. It can actually hinder performance, see “Grouping of stored procedures within application environments” on page 315.
8	I have a stored procedure with some complex SQL that is running for a long time. How do I analyze the access paths?	Use EXPLAIN as you would for any other application program with SQL. You have a variety of tools available, see the *Application Programming and SQL Guide*.
9	If the approximate cost of an SQL CALL is about 30,000 instructions, what is the approximate cost of a CALL to a COBOL subroutine?	COBOL calls are substantially cheaper than SQL CALLs The actual number of instructions is product-sensitive and can change with each release.

A.3 Administration FAQs

Table A-3 contains a list of frequently asked questions that deal with the administration of stored procedures.

Table A-3 Administration FAQs

No.	Question	Answer
1	Are there significant differences in the creation and usage of TCBs between DB2-managed stored procedures and WLM-managed stored procedures?	There is not much of a difference in the way TCBs are created and used between DB2-managed and WLM-managed address spaces. Tasks that terminate abnormally are replaced in both environments.
2	Are there differences in how TCBs are created and used in WLM-managed address spaces between Versions 6, 7, and 8 of DB2?	There is no difference between DB2 V6 and V7 in the area of TCB creation and use. In V8 we will exploit WLM server task management, which means that server tasks may be created and detached while the address space stays up. For more information on WLM server task management see 22.3.2, “Exploit WLM server task thread management” on page 354.
3	When granting authority to run a stored procedure, does execute authority need to be granted to both the stored procedure object and the package associated with the procedure, or just the stored procedure object?	If the owner of the stored procedure has execute authority on the SQL package for the procedure, then this authority does not need to be granted to the user ID calling the stored procedure. If the owner of the stored procedure does not have execute authority on the SQL package, then execute authority on both the procedure and the SQL package must be granted to the user ID calling the stored procedure. For more information on the security issues for stored procedures, see Chapter9, “Security and authorization” on page 51 of this redbook.
4	The WLM stored procedure address spaces shut down after one hour of inactivity at our shop. Why does this happen? Is there a way to have them stay up and avoid the overhead of startup when someone runs a stored procedure?	WLM shuts down the address spaces to avoid using system resources for nothing. All but the last one for an application environment/service class will be shut down after around 5 minutes of inactivity, with the last one shut down after around an hour of inactivity. The expectation is that stored procedure requests that require high performance will execute often enough to keep the address space active. There is no way to override this behavior for DB2 server address spaces.
5	When is stop/start needed? When is wlm/refresh needed? Does it make a difference whether the stored procedure is invoked by a server application, distributed application or by a trigger?	In general, only a WLM refresh is necessary. See 8.1, “Refreshing the stored procedure environment” on page 68 for a discussion of this topic.
6	We have a critical stored procedure that is executed very frequently. In the unlikely case where it fails, it changed to a STOBABND status and all subsequent executions also fail. Is there a way to control how many abends I can tolerate?	DB2 for OS/390 and z/OS Version 7 allows you to control how many abends you can tolerate on a DB2 subsystem level. DB2 for z/OS Version 8 also allows you to control how many abends you can tolerate for each specific stored procedure. See 8.4, “Handling application failures” on page 71 for a more detailed discussion of this topic.
7	I cannot browse message files that were written by a stored procedure. Why are they still in use after the stored procedure completes execution?	The message files are in use because they are defined in the JCL for the stored procedures address space, and they are shared by all procedures that run in this address space. The way to free the message files depends on the options specified when allocating the files through JCL, and involve STOP procedure DB2 command, REFRESH, and WLM QUIESCE commands.
8	How do I cancel threads that are executing stored procedures?	See 8.3, “Terminating hanging or looping stored procedures” on page 70 for details.

No.	Question	Answer
9	Will I be able to continue to run my DB2-managed stored procedures in DB2 for z/OS R810?	Yes. DB2-managed address space stored procedures will still exist in DB2 for z/OS R810, so existing DB2-managed stored procedures will continue to run. However, only WLM-managed stored procedures will be allowed to be created in DB2 for z/OS R810. Please consider converting your DB2-managed stored procedures while you are still running on R710, and creating any new stored procedures as WLM-managed.
10	When you define a WLM application environment, you can choose to have WLM start an unlimited number of address spaces or just one address space. I limit the number of stored procedures in an address space to 20, but at peak time, because of transaction volumes, over 1000 stored procedures try to start at once. How do I prevent WLM from starting 50 address spaces and essentially taking over my processor trying to run stored procedures?	Your WLM service policy can help prevent against this. If you have very aggressive goals in your WLM service policy, then WLM will continue to start address spaces because the goals are not being met. If your goals are more reasonable, then it is less likely that WLM will have to start more address spaces. See Chapter 4, “Setting up and managing Workload Manager” on page 33, and Chapter 20, “Server address space management” on page 319 for more details on managing WLM address spaces.

A.4 Miscellaneous FAQs

Table A-4 contains a list of other miscellaneous frequently asked questions about stored procedures.

Table A-4 Miscellaneous FAQs

No.	Question	Answer
1	I am getting a -440 SQLCODE when running a stored procedure. Where do I start?	This can be caused by a variety of reasons including the not-so-obvious security issue. See Chapter 14, “Debugging” on page 173 for details. DB2 was changed to issue -551 for authorization errors with APAR PQ54847.
2	I am getting an SQLCODE -723 when updating a table that has a trigger defined on it. Where do I look?	AN ERROR OCCURRED IN A TRIGGERED SQL STATEMENT IN trigger-name. INFORMATION RETURNED: SQLCODE: sqlerror, SQLSTATE: sqlstate, MESSAGE TOKENS token-list, SECTION NUMBER section-number Response: First, read the error message carefully for the section number associated with the failing triggered SQL statement. Then, locate the CREATE TRIGGER statement: ► For triggers that contain a WHEN clause, the WHEN clause is section number one. ► The triggered SQL statements are numbered sequentially, beginning with section number two. This is true for triggers with or without a WHEN clause. If the statement in error is a CALL statement (meaning the error is inside the stored procedure), note the sqlerror and see Chapter 14, “Debugging” on page 173 for details. Otherwise, see *DB2 UDB for z/OS Version 8 Messages and Codes*, GC18-7422.

No.	Question	Answer
3	My DSNUTILS or DSNUTILU stored procedure is trying to restart a utility. It is encountering allocation failures for template-defined data sets. Why?	This failure can occur for those template-defined data sets that were open when the utility terminated. In this case, the deallocation of the data set during the termination processing failed, and the data set is still allocated. The only workaround in this case is to stop and restart the WLM-managed address space for the data set allocation.
4	How do I fix an SQLCODE -471 with reason code 00E79002 on a stored procedure or user-defined function (UDF) call	See Chapter 14, “Debugging” on page 173 for details.
5	I encounter problems when multiple users build SQL procedures concurrently. How do I resolve this problem?	This problem is most likely due to an incorrect setup of the stored procedure builder service routine DSNTPSMP. Built-in stored procedure DSNTPSMP must have a separate WLM application environment that is defined as NUMTCB=1. This application environment must not be used to run any user stored procedures, only DSNTPSMP.
6	What is DSNTPSMP?	See 29.1, “Development Center start up” on page 500 for details.

B

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 25, “DB2-supplied stored procedures” on page 403.

This appendix details of the invocation of stored procedures providing the following functions:

- Display DB2 system information with DB2SystemInformation
- Refresh a WLM environment with DB2WLMRefresh
- Query the USS User DB with DB2USSUserInformation
- Issue DB2 commands with DB2Command
- Automate RUNSTATS with DB2Runstats
- Manage data sets with DB2DatasetUtilities
- Submit JCL with DB2JCLUtilities
- Issue USS commands with DB2USSCommand

The structure of the sample programs is very similar across the eight functions. To familiarize yourself, you should read through Appendix B.1, “Display DB2 system information with DB2SystemInformation” on page 596 where the sample program structure is described in detail. The other samples focus on calling the DB2-supplied stored procedures and do not repeat this information.

B.1 Display DB2 system information with DB2SystemInformation

DB2SystemInformation displays the MVS subsystem name, the fully qualified domain name, the zparms, 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 zparm, DSNACCSS 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** command in DB2 for z/OS Version 7 or higher, DSNACCSI is easier to use, and does not require any special privileges such as DSNACCMD.

While the zparms 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 zparms in any application program that you write. For example if you use dynamic SQL and your database object names require you to enclose them 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 DSNUTILS to run the DIAGNOSE online utility to find out which licensed utilities are installed.

Example B-1 lists the source code for the DB2SystemInformation class where we need to import the required Java packages and classes first.

Example: B-1 DB2SystemInformation class

```
//*************************************************************************
// Licensed Materials - Property of IBM
//
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*************************************************************************
// Source file name: DB2SystemInformation.java
//
// Sample: How to use the DB2 provided system information stored
//         procedures
```

```
//
// The user runs the program by issuing:
//   java DB2SystemInformation <alias> <userid> <password>
//
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//*********************************************************************
import java.sql.*;
```

In Example B-2 we define a bitmask for every DB2 utility. We will use these bitmasks later when we display the installed licensed utilities.

Example: B-2 Defining a bitmask for each utility

```
public class DB2SystemInformation
{
  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;
```

Most of our sample applications only contain a single main() method and an exception class, which is thrown to indicate a program error rather then a system error such as a SQL error. The main() method checks if all the required arguments have been passed to the program. In case of an error, a usage message is displayed as listed in Example B-3.

Example: B-3 Errors on arguments verification

```
public static void main(String args[])
  {
    Connection con = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
```

```
    {
      System.err.println("Usage: DB2SystemInformation <alias> <userid> <password>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid 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 appropriate type 2 driver for applications COM.ibm.db2.jdbc.app.DB2Driver as shown in Example B-4. To use a type 4 driver, we would have to specify COM.ibm.db2.jcc.DB2Driver and add the port number in our connection URL. For more information on using JDBC drivers in your application, see the *DB2 UDB V8 Application Development Guide: Programming Client Applications,* SC09-4826. A number of exceptions can be thrown here. The most likely problem is that the db2java.zip archive that contains the type 2 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: B-4 Load and connect with type 2 driver for COM.ibm.db2.jdbc.app.DB2Driver

```
// Load type 2 JDBC driver
    try
    {
      Class.forName(driver).newInstance();
    }
    catch(Exception e)
    {
      System.err.println("Failed to load current driver: " + driver);
      return;
    }

    try
    {
      int rc = 0;
      String message = null;
      boolean hasResultSet = false;

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
```

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 B-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. DSNACCSS has no IN parameters and three OUT parameters, which we register and then call execute to call the stored procedure.

Example: B-5 Preparing the CallableStatement

```
// Query SSID
      cs = con.prepareCall("CALL SYSPROC.DSNACCSS(?, ?, ?)");
      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 DSNACCSS 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 DB2SystemInformation 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 shown in Example B-6.

Example: B-6 Error handling within the procedure

```
// Obtain the return code
      rc = cs.getInt(2);
      if (rc > 0)
      {
      message = cs.getString(3);
        throw new DB2SystemInformationException(rc, "DSNACCSS execution failed: " +
message);
      }
      else
      {
        String sSSID = cs.getString(1);
        System.out.println("SSID = " + sSSID);
        cs.close();
      }
```

The call to DSNACCSI to retrieve the fully qualified domain name, shown in Example B-7, is almost identical to DSNACCSS.

Example: B-7 Retrieving the domain name

```
// Query FQDN
      cs = con.prepareCall("CALL SYSPROC.DSNACCSI(?, ?, ?)");
      cs.registerOutParameter(1, Types.VARCHAR); // FQDN
      cs.registerOutParameter(2, Types.INTEGER); // Return code
      cs.registerOutParameter(3, Types.VARCHAR); // Message area
      cs.execute();

      // Obtain the return code
      rc = cs.getInt(2);
      if (rc > 0)
      {
      message = cs.getString(3);
        throw new DB2SystemInformationException(rc, "DSNACCSI execution failed: " +
message);
      }
      else
      {
        String fqdn = cs.getString(1);
        System.out.println("FQDN = " + fqdn);
        cs.close();
      }
```

We use DSNWZP to retrieve the zparms 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 B-8.

Example: B-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]");
      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 DSNUTILS to determine which licensed utilities are installed on the connected subsystem. This is shown in Example B-9. If you are familiar with running online utilities in batch using DSNUTILB or DSNUPROC, you will find it very easy to use DSNUTILS. DSNUTILS has three parameters for every DDNAME you can use with online utilities, because it dynamically allocates these data sets. You can specify a device type, the primary space in cylinders, and the cataloged data set name. The DIAGNOSE utility does not require any data sets, so we can set the IN parameters to blank or 0 as required.

Example: B-9 Running DIAGNOSE through DSNUTILS

```
// Query installed utilities
      String map = null;
      cs = con.prepareCall("CALL SYSPROC.DSNUTILS(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?," +
                                                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?," +
                                                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
      cs.setString(1, "DB2SYSINFO");                 // Utility ID
      cs.setString(2, "NO");                         // Restart
      cs.setString(3, "DIAGNOSE DISPLAY AVAILABLE"); // Utility statement
      cs.registerOutParameter(4, Types.INTEGER);     // Return code
      cs.setString(5, "DIAGNOSE");                   // Utility name
      cs.setString(6, "");                           // Data sets
      cs.setString(7, "");
      cs.setShort(8, (short) 0);
      cs.setString(9, "");
      cs.setString(10, "");
      cs.setShort(11, (short) 0);
      cs.setString(12, "");
      cs.setString(13, "");
```

```
      cs.setShort(14, (short) 0);
      cs.setString(15, "");
      cs.setString(16, "");
      cs.setShort(17, (short) 0);
      cs.setString(18, "");
      cs.setString(19, "");
      cs.setShort(20, (short) 0);
      cs.setString(21, "");
      cs.setString(22, "");
      cs.setShort(23, (short) 0);
      cs.setString(24, "");
      cs.setString(25, "");
      cs.setShort(26, (short) 0);
      cs.setString(27, "");
      cs.setString(28, "");
      cs.setShort(29, (short) 0);
      cs.setString(30, "");
      cs.setString(31, "");
      cs.setShort(32, (short) 0);
      cs.setString(33, "");
      cs.setString(34, "");
      cs.setShort(35, (short) 0);
      cs.setString(36, "");
      cs.setString(37, "");
      cs.setShort(38, (short) 0);
      cs.setString(39, "");
      cs.setString(40, "");
      cs.setShort(41, (short) 0);
      cs.execute();

      // Obtain the return code
      hasResultSet =
cs.execute();
```

The error handling of stored procedures that return error messages in a result set cursor is different from 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 loose any results. The output from DIAGNOSE DISPLAY AVAILABLE looks similar to Figure B-1.

```
DSNU050I    DSNUGUTC -  DIAGNOSE DISPLAY AVAILABLE
DSNU862I    DSNUDIAG - DISPLAY AVAILABLE UTILITIES.  MAP: 11111111111111111111100000000
-------------------------------------------------------------------------------
!CATMAINT !CHECK    !COPY     !DIAGNOSE !LISTDEF  !LOAD     !MERGECOPY!MODIFY
!
!OPTIONS  !QUIESCE  !REBUILD  !RECOVER  !REORG    !REPAIR   !REPORT   !RUNSTATS
!
!STOSPACE !TEMPLATE !UNLOAD   !COPYTOCOP!EXEC     !         !         !
!
!         !         !         !         !         !         !         !
!
-------------------------------------------------------------------------------
DSNU860I    DSNUGUTC - DIAGNOSE UTILITY COMPLETE
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

Figure B-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 the message number DSNU8621 and store the map string for later processing as shown in Example B-10.

Example: B-10 Parsing DSNU8621

```
if (hasResultSet)
      {
        rs = cs.getResultSet();

        while (rs.next())
        {
          rs = cs.getResultSet();
          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 DSNUTILS and the DIAGNOSE utility, refer to the *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427. 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 B-11.

Example: B-11 Displaying the installed utilities

```
rc = cs.getInt(4);
      if (map == null || rc > 4)
      {
```

```
        throw new DB2SystemInformationException(rc, "DSNUTILS 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));
        cs.close();
      }
    }
```

In order to provide meaningful error messages, we catch our application defined exception DB2SystemInformationException and SQLException. This is good practice so that we do not loose 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 B-12.

Example: B-12 The finally block code

```
catch (DB2SystemInformationException sie)
    {
      System.out.println("Program error: rc=" + sie.getRC() + " message=" +
sie.getMessage());
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
```

```
    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) {}
    }
  }
}
```

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 B-13.

Example: B-13 Setting and getting the return code

```
class DB2SystemInformationException extends Exception
{
  private int rc;

  DB2SystemInformationException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile DB2SystemInformation and enter the following command to execute it:

```
java DB2SystemInformation DBALIAS USERID PASSWORD
```

You should get the response shown in Figure B-14.

Example: B-14 Output from DSNWZP

```
SSID = V81A
FQDN = v33ec060.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

:
```

```
 DSNU050I    DSNUGUTC -  DIAGNOSE DISPLAY AVAILABLE
 DSNU862I    DSNUDIAG - DISPLAY AVAILABLE UTILITIES.  MAP: 11111111111111111111000000000000
 ---------------------------------------------------------------------------------
 !CATMAINT !CHECK    !COPY     !DIAGNOSE !LISTDEF  !LOAD     !MERGECOPY!MODIFY
 !
 !OPTIONS  !QUIESCE  !REBUILD  !RECOVER  !REORG    !REPAIR   !REPORT   !RUNSTATS
 !
 !STOSPACE !TEMPLATE !UNLOAD   !COPYTOCOP!EXEC     !         !         !
 !
 !         !         !         !         !         !         !         !
 !
 ---------------------------------------------------------------------------------
 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:
```

B.2 Refresh a WLM environment with DB2WLMRefresh

DB2WLMRefresh 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 for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415 describes WLM_REFRESH and the *DB2 UDB for z/OS Version 8 Installation Guide,* GC18-7418 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 a 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 B-15 lists the source code for the DB2WLMRefresh class.

Example: B-15 DB2WLMRefresh source code

```
//*********************************************************************
// Licensed Materials - Property of IBM
//
```

```
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*************************************************************************
// Source file name: DB2WLMRefresh.java
//
// Sample: How to use the DB2 provided system information stored
//         procedures
//
// The user runs the program by issuing:
//   java DB2WLMRefresh <alias> <userid> <password>
//
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//*************************************************************************
import java.sql.*;

public class DB2WLMRefresh
{
  public static void main(String args[])
  {
    Connection con = null;
    CallableStatement cs = null;
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: DB2WLMRefresh <alias> <userid> <password>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    // Load type 2 JDBC driver
    try
    {
      Class.forName(driver).newInstance();
    }
    catch(Exception e)
    {
      System.err.println("Failed to load current driver: " + driver);
      return;
    }

    try
```

```
    {
      int rc = 0;
      String message = null;

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);

      // Refresh WLM application environment
      cs = con.prepareCall("CALL SYSPROC.WLM_REFRESH(?, ?, ?, ?)");
      cs.setString(1, "WLMENV1");
      cs.setString(2, "DSN");
      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 DB2WLMRefreshException(rc, "WLM_REFRESH execution failed: " + message);
      else
      {
        System.out.println("WLM refresh successful:" + message);
        cs.close();
      }
    }
    catch (DB2WLMRefreshException wre)
    {
      System.out.println("Program error: rc=" + wre.getRC() + " message=" +
wre.getMessage());
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
    catch (Exception e)
    {
      System.out.println("Error: message=" + e.getMessage());
    }
    finally
    {
      // Release resources and disconnect
      try { cs.close(); }
      catch(Exception e) {}

      try { con.close(); }
      catch(Exception e) {}
    }
  }
}

class DB2WLMRefreshException extends Exception
{
  private int rc;

  DB2WLMRefreshException(int rc, String message)
  {
    super(message);
    this.rc = rc;
```

```
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile DB2WLMRefresh and enter the following command to execute it:

```
java DB2WLMRefresh DBALIAS USERID PASSWORD
```

You should get the response:

```
WLM refresh successful:DSNT540I  WLMENV1 WAS REFRESHED BY PAOLOR8 USING AUTHORITY FROM
SQL ID PAOLOR8
```

B.3 Query the USS User DB with DB2USSUserInformation

You can query the USS User Database with the DB2USSUserInformation procedure shown in Example B-16.

Example: B-16 DSNAICUG invocation

```
//*************************************************************************
// Licensed Materials - Property of IBM
//
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*************************************************************************
// Source file name: DB2USSUserInformation.java
//
// Sample: How to use the DB2 provided system user-defined function
//         DSNAICUG to query the USS user database
//
// The user runs the program by issuing:
//   java DB2USSUserInformation <alias> <userid> <password>
//
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//*************************************************************************
import java.sql.*;

public class DB2USSUserInformation
{
  public static void main(String args[])
  {
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
```

```
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: DB2USSUserInformation <alias> <userid> <password>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    // Load type 2 JDBC driver
    try
    {
      Class.forName(driver).newInstance();
    }
    catch(Exception e)
    {
      System.err.println("Failed to load current driver: " + driver);
      return;
    }

    try
    {
      // Connect to database
      con = DriverManager.getConnection(url, userid, password);

      // List all groups
      ps = con.prepareStatement("SELECT NAME FROM TABLE(ICM.USER_GROUPS(?, ?)) AS T");
      ps.setInt(1, 0);
      ps.setString(2, "");
      rs = ps.executeQuery();
      while (rs.next())
        System.out.println("Group: " + rs.getString(1).trim());
      rs.close();

      // List all users
      ps.setInt(1, 1);
      ps.setString(2, "");
      rs = ps.executeQuery();
      while (rs.next())
        System.out.println("User: " + rs.getString(1).trim());
      rs.close();
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
    catch (Exception e)
    {
      System.out.println("Error: message=" + e.getMessage());
    }
```

```
    finally
    {
      // Release resources and disconnect
      try { rs.close(); }
      catch(Exception e) {}

      try { ps.close(); }
      catch(Exception e) {}

      try { con.close(); }
      catch(Exception e) {}
    }
  }
}
```

Compile DB2USSUserInformation.java and enter the following command to execute it:

```
java DB2USSUserInformation DBALIAS USERID PASSWORD
```

You should receive the response shown in Example B-17.

Example: B-17 Response to command

```
Group: CBADMGP
Group: OEDFLTG
Group: OEGRP10
Group: OEGRP11
Group: OMVSGRP
:
:
User: BPXROOT
User: OEDFLTU
:
:
User: WEBSRV
```

B.4 Issue DB2 commands with DB2Command

DB2Command shows how to use DSNACCMD to issue the following DB2 commands:

```
-DISPLAY ARCHIVE
-DISPLAY BUFFERPOOL
-DISPLAY DATABASE
-DISPLAY THREAD
-DISPLAY UTILITY
```

Except for the **-DISPLAY ARCHIVE** command, the corresponding parse type is specified. The sample program also queries the version of DB2 to accommodate for differences in the buffer pool table.

Example B-18 shows the source code for the CallDSNACCSI class.

Example: B-18 DB2Command class

```
//*************************************************************************
// Licensed Materials - Property of IBM
//
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
```

```
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*****************************************************************************
// Source file name: DB2Command.java
//
// Sample: How to use the DB2 provided stored procedure DSNACCMD
//
// The user runs the program by issuing:
//   java DB2Command <alias> <userid> <password>
//
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//*****************************************************************************
import java.sql.*;
import java.util.*;

public class DB2Command
{
  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";
  private static final int COMMAND_DID_NOT_COMPLETE = 15075360;

  public static void main(String args[])
  {
    Connection con = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;
    int db2release = 0;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: DB2Command <alias> <userid> <password>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    // Load type 2 JDBC driver
    try
    {
```

```
  Class.forName(driver).newInstance();
}
catch(Exception e)
{
  System.err.println("Failed to load current driver: " + driver);
  return;
}

try
{
  int rc = 0;
  String message = null;
  boolean hasResultSet = false;
  ArrayList messageText = new ArrayList();
  String commandArea = null;
  String parseType = null;

  // Connect to database
  con = DriverManager.getConnection(url, userid, password);
  DatabaseMetaData databaseMetaData = con.getMetaData();
  String[] version = databaseMetaData.getDatabaseProductVersion().split("\\.");
  db2release = Integer.parseInt(version[0]);

  // Issue DB2 commands
  cs = con.prepareCall("CALL SYSPROC.DSNACCMD(?, ?, ?, ?, ?, ?, ?, ?)");
  cs.registerOutParameter(4, Types.INTEGER); // Number of commands executed
  cs.registerOutParameter(5, Types.INTEGER); // IFI return code
  cs.registerOutParameter(6, Types.INTEGER); // IFI reason code
  cs.registerOutParameter(7, Types.INTEGER); // IFI excess bytes
  cs.registerOutParameter(8, 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
  rs = executeCommand(cs, messageText);
  parseResponse(parseType, db2release, rs, messageText);
  if (rs != null)
    rs.close();

  commandArea = "-DISPLAY BUFFERPOOL (*)";
  parseType = PARSE_BP;
  cs.setString(1, commandArea);
  cs.setInt(2, commandArea.length());
  cs.setString(3, parseType);
  rs = executeCommand(cs, messageText);
  parseResponse(parseType, db2release, 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);
  rs = executeCommand(cs, messageText);
  parseResponse(parseType, db2release, rs, messageText);
  // Close result set
  if (rs != null)
    rs.close();
```

```
      commandArea = "-DISPLAY THREAD (*)";
      parseType = PARSE_THD;
      cs.setString(1, commandArea);
      cs.setInt(2, commandArea.length());
      cs.setString(3, parseType);
      rs = executeCommand(cs, messageText);
      parseResponse(parseType, db2release, 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);
      rs = executeCommand(cs, messageText);
      parseResponse(parseType, db2release, rs, messageText);
      if (rs != null)
        rs.close();
      cs.close();
    }
    catch (IFIException ie)
    {
      System.out.println("Program error message=" + ie.getMessage());
      System.out.println("IFI return code=" + ie.getIFIReturnCode());
      System.out.println("IFI reason code=" + ie.getIFIReasonCode());
      System.out.println("IFI excess bytes=" + ie.getIFIExcessBytes());
      System.out.println("IFI message text=" + ie.getIFIMessageText());
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
    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 boolean isIFIError(int IFIReasonCode, int IFIExcessBytes, ArrayList
messageText)
  {
    boolean isError = true;

    if (IFIReasonCode == COMMAND_DID_NOT_COMPLETE && IFIExcessBytes > 0)
```

```
    {
      String message = ("MESSAGE LIMIT EXCEEDED. DISPLAY IS TERMINATED.");
      messageText.add(message);
      isError = false;
    }
    else
    {
      int size = messageText.size();
      if (size > 1)
      {
        String str = (String)messageText.get((size - 2));

        // Message limit exceeded
        if (str.startsWith("DSNT311I"))
          isError = false;
      }
    }

    return isError;
  }

  private static ResultSet executeCommand(CallableStatement cs, ArrayList messageText)
throws Exception
  {
    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(rs.getString(2).trim());

      if (cs.getMoreResults())
        rs = cs.getResultSet();
      else
        rs = null;
    }

    int execute_count = cs.getInt(4);
    int IFIReturnCode = cs.getInt(5);
    int IFIReasonCode = cs.getInt(6);
    int IFIExcessBytes = cs.getInt(7);
    String IFIMessageText = cs.getString(8);
    if (IFIMessageText != null)
      hasErrorMessage = true;

    if (IFIReturnCode != 0 || !hasResultSet)
    {
      if (hasErrorMessage)
        messageText.add(IFIMessageText);
      else if (messageText.isEmpty())
        messageText.add("No results returned.");

      if (isIFIError(IFIReasonCode, IFIExcessBytes, messageText))
      {
```

```
          String message = "IFI command execution failed.";
          throw new IFIException(IFIReturnCode, IFIReasonCode, IFIExcessBytes,
IFIMessageText, message);
       }
     }

     return rs;
   }

   private static void parseResponse(String parseType, int db2release, ResultSet rs,
ArrayList messageText) throws SQLException
   {
     if (rs != null)
     {
       while (rs.next())
       {
         if (parseType.equals(PARSE_BP))
         {
           if (db2release < 8)
           {
             System.out.println("NAME      = " + rs.getString(2).trim());
             System.out.println("CASTOUT   = " + rs.getString(3).trim());
             System.out.println("VPSIZE    = " + rs.getInt(4));
             System.out.println("HPSIZE    = " + rs.getInt(5));
             System.out.println("VPSEQT    = " + rs.getInt(6));
             System.out.println("VPPSEQT   = " + rs.getInt(7));
             System.out.println("VPXPSEQT  = " + rs.getInt(8));
             System.out.println("HPSEQT    = " + rs.getInt(9));
             System.out.println("DWQT      = " + rs.getInt(10));
             System.out.println("PCT VDWQT = " + rs.getInt(11));
             System.out.println("ABS VDWQT = " + rs.getInt(12));
             System.out.println("VPTYPE    = " + rs.getString(13).trim());
             System.out.println("PGSTEAL   = " + rs.getString(14).trim());
             System.out.println("ID        = " + rs.getInt(15));
             System.out.println("USECOUNT  = " + rs.getInt(16));
           }
           else
           {
             System.out.println("NAME      = " + 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("USECOUNT  = " + 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_CTR     = " + rs.getString(6).trim());
          System.out.println("CORR_ID     = " + rs.getString(7).trim());
          System.out.println("AUTH_ID     = " + rs.getString(8).trim());
          System.out.println("PNAME       = " + 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("LOCATION    = " + rs.getString(16).trim());
          System.out.println("DETAIL      = " + rs.getString(17).replace('\0',
'\n').trim());
        }
        else if (parseType.equals(PARSE_UT))
        {
          System.out.println("CSECT     = " + rs.getString(2).trim());
          System.out.println("USERID    = " + 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("NAME      = " + 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("NUMOBJ    = " + rs.getInt(12));
          System.out.println("LASTOBJ   = " + rs.getInt(13));
        }
      }
    }

    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 IFIException extends Exception
{
  private int IFIReturnCode;
  private int IFIReasonCode;
  private int IFIExcessBytes;
  private String IFIMessageText;

  IFIException (int IFIReturnCode, int IFIReasonCode, int IFIExcessBytes, String
IFIMessageText, String message)
  {
```

```
    super(message);
    this.IFIReturnCode = IFIReturnCode;
    this.IFIReasonCode = IFIReasonCode;
    this.IFIExcessBytes = IFIExcessBytes;
    this.IFIMessageText = IFIMessageText;
  }

  public int getIFIReturnCode()
  {
    return IFIReturnCode;
  }

  public int getIFIReasonCode()
  {
    return IFIReasonCode;
  }

  public int getIFIExcessBytes()
  {
    return IFIExcessBytes;
  }

  public String getIFIMessageText()
  {
    return IFIMessageText;
  }
}
```

Compile DB2Command.java and enter the following command to execute it:

```
java DB2Command DBALIAS USERID PASSWORD
```

You should receive the response shown in Example B-19.

Example: B-19 Response to command

```
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 TAPE ARCHIVE READING ACTIVITY.
END OF DISPLAY ARCHIVE REPORT.
DSN9022I  = DSNJCOO1 '-DISPLAY ARCHIVE' NORMAL COMPLETION
NAME      = BPO
VPSIZE    = 2000
VPSEQT    = 80
VPPSEQT   = 50
VPXPSEQT  = 0
DWQT      = 85
PCT VDWQT = 80
ABS VDWQT = 0
PGSTEAL   = LRU
ID        = 0
USECOUNT  = 123
PGFIX     = NO
NAME      = BP1
VPSIZE    = 500
```

```
VPSEQT    = 80
VPPSEQT   = 50
VPXPSEQT  = 0
DWQT      = 85
PCT VDWQT = 80
ABS VDWQT = 0
PGSTEAL   = LRU
ID        = 1
USECOUNT  = 3
PGFIX     = NO
:
:
DBNAME   = DSNDB06
SPACENAM =
TYPE     = DB
PART     = 0
STATUS   = RW
TYPE        = 1
NAME        = SERVER
STATUS      = SP
ACTIVE      = *
REQ_CTR     = 124
CORR_ID     = java.exe
AUTH_ID     = PAOLOR8
PNAME       = DISTSERV
ASID        = 0031
TOKEN       = 168
COORDINATOR =
RESET       =
URID        =
LUWID       =
LOCATION    =
DETAIL      = V437-WORKSTATION=PWANSCH, USERID=PAOLOR8,
      APPLICATION NAME=java.exe
 V429 CALLING
:
:
PROC=V81AWLM1, ASID=003A, WLM_ENV=WLMENV1
 V445-K27EAFD1.LF06.008D48235327=168 ACCESSING DATA FOR 9.65.96.110
DSNU112I  = DSNUGDIS - NO AUTHORIZED UTILITY FOUND FOR UTILID = *
```

B.5 Automate RUNSTATS with DB2Runstats

DB2Runstats runs RUNSTATS by exception. First, it calls DSNACCOR 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 DSNACCMO to execute RUNSTATS in parallel.

Example B-20 shows the source code for invoking RUNSTATS.

Example: B-20 Invoking RUNSTATS

```
//*****************************************************************************
// Licensed Materials - Property of IBM
//
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
```

```
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*************************************************************************
// Source file name: DB2Runstats.java
//
// Sample: How to use the DB2 provided stored procedures DSNACCOR and
//         DSNACCMO
//
// The user runs the program by issuing:
//   java DB2Runstats <alias> <userid> <password>
//
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//*************************************************************************
import java.sql.*;

public class DB2Runstats
{
  public static void main(String args[])
  {
    Connection con = null;
    PreparedStatement ps = null;
    PreparedStatement ps2 = null;
    PreparedStatement psType = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    ResultSet rsType = null;
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: DB2Runstats <alias> <userid> <password>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    // Load type 2 JDBC driver
    try
    {
      Class.forName(driver).newInstance();
    }
    catch(Exception e)
    {
      System.err.println("Failed to load current driver: " + driver);
      return;
```

```
}

try
{
  String QueryType = "RUNSTATS";
  String ObjectType = "TS";
  String ICType = "B";
  String StatsSchema = "SYSIBM";
  String CatlgSchema = "SYSIBM";
  String LocalSchema = "DSNACC";
  int ChkLvl = 1 + 2 + 4 + 8;
  String Criteria = "";
  String Unused = "";
  int CRUpdatedPagesPct = 20;
  int CRChangesPct = 10;
  int CRDaySncLastCopy = 7;
  int ICRUpdatedPagesPct = 1;
  int ICRChangesPct = 1;
  int CRIndexSize = 1;
  int RRTInsDelUpdPct = 20;
  int RRTUnclustInsPct = 10;
  int RRTDisorgLOBPct = 10;
  int RRTMassDelLimit = 0;
  int RRTIndRefLimit = 10;
  int RRIInsertDeletePct = 20;
  int RRIAppendInsertPct = 10;
  int RRIPseudoDeletePct = 10;
  int RRIMassDelLimit = 0;
  int RRILeafLimit = 10;
  int RRINumLevelsLimit = 0;
  int SRTInsDelUpdPct = 15;
  int SRTInsDelUpdAbs = 50;
  int SRTMassDelLimit = 0;
  int SRIInsDelUpdPct = 15;
  int SRIInsDelUpdAbs = 50;
  int SRIMassDelLimit = 0;
  int ExtentLimit = 2;
  String lastStatement = null;
  int IFCARetCode = 0;
  int IFCAResCode = 0;
  int XSBytes = 0;
  int rc = 0;
  String message = null;
  String errorMessage = null;
  boolean hasResultSet = false;

  // Connect to the database
  con = DriverManager.getConnection(url, userid, password);
  con.setAutoCommit(false);

  // Get RUNSTATS recommendations
  cs = con.prepareCall("CALL SYSPROC.DSNACCOR(" +
                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?" +
                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?" +
                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?" +
                  "?, ?, ?, ?, ?, ?, ?, ?, ?)");
  cs.setString(1, QueryType);
  cs.setString(2, ObjectType);
  cs.setString(3, ICType);
  cs.setString(4, StatsSchema);
```

```
        cs.setString(5, CatlgSchema);
        cs.setString(6, LocalSchema);
        cs.setInt(7, ChkLvl);
        cs.setString(8, Criteria);
        cs.setString(9, Unused);
        cs.setInt(10, CRUpdatedPagesPct);
        cs.setInt(11, CRChangesPct);
        cs.setInt(12, CRDaySncLastCopy);
        cs.setInt(13, ICRUpdatedPagesPct);
        cs.setInt(14, ICRChangesPct);
        cs.setInt(15, CRIndexSize);
        cs.setInt(16, RRTInsDelUpdPct);
        cs.setInt(17, RRTUnclustInsPct);
        cs.setInt(18, RRTDisorgLOBPct);
        cs.setInt(19, RRTMassDelLimit);
        cs.setInt(20, RRTIndRefLimit);
        cs.setInt(21, RRIInsertDeletePct);
        cs.setInt(22, RRIAppendInsertPct);
        cs.setInt(23, RRIPseudoDeletePct);
        cs.setInt(24, RRIMassDelLimit);
        cs.setInt(25, RRILeafLimit);
        cs.setInt(26, RRINumLevelsLimit);
        cs.setInt(27, SRTInsDelUpdPct);
        cs.setInt(28, SRTInsDelUpdAbs);
        cs.setInt(29, SRTMassDelLimit);
        cs.setInt(30, SRIInsDelUpdPct);
        cs.setInt(31, SRIInsDelUpdAbs);
        cs.setInt(32, SRIMassDelLimit);
        cs.setInt(33, ExtentLimit);
        cs.registerOutParameter(34, Types.VARCHAR); // Last statement
        cs.registerOutParameter(35, Types.INTEGER); // Return code
        cs.registerOutParameter(36, Types.VARCHAR); // Message area
        cs.registerOutParameter(37, Types.INTEGER); // IFI return code
        cs.registerOutParameter(38, Types.INTEGER); // IFI reason code
        cs.registerOutParameter(39, Types.INTEGER); // IFI excess bytes
        hasResultSet = cs.execute();
        con.commit();

        message = cs.getString(36);
        rc = cs.getInt(35);
        if (cs.wasNull())
          throw new DB2RunstatsException(rc, "DSNACCOR execution failed: " + message);

        if (rc > 4)
        {
          switch (rc)
          {
            case 8:
              errorMessage = "DSNACCOR execution failed: " + message;
            break;

          case 12:
              lastStatement = cs.getString(34);
            errorMessage = "DSNACCOR execution failed: " + message + " last statement: " +
lastStatement;
            break;

          case 14:
            errorMessage = "DSNACCOR cannot access real-time statistics tables: " + message;
            break;
```

```
        case 15:
          errorMessage = "DSNACCOR encountered problem with declared temporary tables: " +
message;
          break;

        case 16:
          errorMessage = "DSNACCOR was unable to define declared temporary tables: " +
message;
          break;
        }

        throw new DB2RunstatsException(rc, errorMessage);
      }
      else
        System.out.println("DSNACCOR message: " + message);

      if (hasResultSet)
      {
        rs = cs.getResultSet();
        while (rs.next())
          System.out.println("IFI message: " + rs.getString(2));
        rs.close();

        // Prepare the statements for the DSNACCMO parameter tables
        ps = con.prepareStatement("INSERT INTO DSNACC.MO_TBL VALUES (" +
                                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                                  "?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                                  "?, ?, ?, ?)");

        ps2 = con.prepareStatement("INSERT INTO DSNACC.MO_TBL2 VALUES (" +
                                   "?,?,?)");

        ps2 = con.prepareStatement("INSERT INTO DSNACC.MO_TBL2 VALUES (" +
                                   "?,?,?)");
        // Insert the utility commands
        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
        psType = con.prepareStatement("SELECT TYPE FROM SYSIBM.SYSDATABASE WHERE NAME = ?
AND TYPE NOT IN ('W', 'T')");

        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(5);
```

```
// 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
psType.setString(1, db);
rsType = psType.executeQuery();
if (!rsType.next())
  continue;
rsType.close();

// Insert the tablespace
ps.setInt(1, objects);                    // Object ID
ps.setInt(2, 0);                          // Statement ID
ps.setString(3, "TABLESPACE");            // Object type
ps.setString(4, db + "." + name);         // Object name
ps.setNull(5, Types.SMALLINT);            // Object partition
ps.setString(6, "NO");                    // Restart
ps.setString(7, "RUNSTATS TABLESPACE"); // Utility name
ps.setString(8, "N");                     // Use templates
ps.setString(9, "");                      // DSNUTILS data sets
ps.setString(10, "");
ps.setShort(11, (short) 0);
ps.setString(12, "");
ps.setString(13, "");
ps.setShort(14, (short) 0);
ps.setString(15, "");
ps.setString(16, "");
ps.setShort(17, (short) 0);
ps.setString(18, "");
ps.setString(19, "");
ps.setShort(20, (short) 0);
ps.setString(21, "");
ps.setString(22, "");
ps.setShort(23, (short) 0);
ps.setString(24, "");
ps.setString(25, "");
ps.setShort(26, (short) 0);
ps.setString(27, "");
ps.setString(28, "");
ps.setShort(29, (short) 0);
ps.setString(30, "");
ps.setString(31, "");
ps.setShort(32, (short) 0);
ps.setString(33, "");
ps.setString(34, "");
ps.setShort(35, (short) 0);
ps.setString(36, "");
ps.setString(37, "");
ps.setShort(38, (short) 0);
ps.setString(39, "");
ps.setString(40, "");
ps.setShort(41, (short) 0);
ps.setString(42, "");
ps.setString(43, "");
```

```
            ps.setShort(44, (short) 0);
            ps.executeUpdate();
            objects++;
          }
          ps.close();
          psType.close();
          rs.close();
          cs.close();

          if (objects > 0)
          {
            // Execute utilities in parallel
            cs = con.prepareCall("CALL SYSPROC.DSNACCMO(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?)");
            cs.setShort(1, (short) 4);                    // Maximum parallel subtasks
            cs.setString(2, "YES");                       // Optimize workload
            cs.setString(3, "ERROR");                     // Stop condition
            cs.setString(4, "NO");                        // Save restart info in Control
Center tables
            cs.setString(5, "RUNSTATS");                  // Utility ID stem
            cs.setInt(6, objects);                        // Number of objects
            cs.setNull(7, Types.FLOAT);                   // Shutdown
            cs.setString(8, "D");                         // Escape code
            cs.registerOutParameter(9, Types.INTEGER);    // Utilities executed
            cs.registerOutParameter(10, Types.INTEGER);   // Highest DSNUTILS return code
            cs.registerOutParameter(11, Types.SMALLINT); // Actual parallel substasks
            cs.registerOutParameter(12, Types.INTEGER);   // Return code
            cs.registerOutParameter(13, Types.VARCHAR);   // Message area
            hasResultSet = cs.execute();
            con.commit();

            if (hasResultSet)
            {
              rs = cs.getResultSet();
              while (rs.next())
              {
                System.out.println(" OID = " + rs.getInt(1) +
                                   " TEXT = " + rs.getString(3));
              }
              rs.close();
            }

            int highestDSNUTILSretCode = cs.getInt(10);
            rc = cs.getInt(12);
            message = cs.getString(13);
            if (rc > 4)
            {
             errorMessage = "DSNACCMO execution failed: " + message;
              throw new DB2RunstatsException(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 DB2RunstatsException(rc, errorMessage);
              }
```

```
            }
          }

          cs.close();
          System.out.println("DB2Runstats successful.");
        }
        else
        throw new DB2RunstatsException(rc, "DSNACCOR execution failed: no result set.");

      }
      else
        throw new DB2RunstatsException(rc, "DSNACCOR execution failed: no IFI command
result set.");
    }
    catch (DB2RunstatsException re)
    {
      System.out.println("Program error: rc=" + re.getRC() + " message=" +
re.getMessage());
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
    catch (Exception e)
    {
      System.out.println("Error: message=" + e.getMessage());
    }
    finally
    {
      // Release resources and disconnect
      try { ps.close(); }
      catch(Exception e) {}

      try { ps2.close(); }
      catch(Exception e) {}

      try { psType.close(); }
      catch(Exception e) {}

      try { rs.close(); }
      catch(Exception e) {}

      try { rsType.close(); }
      catch(Exception e) {}

      try { cs.close(); }
      catch(Exception e) {}

      try { con.close(); }
      catch(Exception e) {}
    }
  }
}

class DB2RunstatsException extends Exception
{
  private int rc;

  DB2RunstatsException(int rc, String message)
```

```
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

B.6 Manage data sets with DB2DatasetUtilities

DB2DatasetUtilties use DSNACCDS to create a PS data set named DSTEST using the callers user ID as HLQ. It then renames it to DSTEST2 using DSNACCDR. Next, it checks if it is cataloged using DSNACCDE, lists it using DSNACCDL, and finally deletes it using DSNACCDD.

Example B-21 shows how to use the stored procedures for data set manipulation.

Example: B-21 DB2DatasetUtilities

```
//*************************************************************************
// Licensed Materials - Property of IBM
//
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*************************************************************************
// Source file name: DB2DatasetUtilities.java
//
// Sample: How to use the DB2 provided data set manipulation stored
//         procedures
//
// The user runs the program by issuing:
//   java DB2DatasetUtilities <alias> <userid> <password>
//
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//*************************************************************************
import java.sql.*;

public class DB2DatasetUtilities
{
  public static void main(String args[])
  {
    Connection con = null;
    PreparedStatement ps = null;
    CallableStatement cs = null;
    ResultSet rs = null;
```

```
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: java DB2DatasetUtilities <alias> <userid> <password>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    // Load type 2 JDBC driver
    try
    {
      Class.forName(driver).newInstance();
    }
    catch(Exception e)
    {
      System.err.println("Failed to load current driver: " + driver);
      return;
    }

    try
    {
      String dsName = userid + ".DSTEST";
      String dsName2 = userid + ".DSTEST2";
      int rc = 0;
      String message = null;
      boolean hasResultSet = false;

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      // Create a data set
      ps = con.prepareStatement("INSERT INTO DSNACC.DSNRECORDS(UTIL_SEQNBR, UTIL_RECORD)
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.DSNACCDS(?, ?, ?, ?, ?, ?, ?)");
      cs.setInt(1, 1);                              // Parm level = 1
      cs.setString(2, dsName);                      // Data set name
      cs.setString(3, "");                          // Member or Generation # (+1, -1, 0,
+2)
      cs.setString(4, "ND");                        // Option (ND, NM, A, R)
      cs.registerOutParameter(5,Types.INTEGER);     // Return code
      cs.registerOutParameter(6,Types.LONGVARCHAR); // Message area
      cs.setString(7, "N");                         // No trace
```

```
      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(5);
      if (rc > 0)
      {
        message = cs.getString(6);
        throw new DB2DatasetUtilitiesException(rc, "DSNACCDS execution failed: " +
message);
      }
      else
      {
      ps.close();
      cs.close();
        System.out.println(dsName + " created.");
      }

      // Rename the data set
      cs = con.prepareCall("CALL SYSPROC.DSNACCDR(?, ?, ?, ?, ?, ?, ?, ?)");
      cs.setInt(1, 1);                              // Parm level = 1
      cs.setInt(2, 4);                              // Data set type
(1-pds,2-pdse,3-mbr,4-ps)
      cs.setString(3, dsName);                      // Data set name
      cs.setString(4, "");                          // Member name
      cs.setString(5, dsName2);                     // New name
      cs.registerOutParameter(6,Types.INTEGER);     // Return code
      cs.registerOutParameter(7,Types.LONGVARCHAR); // Messages area
      cs.setString(8, "N");                         // No trace
      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(6);
      if (rc > 0)
      {
        message = cs.getString(7);
        throw new DB2DatasetUtilitiesException(rc, "DSNACCDR execution failed: " +
message);
      }
      else
      {
      cs.close();
        System.out.println(dsName + " renamed to " + dsName2 + ".");
      }

      // Check if the data set exists
      cs = con.prepareCall("CALL SYSPROC.DSNACCDE(?, ?, ?, ?, ?, ?)");
      cs.setInt(1, 1);                              // Parm level = 1
      cs.setString(2, dsName2);                     // Data set name
      cs.setString(3, "");                          // Member
      cs.registerOutParameter(4,Types.INTEGER);     // Return code
      cs.registerOutParameter(5,Types.LONGVARCHAR); // Messages area
      cs.setString(6, "N");                         // No trace
      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(4);
      if (rc > 1)
```

```
      {
        message = cs.getString(5);
        throw new DB2DatasetUtilitiesException(rc, "DSNACCDE execution failed: " +
message);
      }
      else
      {
      cs.close();
        if (rc == 0)
          System.out.println(dsName2 + " exists.");
        else if (rc == 1)
          System.out.println(dsName2 + " does not exist.");
      }

      // List everything under the user HLQ
      cs = con.prepareCall("CALL SYSPROC.DSNACCDL(?, ?, ?, ?, ?, ?, ?, ?)");
      cs.setInt(1, 1);                                // Parm level = 1
      cs.setString(2, userid + ".**");                // Data set name or filter
      cs.setString(3, "N");                           // List members (Y, N)
      cs.setString(4, "N");                           // List generations (Y, N)
      cs.setInt(5, 50);                               // Maximum number of data sets returned
      cs.registerOutParameter(6,Types.INTEGER);       // Return code
      cs.registerOutParameter(7,Types.LONGVARCHAR); // Messages area
      cs.setString(8, "N");                           // No trace
      hasResultSet = cs.execute();

      // Obtain the return code
      rc = cs.getInt(6);
      if (rc > 0)
      {
        message = cs.getString(7);
        throw new DB2DatasetUtilitiesException(rc, "DSNACCDL 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("DS_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.getInt(10));
            System.out.println("DS_HARBA         = " + rs.getInt(11));
            System.out.println("DS_HURBA         = " + rs.getInt(12));

System.out.println("------------------------------------------------------------");
          }
        }
        rs.close();
```

```
        cs.close();
      }

      // Delete the data set
      cs = con.prepareCall("CALL SYSPROC.DSNACCDD(?, ?, ?, ?, ?, ?, ?)");
      cs.setInt(1, 1);                              // Parm level = 1
      cs.setInt(2, 4);                              // Data set type
(1-pds,2-pdse,3-mbr,4-ps)
      cs.setString(3, dsName2);                     // Data set name
      cs.setString(4, "");                          // Member name
      cs.registerOutParameter(5,Types.INTEGER);     // Return code
      cs.registerOutParameter(6,Types.LONGVARCHAR); // Messages area
      cs.setString(7, "N");                         // No trace
      cs.execute();
      con.commit();

      // Obtain the return code
      rc = cs.getInt(5);
      if (rc > 0)
      {
        message = cs.getString(6);
        throw new DB2DatasetUtilitiesException(rc, "DSNACCDD Execution failed: " +
message);
      }
      else
      {
      cs.close();
        System.out.println(dsName2 + " deleted.");
      }
    }
    catch (DB2DatasetUtilitiesException dsue)
    {
      System.out.println("Program error: rc=" + dsue.getRC() + " message=" +
dsue.getMessage());
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
    catch (Exception e)
    {
      System.out.println("Error: message=" + e.getMessage());
    }
    finally
    {
      // Release resources and disconnect
      try { rs.close(); }
      catch(Exception e) {}

      try { ps.close(); }
      catch(Exception e) {}

      try { cs.close(); }
      catch(Exception e) {}

      try { con.close(); }
      catch(Exception e) {}
    }
  }
```

```
}

class DB2DatasetUtilitiesException extends Exception
{
  private int rc;

  DB2DatasetUtilitiesException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile DB2DatasetUtilities.java and enter the following command to execute it:

```
java DB2DatasetUtilities DBALIAS USERID PASSWORD
```

You should receive the response shown in Example B-22.

Example: B-22 Response to command

```
PAOLOR8.DSTEST created.
PAOLOR8.DSTEST renamed to PAOLOR8.DSTEST2.
PAOLOR8.DSTEST2 exists.
--------------------------------------------------------------
DSNAME           = PAOLOR8.DCLGEN
CREATE_YEAR      = 1992
CREATE_DAY       = 91
DS_TYPE          = 1
VOLUME           = USER99
PRIMARY_EXTENT   = 1
SECONDARY_EXTENT = 8
MEASUREMENT_UNIT = TRKS
EXTENTS_IN_USE   = 1
DASD_USAGE       = 58786
DS_HARBA         = -1
DS_HURBA         = -1
--------------------------------------------------------------
--------------------------------------------------------------
DSNAME           = PAOLOR8.DSTEST2
CREATE_YEAR      = 2004
CREATE_DAY       = 18
DS_TYPE          = 4
VOLUME           = SCR05
PRIMARY_EXTENT   = 2
SECONDARY_EXTENT = 10
MEASUREMENT_UNIT = BLKS
EXTENTS_IN_USE   = 1
DASD_USAGE       = 58786
DS_HARBA         = -1
DS_HURBA         = -1
--------------------------------------------------------------
--------------------------------------------------------------
DSNAME           = PAOLOR8.SPFLOG1.LIST
CREATE_YEAR      = 2004
```

```
CREATE_DAY       = 18
DS_TYPE          = 4
VOLUME           = SCR05
PRIMARY_EXTENT   = 576
SECONDARY_EXTENT = 563
MEASUREMENT_UNIT = BLKS
EXTENTS_IN_USE   = 1
DASD_USAGE       = 470288
DS_HARBA         = -1
DS_HURBA         = -1
------------------------------------------------------------
------------------------------------------------------------
DSNAME           = PAOLOR8.SRCHFOR.LIST
CREATE_YEAR      = 2004
CREATE_DAY       = 18
DS_TYPE          = 4
VOLUME           = SCR05
PRIMARY_EXTENT   = 14
SECONDARY_EXTENT = 100
MEASUREMENT_UNIT = BLKS
EXTENTS_IN_USE   = 1
DASD_USAGE       = 58786
DS_HARBA         = -1
DS_HURBA         = -1
------------------------------------------------------------
PAOLOR8.DSTEST2 deleted.
```

B.7 Submit JCL with DB2JCLUtilities

DB2JCLUtilities use DSNACCJS to submit JCL to compress an existing PDS. Then it uses DSNACCJQ to poll the job status until the job is in the OUT queue. It then uses DSNACCJF to fetch the job output and prints it. Finally, it calls DSNACCJP to purge the job output.

Example B-23 shows how to submit JCL with DB2JCLUtilities.

Example: B-23 DB2JCLUtilities

```
//*************************************************************************
// Licensed Materials - Property of IBM
//
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*************************************************************************
// Source file name: DB2JCLUtilities.java
//
// Sample: How to use the DB2 provided JCL stored procedures
//
// The user runs the program by issuing:
//   java DB2JCLUtilities <alias> <userid> <password>
//
```

```
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//*************************************************************************
import java.sql.*;

public class DB2JCLUtilities
{
  public static void main(String[] args)
  {
    Connection con = null;
    PreparedStatement ps = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;

    // Parse arguments
    if (args.length != 3)
    {
      System.err.println("Usage: java DB2JCLUtilities <alias> <userid> <password>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid to connect as");
      System.err.println("       <password> is password to connect with");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];

    // Load type 2 JDBC driver
    try
    {
      Class.forName(driver).newInstance();
    }
    catch(Exception e)
    {
      System.err.println("Failed to load current driver: " + driver);
      return;
    }

    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;

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);
      con.setAutoCommit(false);

      // Submit JCL
      ps = con.prepareStatement("INSERT INTO DSNACC.JSRECORDS(JS_SEQUENCE, JS_TEXT)
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.DSNACCJS(?, ?, ?, ?, ?)");
      cs.setString(1, "PAOLOR8");                   // User ID
      cs.setString(2, "PAOLOPW");                   // 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 DB2JCLUtilitiesException(rc, "DSNACCJS 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.DSNACCJQ(?, ?, ?, ?, ?, ?, ?, ?)" );
      cs.setInt(1, 1);                              // Operation
      cs.setString(2, "PAOLOR8");                   // User ID
      cs.setString(3, "PAOLOPW");                   // Password
      cs.setString(4, jobid);                       // Job ID
      cs.setInt(5, 0);                              // Timeout
      cs.registerOutParameter(6,Types.INTEGER);     // Return code
      cs.registerOutParameter(7,Types.INTEGER);     // Job status
      cs.registerOutParameter(8,Types.LONGVARCHAR); // Message area

      while (true)
      {
        cs.execute();
        con.commit();

        // Obtain return code
        rc = cs.getInt(6);
        if (rc > 4)
        {
```

```
          message = cs.getString(8);
          throw new DB2JCLUtilitiesException(rc, "DSNACCJQ execution failed:" + message);
        }
        else
        {
          jobstatus = cs.getInt(7);

          if (rc == 0)
          {
            // The job is in the OUTPUT queue
            if (jobstatus == 3)
              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)
          {
            jobstatus = cs.getInt(7);
            message = cs.getString(8);
            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 DB2JCLUtilitiesException(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.DSNACCJF(?, ?, ?, ?, ?)");
      cs.setString(1, "PAOLOR8");                    // User ID
      cs.setString(2, "PAOLOPW");                    // 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 DB2JCLUtilitiesException(rc, "DSNACCJF execution failed: " + message);
        }
        else
        {
          if (hasResultSet)
          {
            rs = cs.getResultSet();
            while (rs.next())
            {
              String sLine = rs.getString(2);
              System.out.println(sLine);
            }
            rs.close();
          }
          else
            System.out.println("Job " + jobid + " has no output.");
          cs.close();
        }

        // Purge job output
        cs = con.prepareCall("CALL SYSPROC.DSNACCJP(?, ?, ?, ?, ?, ?, ?)");
        cs.setInt(1, 2);                              // Operation (Cancel=1, Purge=2)
        cs.setString(2, "PAOLOR8");                   // User ID
        cs.setString(3, "PAOLOPW");                   // Password
cs.setString(4, jobid);                         // Job ID
        cs.setInt(5, 1);                              // Timeout
        cs.registerOutParameter(6,Types.INTEGER);     // Return code
        cs.registerOutParameter(7,Types.LONGVARCHAR); // Message area
        cs.execute();
        con.commit();

        // Obtain return code
        rc = cs.getInt(6);
        if (rc > 4)
        {
          message = cs.getString(8);
          throw new DB2JCLUtilitiesException(rc, "DSNACCJP execution failed: " + message);
        }
        else
        {
          if (rc == 0)
            System.out.println("Job " + jobid + " has been purged.");
          else
          {
            message = cs.getString(7);
            int index;
            if ((index = message.indexOf("HASP890")) != -1 &&
            message.substring(index).indexOf("AWAITING PURGE") != -1)
              System.out.println("Job " + jobid + " is awaiting to be purged.");
            else if ((index = message.indexOf("HASP003")) != -1 &&
            message.substring(index).indexOf("NO SELECTABLE ENTRIES FOUND") != -1)
              System.out.println("Job " + jobid + " will be purged.");
```

```
          else
            System.out.println(message);
        }

        cs.close();
      }
    }
    catch (DB2JCLUtilitiesException jue)
    {
      System.out.println("Program error: rc=" + jue.getRC() + " message=" +
jue.getMessage());
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
    catch (Exception e)
    {
      System.out.println("Error: message=" + e.getMessage());
    }
    finally
    {
      // Release resources and disconnect
      try { rs.close(); }
      catch(Exception e) {}

      try { ps.close(); }
      catch(Exception e) {}

      try { cs.close(); }
      catch(Exception e) {}

      try { con.close(); }
      catch(Exception e) {}
    }
  }
}

class DB2JCLUtilitiesException extends Exception
{
  private int rc;

  DB2JCLUtilitiesException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile DB2JCLUtilities.java and enter the following command to execute it:

```
java DB2JCLUtilities DBALIAS USERID PASSWORD
```

You should get the response shown in Example B-24.

Example: B-24 Response to command

```
Job JOB00098 submitted successfully.
Job JOB00098 is in the ACTIVE queue. Waiting for job to finish...
Job JOB00098 is in the ACTIVE queue. Waiting for job to finish...
Job JOB00098 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

16.11.08 JOB00098 ---- SUNDAY,    18 JAN 2004 ----
16.11.08 JOB00098  IRR010I  USERID PAOLOR8 IS ASSIGNED TO THIS JOB.
16.11.08 JOB00098  ICH70001I PAOLOR8 LAST ACCESS AT 12:16:37 ON SUNDAY, JANUARY
 18, 2004
16.11.08 JOB00098  $HASP373 IEBCOPY  STARTED - INIT 4    - CLASS K - SYS STL0
16.11.09 JOB00098  SMF000I  IEBCOPY     COPY        IEBCOPY     0000
16.11.09 JOB00098  $HASP395 IEBCOPY  ENDED
------ JES2 JOB STATISTICS ------
  18 JAN 2004 JOB EXECUTION DATE
            6 CARDS READ
           50 SYSOUT PRINT RECORDS
            0 SYSOUT PUNCH RECORDS
            3 SYSOUT SPOOL KBYTES
         0.01 MINUTES EXECUTION TIME
        1 //IEBCOPY  JOB ,CLASS=K,MSGCLASS=H,MSGLEVEL=(1,1)
  JOB00098
        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 PAOLOR8  LAST ACCESS AT 12:16:37 ON SUNDAY, JANUARY 18, 2004
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   USER.TESTLIB                                KEPT
IEF285I   VOL SER NOS= USER01.
IEF285I   USER.TESTLIB                                KEPT
IEF285I   VOL SER NOS= USER01.
IEF285I   PAOLOR8.IEBCOPY.JOB00098.D0000102.?         SYSOUT
IEF285I   PAOLOR8.IEBCOPY.JOB00098.D0000101.?         SYSIN
IEF373I STEP/COPY    /START 2004018.1611
IEF374I STEP/COPY    /STOP  2004018.1611 CPU    0MIN 00.02SEC SRB    0MIN 00.01S
EC VIRT  1024K SYS   244K EXT       4K SYS   11020K
IEF375I  JOB/IEBCOPY /START 2004018.1611
IEF376I  JOB/IEBCOPY /STOP  2004018.1611 CPU    0MIN 00.02SEC SRB    0MIN 00.01S
EC
                                      IEBCOPY MESSAGES AND CONTROL STATEMENTS
                             PAGE     1
IEB1135I IEBCOPY  FMID HDZ11G0  SERVICE LEVEL UA05160  DATED 20030904 DFSMS 01.0
3.00 z/OS    01.03.00 HBB7706  CPU 4381
IEB1035I IEBCOPY   COPY     16:11:08 SUN 18 JAN 2004 PARM=''
COPY     COPY      INDD=SYSUT1,OUTDD=SYSUT2         GENERATED STATEMENT
IEB1018I COMPRESSING  PDS  OUTDD=SYSUT2   VOL=USER01 DSN=USER.TESTLIB
IEB152I XSNLNLLM COMPRESSED - WAS ALREADY IN PLACE AND NOT MOVED
IEB152I V81ADELS COMPRESSED - WAS ALREADY IN PLACE AND NOT MOVED
IEB152I PSNACCC1 COMPRESSED - WAS ALREADY IN PLACE AND NOT MOVED
IEB152I XSNLILLM COMPRESSED - WAS ALREADY IN PLACE AND NOT MOVED
IEB153I ALL MEMBERS COMPRESSED - ALL WERE ORIGINALLY COMPRESSED
```

```
IEB144I THERE ARE 1093 UNUSED TRACKS IN OUTPUT DATA SET REFERENCED BY SYSUT2
IEB147I END OF JOB - 0 WAS HIGHEST SEVERITY CODE
Job JOB00098 is awaiting to be purged.
```

B.8 Issue USS commands with DB2USSCommand

DB2USSCommand uses DSNACCUC to issue the following command:

ls -lat

Example B-25 shows how to submit USS commands through stored procedures.

Example: B-25 DB2USSCommand

```
//*************************************************************************
// Licensed Materials - Property of IBM
//
// Governed under the terms of the International
// License Agreement for Non-Warranted Sample Code.
//
// (C) COPYRIGHT International Business Machines Corp. 2003
// All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//*************************************************************************
// Source file name: DB2USSCommand.java
//
// Sample: How to use the DB2 provided stored procedure DSNACCUC
//
// The user runs the program by issuing:
//   java DB2USSCommand <alias> <userid> <password> <uss_userid>
//                      <uss_password> <uss_command>
//
// The arguments are:
//   <alias> - DB2 subsystem alias
//   <userid> - MVS userid to connect as
//   <password> - password to connect with
//   <uss_userid> - MVS userid with OMVS segment
//   <uss_password> - password of MVS userid with OMVS segment
//   <uss_command> - USS command to execute
//*************************************************************************
import java.sql.*;

public class DB2USSCommand
{
  public static void main(String args[])
  {
    Connection con = null;
    CallableStatement cs = null;
    ResultSet rs = null;
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = "jdbc:db2:";
    String userid = null;
    String password = null;
    String uss_userid = null;
    String uss_password = null;
    String uss_command = null;
```

```
    // Parse arguments
    if (args.length != 6)
    {
      System.err.println("Usage: DB2USSCommand <alias> <userid> <password> <uss_userid>
<uss_password> <uss_command>");
      System.err.println("where  <alias> is DB2 subsystem alias");
      System.err.println("       <userid> is MVS userid to connect as");
      System.err.println("       <password> is password to connect with");
      System.err.println("       <uss_userid> is MVS userid with OMVS segment");
      System.err.println("       <uss_password> is password of MVS userid with OMVS
segment");
      System.err.println("       <uss_command> is USS command to execute");
      return;
    }
    url += args[0];
    userid = args[1];
    password = args[2];
    uss_userid = args[3];
    uss_password = args[4];
    uss_command = args[5];

    // Load type 2 JDBC driver
    try
    {
      Class.forName(driver).newInstance();
    }
    catch(Exception e)
    {
      System.err.println("Failed to load current driver: " + driver);
      return;
    }

    try
    {
      int rc = 0;
      String message = null;
      boolean hasResultSet = false;

      // Connect to database
      con = DriverManager.getConnection(url, userid, password);

      // Execute USS command
      cs = con.prepareCall( "CALL SYSPROC.DSNACCUC(?, ?, ?, ?, ?, ?)" );
      cs.setString(1, uss_userid);                  // USS user ID
      cs.setString(2, uss_password);                // USS password
      cs.setString(3, uss_command);                 // 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 DB2USSCommandException(rc, "DSNACCUC execution failed: " + message);
      }
      else
      {
```

```
        if (hasResultSet)
        {
          rs = cs.getResultSet();
          while (rs.next())
            System.out.println(rs.getString(2).trim());
          rs.close();
          cs.close();
        }
      }
    }

    catch (DB2USSCommandException uce)
    {
      System.out.println("Program error: rc=" + uce.getRC() + " message=" +
uce.getMessage());
    }
    catch (SQLException sqle)
    {
      System.out.println("SQL error: ec=" + sqle.getErrorCode() + " SQL state=" +
sqle.getSQLState() + " message=" + sqle.getMessage());
    }
    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) {}
    }
  }
}

class DB2USSCommandException extends Exception
{
  private int rc;

  DB2USSCommandException(int rc, String message)
  {
    super(message);
    this.rc = rc;
  }

  public int getRC()
  {
    return rc;
  }
}
```

Compile DB2USSCommand.java and enter the following command to execute it:

```
java DB2USSCommand DBALIAS USERID PASSWORD USS_USERID USS_PASSWORD USS_COMMAND
```

You should get the response shown in Example B-26.

Example: B-26 Response to command

```
total 32
drwxr-xr-x   2 PAOLOR8 SYS1        8192 Jan 19 00:14 .
-rwxr-xr-x   1 PAOLOR8 SYS1           0 Jan 19 00:14 dsnaccuc019001457
drwxr-xr-x  29 OMVSKERN OMVSGRP     8192 Jan 18 20:21 ..
```

C

DSNAIMS stored procedure

In this appendix we provide more details on the setup and utilization of the DSNAIMS stored procedure mentioned in DSNAIMS description

DSNAIMS is a stored procedure that allows DB2 applications to invoke IMS transactions and commands easily, without having to maintain their own connections to IMS. This stored procedure uses the IMS Open Transaction Manager Access (OTMA) API to connect with IMS and execute the transactions.

The DB2 documentation is being updated to include description of this new function. The setup and installation of DSNAIMS are in the updated *DB2 UDB for z/OS Version 8 Installation Guide,* GC18-7418-03, and the description and how to use DSNAIMS are in *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02.

C.1 DSNAIMS description

We describe the following topics:

- DSNAIMS prerequisites
- DSNAIMS setup
- DSNAIMS messages
- DSNAIMS examples
- DSNAIMS tips

C.1.1 DSNAIMS prerequisites

The following functions are equired before installing and executing the DSNAIMS stored procedure:

- A WLM-managed stored procedure address space in which to run DSNAIMS
- DB2 Version 7 or above with RRSAF enabled
- IMS Version 7 or above with OTMA Callable Interface enabled
- DB2 Version 7 requires PTFs UQ66553 and UQ94695 respectively for PQ44819 and PQ89544
- DB2 Version 8 requires PTF UQ94696 for PQ89544
- The PTF for APAR PK04339, currently open, is required if the text includes X'00's in the input varable 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

C.1.2 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 C/I 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: C-1 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)
```

Option descriptions

We describe the options available in DSNAIMS.

DSNAIMS_FUNCTION

A string indicating the desired operation:

- "SENDRECV"

  A send and receive of IMS data. This will invoke an IMS transaction or command and will return the result to the caller. The transaction can be an IMS full function or Fast Path. This function does not support multi-iteration of a conversational transaction.

- "SEND"

  A send-only of data to IMS. This will invoke an IMS transaction or command, but will return to the caller without receiving the IMS data. The result data, if any, can be retrieved by using the RECEIVE function. If an IMS transaction is submitted using this send-only function, the transaction cannot be an IMS fast path transaction or conversational transaction.

- "RECEIVE"

  A receive-only of data from IMS. The data can be the result of a transaction or command initiated by a previous SEND function, or by an unsolicited output message from an IMS application. This function will not initiate an IMS transaction or command.

DSNAIMS_2PC

Specifies whether or not to use an RRS/MVS two-phase commit process to perform the transaction syncpoint service. Possible values are "Y" or "N". If this parameter is set to “N”, any commits or rollbacks issued by the IMS transaction will not affect commit and rollback processing in the DB2 application invoking DSNAIMS, nor will IMS resources be affected by any commits or rollbacks issued in the calling DB2 application. Two-phase commit is only supported for the function 'SENDRECV' so when the value of DSNAIMS_2PC is "Y" then the value of DSNAIMS_FUNCTION should be "SENDRECV". The default is "N".

This parameter is an optional field.

XCF_GROUP_NAME

Specifies the XCF Group Name the IMS OTMA is to join. This name can be obtained by viewing the GRNAME parameter in IMS PROCLIB member DFSPBxxx. The output of IMS command /DISPLAY OTMA also shows the group name.

This parameter is a required field.

Note: Each instance of DSNAIMS will create only one connection to IMS. The first request to DSNAIMS will determine to which IMS subsystem the stored procedure will connect. It will only attempt to reconnect to IMS if IMS is restarted and the saved connection is no longer valid, or if WLM brings up another DSNAIMS task.

XCF_IMS_NAME

Specifies the XCF member name that IMS uses for the XCF group. This name can be obtained from the OTMANM parameter specified in the IMS PROCLIB member DFSPBxxx when IMS is not using the XRF or RSR feature. If IMS is using XRF or RSR, the XCF member

name that IMS uses comes from the USERVAR parameter specified in the IMS PROCLIB member DFSPBxxxx or DFSHSBxx.

This parameter is a required field.

Note: Each instance of DSNAIMS will create only one connection to IMS. The first request to DSNAIMS will determine to which IMS subsystem the stored procedure will connect. It will only attempt to reconnect to IMS if IMS is restarted and the saved connection is no longer valid, or if WLM brings up another DSNAIMS task.

RACF_USERID

Specifies the RACF User ID to be used for IMS to perform the transaction or command authorization checking. This can only be used when the stored procedure is APF-Authorized. This field is ignored when the stored procedure is not APF-Authorized.

This parameter is an optional field.

RACF_GROUPID

Specified the RACF Group ID to be used for IMS to perform the transaction or command authorization checking. This can only be used when the stored procedure is APF-Authorized. This field is ignored when the stored procedure is not APF-Authorized.

This parameter is an optional field.

IMS_LTERM

Specifies an IMS LTERM name used to override the LTERM name in the IMS application program's I/O PCB. This field is used as an input and output field. For SENDRECV, on input, the value is sent to IMS, and may be updated on output by IMS. For SEND, this parameter is IN only. For RECEIVE, it is OUT only. An empty or NULL value will tell IMS to ignore the parameter.

This parameter is an optional field.

IMS_MODNAME

Specified the formatting map name used by the server to map output data streams (for example, 3270 data streams). Although this invocation does not have the 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 (MOD) name that is placed in the I/O PCB. IMS places this name in the message prefix with the map name in the

IMS application program's I/O PCB when the message is inserted.

For SENDRECV, on input, the value is sent to IMS, and may be updated on output. For SEND, the parameter is IN only. For RECEIVE, it is OUT only. An empty or NULL value will tell IMS to ignore the parameter.

This parameter is an optional field.

IMS_TRAN_NAME

Specifies the name of an IMS transaction or command to be sent to IMS. If the IMS command desired is longer than eight characters, the first eight characters including the "/" of the command can be provided here. The remaining characters of the command need to be provided in the IMS_DATA_IN parameter. If an empty or NULL value is passed, the full transaction name or command must be provided in IMS_DATA_IN.

This parameter can be used for SENDRECV and SEND functions only. This parameter is ignored for RECEIVE.

This parameter is an optional field.

IMS_DATA_IN

Specifies the data to be sent to IMS.

This parameter is required if input data is needed for IMS, if no transaction name or command is passed in IMS_TRAN_NAME, or if the command is longer than 8 characters. This parameter is ignored for RECEIVE.

IMS_DATA_OUT

Data returned after successful completion of the transaction or command.

This parameter is required for SENDRECV and RECEIVE functions only. This parameter is ignored for SEND.

OTMA_TPIPE_NAME

Specifies an 8-byte user-defined communication session name for IMS to flow the input and output data for the transaction or command on SEND and RECEIVE. If an OTMA_TPIPE_NAME is used for a SEND function to generate an IMS output message, the same OTMA_TPIPE_NAME must be used for the subsequent RECEIVE function to retrieve the output. It is recommended not to have many different OTMA TPIPE names to save system storage.

This parameter is required for SEND and RECEIVE functions only. This parameter is ignored for SENDRECV.

OTMA_DRU_NAME

Specifies the name of an IMS user-defined exit, OTMA Destination Resolution User Exit Routine, if any. This IMS exit 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 will ignore this parameter.

USER_DATA_IN

This field does not contain the IMS transaction or command input. Rather, it is an optional field to contain any data to be included in the IMS message prefix so that the data will be seen by IMS OTMA user exits (DFSYIOE0 and DFSYDRU0 exits) and can be tracked by IMS log records. IMS application programs running in the dependent regions will not see this data. This specified user data will be included in the output message prefix for output. In some cases, users can use it to store the input/output correlator token or information. This parameter is ignored for RECEIVE.

This parameter is an optional field.

USER_DATA_OUT

This field does not contain the IMS transaction or command output data. On output, it contains the USER_DATA_IN in the IMS output prefix. IMS OTMA user exits (DFSYIOE0 and DFSYDRU0 exits) can also create the USER_DATA_OUT for SENDRECV or RECEIVE functions. This parameter is not updated for SEND.

This parameter is an optional field.

STATUS_MESSAGE

Any error message returned from the transaction or command, OTMA, RRS, or DSNAIMS.

This parameter is a required field.

RETURN_CODE

Return code returned from the transaction or command, OTMA, RRS, or DSNAIMS.

This parameter is a required field.

C.1.3 DSNAIMS messages

```
DSNA315I  DSNAIMS FUNCTION func-name HAS COMPLETED SUCCESSFULLY.
Explanation:  The stored procedure, DSNAIMS, executed successfully.
System Action:  None.
Programmer Response:  None.


DSNA316I DSNAIMS INPUT PARAMETER ERROR.  CALLER MUST PROVIDE INPUT FOR PARAMETER parm-name
WITH FUNCTION func-name.
Explanation:  A value for the specified parameter is required for the given function.
System Action:  DSNAIMS execution terminated before invoking IMS.
Programmer Response:  Provide an appropriate value for the parameter.


DSNA317I DSNAIMS ERROR IN otmaci-api API.  RC=return-code, RSN1=reason-code1,
RSN2=reason-code2, RSN3=reason-code3, RSN4=reason-code4.
Explanation:  The specified OTMA Callable Interface API encountered an error.
System Action:  DSNAIMS execution terminated after executing the specified API call.
Programmer Response:  Refer to Chapter 7 of the IMS OTMA Guide and Reference for
explanations of the return and reason codes.


DSNA318I DSNAIMS ERROR IN RRS CTXRCC API.  RC=return-code.
Explanation:  An error was encountered in RRS when processing the two-phase commit.
System Action:  DSNAIMS execution terminated before invoking IMS.
Programmer Response:  Refer to the book MVS Programming Resource Recovery for an
explanation of the return code.


DSNA319I DSNAIMS RECEIVED UNKNOWN FUNCTION func-name.
Explanation:  DSNAIMS received an unknown function name in parameter 1 of the stored
procedure call.
System Action:  DSNAIMS terminated before invoking IMS.
Programmer Response:  Specify a function name known to DSNAIMS in parameter 1.
```

C.1.4 DSNAIMS examples

Sample parameters for executing a simple IMS command are shown in Example C-2.

Example: C-2 IMS command

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMS7GRP", "IMS7TMEM",
                     "", "", "", "", "",
                     "/LOG Hello World.", ims_data_out, "", "", "",
                     user_out, error_message, rc)
```

Sample parameters for executing the IMS IVTNO transaction are shown in Example C-3.

Example: C-3 IMS transaction

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMS7GRP", "IMS7TMEM",
                     "", "", "", "", "",
                     "IVTNO     DISPLAY LAST1     ", ims_data_out,
                     "", "", "", user_out, error_message, rc)
```

Sample parameters for Send-only transaction invocation are shown in Example C-4.

Example: C-4 Send only transaction

```
CALL SYSPROC.DSNAIMS("SEND "N", "IMS7GRP", "IMS7TMEM",
                     "", "", "", "", "",
                     "IVTNO     DISPLAY LAST1     ", ims_data_out,
                     "DSNAPIPE", "", "", user_out, error_message,
                     rc)
```

Sample parameters for Receive-only invocation are shown in Example C-2.

Example: C-5 Receive only transaction

```
CALL SYSPROC.DSNAIMS("RECEIVE", "N", "IMS7GRP", "IMS7TMEM",
                     "", "", "", "", "",
                     "IVTNO     DISPLAY LAST1     ", ims_data_out,
                     "DSNAPIPE", "", "", user_out, error_message,
                     rc)
```

C.1.5 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)
- 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”, …)
```

D

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

D.1 Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

```
ftp://www.redbooks.ibm.com/redbooks/SG247083
```

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 the redbook form number, SG247083.

The zip files that accompany this redbook contain all of the sample files referenced within the book.

Important: Before downloading, we strongly suggest that you first read 3.3, “Sample application components” on page 22 to decide what components are applicable to your environment.

Note: The additional Web material that accompanies this redbook includes the files described in the following sections.

D.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\SG247083-DCLGEN.ZIP*

File name	*Description*
DEPT.TXT	Sample DCLGEN for DEPT table

EMP.TXT Sample DCLGEN for EMP table

D.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\SG247083-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=SG247083.CALDTL1C.CARDIN,DISP=SHR
```

simply indicates a sequential file to be allocated and used to pass a valid employee number to the stored procedure.

D.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\SG247083-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

D.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.

The following files can be found in *\Samples\SG247083-JAVA.ZIP*

CALDTLS.JAVA
EMPCLOBJ.DDL
EMPCOLBJ.JAVA
EMPCLOBJ.JCL
EMPCLOBSPSERVLET.JAVA
EMPDTL1J.DDL
EMPDTL1J.JCC.JCL
EMPDTL1J.JCL
EMPDTL1J.SQLJ
EMPDTLSJ.DDL
EMPDTLSJ.JAVA
EMPDTLSJ.JCL
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

D.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\SG247083-REXXSP.ZIP*

EMPDTLSR.DDL
EMPDTLSR.SRC
EMPRSETR.DDL
EMPRSETR.SRC

D.1.6 Sample SQL language stored procedures

This file contains DDL and program preparation JCL for all SQL language stored procedures used in the case studies. Since the source code for 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\SG247083-SQL.ZIP*

EMPDTLSS.DDL
EMPDTLSS.JCL
EMPDTLV8.DDL
EMPDTLV8.JCL
EMPRSETS.DDL
EMPRSETS.JCL
SQLSPCUS.JCL

D.1.7 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 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\SG247083-MULTITHD.ZIP*

DB2CallRUNSTATP.java
RUNSTATP.DDL
RUNSTATP.JCL
RUNSTATP.SRC

D.1.8 Sample code to invoke DB2-supplied stored procedures

This file contains source code for Java programs that were used for invoking DB2-supplied stored procedure. The stored procedures are introduced in Chapter 25, “DB2-supplied stored procedures” on page 403 and described in Appendix B, “Samples for using DB2-supplied stored procedures” on page 595.

The following files can be found in *\Samples\SG247083-DB2SUPPLIED.ZIP*

DB2Command.java
DB2DatasetUtilities.java
DB2JCLUtilities.java
DB2Runstats.java
DB2SystemInformation.java
DB2USSCommand.java
DB2USSUserInformation.java
DB2WLMRefresh.java

D.1.9 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\SG247083-QMF.ZIP*

FRPARM70.FRM
FRPARMER.FRM
FRTNONLY.FRM
QRPARM70.TXT
QRPARMER.TXT
QRTNONLY.TXT

D.1.10 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\SG247083-TRIGGER.ZIP*

EMPTRIG1.DDL
EMPTRIG2.DDL
EMPTRIG3.DDL

D.1.11 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\SG247083-REXXEXEC.ZIP*

DDLMOD.SRC
DDLMODJB.JCL
GETSQLJB.JCL
GETSQLSP.SRC
PUTSQLJB.JCL
PUTSQLSP.SRC

D.1.12 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\SG247083-JCLIMS.ZIP*

IMS01.TXT
IMS02.TXT
IMS03.TXT
IMS04.TXT
IMS05.TXT
IMS06.TXT
IMS07.TXT
IMS08.TXT
IMS09.TXT
IMS10A.TXT
IMS10B.TXT
IMS11.TXT
IMS12.TXT

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.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 660. Note that some of the documents referenced here may be available in softcopy only.

- *Exploring WebSphere Studio Enterprise Developer 5.1.2,* SG24-6483
- *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
- *DB2 Java Stored Procedures Learning by Example,* SG24-5945
- *Cross-Platform DB2 Stored Procedures: Building and Debugging,* SG24-5485-01
- *Getting Started with DB2 Stored Procedures: Give Them a Call through the Network,* SG24-4693-01
- *Large Objects with DB2 for z/OS and OS/390*, SG24-6571
- *DB2 Performance Expert for z/OS*, SG24-6867
- *Building the Operational Data Store on DB2 UDB Using IBM Data Replicator, WebSphere MQ Family, and DB2 Warehouse Manager,* SG24-6513

Other publications

These publications are also relevant as further information sources:

- *z/OS C/C++ Programming Guide*, SC09-4765-03
- *z/OS C/C++ Run-Time Library Reference.*, SA22-7821
- *z/OS V1R4.0 Security Server RACF Command Language Reference,* SA22-7687
- *z/OS V1R4.0 Language Environment Customization*, SA22-7564-04
- *z/OS V1R4.0 Language Environment Programming Guide,* SA22-7561-04
- *z/OS V1R4.0 MVS Setting Up a Sysplex*, SA22-7625-06
- *z/OS V1R4.0 MVS Programming: Resource Recovery*, SA22-7616-02
- *z/OS MVS Planning: Operations,* SA22-7601-03
- *z/OS MVS Initialization and Tuning Guide,* SA22-7591-01
- *z/OS MVS Support for Unicode: Using Conversion Services*, SC33-7050
- *z/OS Security Server RACF Command Language Reference*, SA22-7687-03
- *z/OS UNIX System Services Planning*, GA22-7800-03
- *Building the Operational Data Store on DB2 UDB Using IBM Data Replicator, WebSphere MQ Family, and DB2 Warehouse Manager,* SG24-6513

- *Enterprise COBOL for z/OS and OS/390 Programming Guide Version 3 Release 2*, SC27-1412-01
- *Enterprise COBOL for z/OS and OS/390 Language Reference Version 3 Release 2*, SC27-1408-01
- *Application Programming Guide and Reference for Java*, SC18-7414
- *CICS Transaction Server for z/OS Version 2.2 CICS DB2 Guide*, SC34-6014-07
- *CICS Transaction Server for z/OS Version 2.2 Application Programming Guide*, SC34-5933
- *CICS Transaction Server for z/OS Version 2.3 CICS Application Programming Guide,* SC34-6231
- *CICS External Interfaces Guide*, SC34-6006-05
- *Query Monitor Facility Reference Version 7,* SC27-0715
- *DB2 Performance Monitor for z/OS Version 7.2, Report Reference*, SC27-1647-02
- *DB2 Performance Monitor for z/OS Version 7.2, Reporting User's Guide*, SC27-1651-02
- *DB2 Performance Expert for z/OS Version 1 Monitoring Performance from ISPF*, SC27-1652-02
- *DB2 UDB V8 Application Development Guide: Programming Client Applications,* SC09-4826
- *z/OS Language Environment Programming Reference,* SA22-7562
- *Debug Tool for z/OS User's Guide,* SC18-9012
- *Debug Tool for z/OS Customization Guide*, SC18-9008
- *Debug Tool for z/OS Reference and Messages*, SC18-9010
- *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201
- *New IBM Technology Featuring Persistent Reusable Java Virtual Machines*, SC34-6034
- *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413
- *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415
- *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java,* SC18-7414
- *DB2 UDB for z/OS Version 8 Command Reference,* SC18-7416
- *DB2 UDB for z/OS Version 8 Data Sharing: Planning and Administration,* SC18-7417
- *DB2 UDB for z/OS Version 8 Diagnosis Guide and Reference,* LY37-3201
- *DB2 UDB for z/OS Version 8 Diagnostic Quick Reference Card,* LY37-3202
- *DB2 UDB for z/OS Version 8 Image, Audio, and Video Extenders Administration and Programming,* SC18-7429
- *DB2 UDB for z/OS Version 8 Installation Guide,* GC18-7418
- *DB2 UDB for z/OS Version 8 Licensed Program Specifications,* GC18-7420
- *DB2 UDB for z/OS Version 8 Messages and Codes,* GC18-7422
- *DB2 UDB for z/OS Version 8 ODBC Guide and Reference,* SC18-7423
- *An Introduction to DB2 Universal Database for z/OS Version 8,* SC18-7419
- *DB2 UDB for z/OS Version 8 Program Directory,* GI10-8566
- *DB2 UDB for z/OS Version 8 RACF Access Control Module Guide,* SC18-7433

- *DB2 UDB for z/OS Version 8 Reference for Remote DRDA Requesters and Servers,* SC18-7424
- *DB2 UDB for z/OS Version 8 Reference Summary,* SX26-3853
- *DB2 UDB for z/OS Version 8 Release Planning Guide,* SC18-7425
- *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426
- *DB2 UDB for z/OS Version 8 Text Extender Administration and Programming,* SC18-7430
- *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427
- *DB2 UDB for z/OS Version 8 What's New?,* GC18-7428
- *DB2 UDB for z/OS Version 8 XML Extender for z/OS Administration and Programming,* SC18-7431

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/`
- Our servlet EmpPhotoSpServlet at:

  `http://localhost:9080/DBASPSERV/servlet/EmpPhotoSpServlet`
- SQL Debugger for DB2 UDB V7.2 and DB2 UDB V8.1:

  `http://www.ibm.com/developerworks/db2/library/techarticle/alazzawe/0108alazzawe.html`
- IBM Distributed Debugger available for download at:

  `http://www7b.boulder.ibm.com/dmdd/library/techarticle/0305rader/0305rader.html`
- DB2 V8.1 UDB Application Development Client available at:

  `http://www.ibm.com/software/data/db2/os390/spb/client.html`
- Unicode Web site at:

  `http://www.ibm.com/servers/s390/os390/bkserv/latest/v2r10unicode.html`
- DB2 DEMO workstation GUI tool at:

  `http://www.ibm.com/developerworks/db2/library/demos/db2demo/index.html`
- DB2 Stored Procedure Builder Web page.

  `http://www-306.ibm.com/software/data/db2/os390/spb/`
- DB2 for z/OS and OS/390 library Web page:

  `http://www-306.ibm.com/software/data/db2/os390/library.html`
- JDBC Driver selection paper by Peggy Rader:

  `http://www-106.ibm.com/developerworks/db2/library/techarticle/dm-0408rader/index.html`

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

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)
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
WAS	WebSphere Application Server
WSAD	WebSphere Studio Application Developer
WSADIE	WebSphere Studio Application Developer Integration Edition
WSED	WebSphere Studio Enterprise Developer
WLM	work load manager

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Redbooks

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

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

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

Develop and test COBOL, C, REXX, Java, and SQL language stored procedures

Set up, control, and tune the operating environment

Learn tools and DB2 supplied stored procedures

This IBM Redbook helps you design, install, manage, and tune stored procedures with DB2 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. DB2's enhancements are related to tooling, language support, system environment, and have opened new possibilities for secure, highly portable applications in line with the e-business strategy of today's organizations.

In this project we show how to develop stored procedures in several languages; we explore the new functions available for the z/OS platform deployment; and provide recommendations on setting up and tuning the appropriate stored procedure environment. The functions we have investigated include setting up the WLM environments, nesting stored procedure, invoking COBOL, C, REXX, SQL language programs, accounting, debugging options, special registers, and diagnostics. We have also set up, developed, and debugged Java stored procedures with the new Java Universal Driver in a DB2 for z/OS Version 8 environment. A chapter is devoted to DB2-supplied stored procedures. They can be used for almost all of a DBA's tasks.

We start with the basic information, which is useful for the reader who is just beginning with stored procedures, but we also deal with more detailed and recent functionalities, which will be of interest for the more advanced users.

SG24-7083-00 ISBN 0738498181

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