Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries

Develop robust DB2 Universal Database for iSeries applications

Discover the details of SQL stored procedures and SQL triggers

Learn the secrets of user-defined functions

Hernando Bedoya
Fredy Cruz
Daniel Lema
Satid Singkorapoom
Take Note! Before using this information and the product it supports, be sure to read the general information in “Notices” on page xi.
Contents

Notices ................................................................. xi
Trademarks ......................................................... xii

Preface ............................................................... xiii
The team that wrote this redbook ................................. xiv
Become a published author ....................................... xvi
Comments welcome .............................................. xvi

Part 1.  Background ................................................. 1

Chapter 1.  Introducing DB2 Universal Database for iSeries .... 3
1.1 An integrated relational database ............................... 4
1.2 DB2 Universal Database for iSeries: An overview ............. 4
1.2.1 DB2 Universal Database for iSeries basics .................. 5
1.2.2 Stored procedures, triggers, and user-defined functions .... 6
1.3 DB2 Universal Database for iSeries sample schema .......... 7

Chapter 2.  Stored procedures, triggers, and user-defined functions: Order entry application .... 11
2.1 Order Entry application overview ............................... 12
2.2 Order Entry database overview ................................. 13
2.3 Stored procedures and triggers in the Order Entry database ... 17
  2.3.1 Stored procedures ........................................ 18
  2.3.2 Triggers .................................................. 18
  2.3.3 User-defined functions ................................... 18

Chapter 3.  Introduction to the SQL Persistent Stored Module in DB2 Universal Database for iSeries .... 19
3.1 Introduction ..................................................... 20
3.2 System requirements and planning ............................. 20
3.3 Structure of a SQL PSM program ............................... 21
3.4 SQL control statements ........................................ 21
  3.4.1 Assignment statement ...................................... 21
  3.4.2 Conditional control ....................................... 22
  3.4.3 Iterative control ......................................... 24
  3.4.4 Calling procedures ....................................... 29
3.5 Compound SQL statement ...................................... 29
  3.5.1 Nested compound statements .............................. 30
  3.5.2 Variable declaration ...................................... 30
  3.5.3 Using cursors ........................................... 31
3.6 Using scrollable cursors in SQL PSM .......................... 33
3.7 Dynamic SQL in SQL PSM ...................................... 34
  3.7.1 PREPARE statement ....................................... 34
  3.7.2 EXECUTE statement ....................................... 35
  3.7.3 EXECUTE IMMEDIATE statement ......................... 35
  3.7.4 Cursors based on dynamic SQL ............................ 35
3.8 Moving into production (save and restore) .................... 35
  3.8.1 Restore processing for V4R5 and prior releases ............ 35
  3.8.2 Restore processing for V5 ................................. 36
Chapter 10. SQL triggers

10.1 Introduction to SQL triggers .................................................... 296
10.2 System requirements and planning ........................................ 296
10.3 Structure of an SQL trigger ..................................................... 297
   10.3.1 Components of the SQL trigger definition ....................... 298
   10.3.2 Simple SQL trigger example ....................................... 300
   10.3.3 Example of a trigger program using WHEN condition .......... 300
10.4 Creating an SQL trigger .......................................................... 301
   10.4.1 Creating an SQL trigger with iSeries Navigator .................. 301
   10.4.2 Creating an SQL trigger with the Run SQL Scripts utility ....... 308
   10.4.3 Creating SQL triggers with traditional interfaces .............. 312
   10.4.4 Verifying the SQL trigger properties ............................ 316
10.5 Deleting or replacing an SQL trigger ..................................... 318
10.6 Trigger component details .................................................... 319
   10.6.1 Trigger time ................................................................ 322
   10.6.2 Trigger modes ............................................................. 323
   10.6.3 Trigger granularity ....................................................... 326
10.7 Accessing triggering data ....................................................... 328
   10.7.1 Correlation variables .................................................... 329
   10.7.2 Transition tables .......................................................... 330
   10.7.3 Changing values in the firing row .................................. 332
10.8 Error handling .................................................................... 332
   10.8.1 Signaling errors from a trigger ..................................... 334
   10.8.2 Recovering errors fired by triggers ............................... 335
10.9 Inoperative triggers ............................................................... 337
10.10 Moving into production (save and restore) .............................. 338
10.11 Resolution of unqualified object references ............................ 338
10.12 Transaction isolation and recovery ......................................... 340
   10.12.1 Savepoints ............................................................... 341
   10.12.2 ATOMIC ............................................................... 341
10.13 Additional considerations ..................................................... 342
   10.13.1 Adding columns to a subject table referenced in the triggered action ...................... 342
   10.13.2 Datetime considerations ............................................. 342
   10.13.3 SQL trigger program object ....................................... 342
   10.13.4 Authority of SQL triggers ......................................... 342
10.14 Testing and debugging ......................................................... 343
   10.14.1 The ILE source debugger ............................................ 343
   10.14.2 Preparing the SQL trigger for debugging ....................... 344
   10.14.3 Testing the SQL trigger ............................................. 346
   10.14.4 Testing the SQL trigger in a client/server environment ....... 347
This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

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

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

- AS/400®
- C/400®
- COBOL/400®
- DB2 Connect™
- DB2 Universal Database™
- DB2®
- DRDA®
- eServer™
- i5/OS®

- IBM®
- ibm.com®
- iSeries™
- Integrated Language Environment®
- Language Environment®
- Lotus®
- NetServer™
- Operating System/400®
- OS/2®

- OS/390®
- OS/400®
- Redbooks™
- Redbooks (logo)™
- System i™
- SQL/400®
- VisualAge®
- WebSphere®

The following terms are trademarks of other companies:

Java, JDBC, JDK, JRE, JVM, J2EE, J2SE, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Expression, Microsoft, MS-DOS, Visual Basic, Windows NT, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

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.
Preface

Stored procedures, triggers, and user-defined functions (UDFs) are the key database features for developing robust and distributed applications. DB2® Universal Database™ for iSeries™ has supported these features for many years, and they have been enhanced in V5R1, V5R2, and V5R3 of IBM® OS/400® and V5R4 of IBM i5/OS®.

This IBM Redbook includes some of the announced features for stored procedures, triggers, and UDFs in V5R1, V5R2, V5R3, and V5R4. Among the topics in this IBM Redbook, you will find suggestions, guidelines, and practical examples on how to effectively develop DB2 Universal Database for iSeries stored procedures, triggers, and UDFs. Some of the topics that are covered in this book include:

- Introduction to the SQL Persistent Stored Module Language used in SQL stored procedures, triggers, and UDFs
- SQL stored procedures
- External stored procedures and triggers
- Java™ stored procedures (both Java Database Connectivity (JDBC™) and Structured Query Language for Java (SQLJ))
- SQL triggers
- External triggers
- SQL UDFs
- External UDFs

This IBM Redbook also offers examples that were developed in several programming languages, including RPG, COBOL, C, Java, and Visual Basic®, using native and SQL data access interfaces.

Some of the material related to stored procedures and triggers was originally published in the IBM Redbook DB2 UDB for AS/400 Advanced Database Functions, SG24-4249-02. However, due to the importance of this information, we decided to move the topics of stored procedures and triggers into their own publication. The remaining topics from the original redbook are covered in the updated IBM Redbook Advanced Functions and Administration on DB2 Universal Database for iSeries, SG24-4249-03. The material related to UDFs was originally published in the IBM Redbook DB2 UDB for AS/400 Object Relational Support, SG24-5409, and it has been enhanced in this IBM Redbook.

Prior to reading this IBM Redbook, you should have some knowledge of relational database technology and the application development environment on the IBM eServer™ iSeries server.

Note: With the release of IBM i5/OS V5R4, the name of DB2 Universal Database for iSeries has changed to DB2 for i5/OS. Considering the previous editions of this book and the minimal changes in this update, we have chosen to continue using the name DB2 Universal Database for iSeries in this edition.

© Copyright IBM Corp. 2001, 2004, 2006
The team that wrote this redbook

This IBM Redbook was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Rochester Center.

Hernando Bedoya is an IT Specialist at the IBM ITSO, in Rochester, Minnesota. He writes extensively and teaches IBM classes worldwide in all areas of DB2 Universal Database for iSeries. Before joining the ITSO more than six years ago, he worked for IBM Colombia as an IBM AS/400® IT Specialist doing presales support for the Andean countries. He has 24 years of experience in the computing field and has taught database classes in Colombian universities. He holds a Master in Computer Science degree from EAFIT, Colombia. His areas of expertise are database technology, application development, and data warehousing.

Fredy Cruz is the ISV Coordinator at IBM Colombia. He helps developers with infrastructure migration over software including WebSphere® and DB2 on both Linux® on System i™ and i5/OS. He also teaches ISVs and customers about the utilities and technologies related to this task. His responsibilities are to show customers how to implement the new utilities in the IBM System i environment as well as to sales or technical people that work with the System i platform. His areas of expertise include working with the Linux and Microsoft® Windows® environment on the System i platform, DB2 and WebSphere in OS/400, i5/OS, and Linux, and Lotus® over OS/400 and i5/OS environment.

Daniel Lema is an IT Architect at IBM Andean, with 15 years of experience. Some of his projects include working with Business Intelligence, database modeling, and extract, transform, and load (ETL) modeling and implementation, with experience in the Banking Data Warehouse Model and the banking industry. Previously, he worked as a sales specialist for the Midrange Server Product Unit (formerly the AS/400 Product Unit) helping customers and sales people in designing AS/400- and DB2/400-based solutions. He has been a lecturer in Information Management and Information Technology Planning in the Graduate School at EAFIT University and other Colombian universities. He is also an Information Systems Engineer and is working on earning an Applied Mathematics master degree at EAFIT University, where he has already finished his academic activities.

Satid Singkorapoom is an Advisory Product Specialist for System i Sales unit of IBM Thailand. He has 16-years of experience with System i products. He holds a Master of Computer Engineering degree from the Asian Institute of Technology in Thailand. His areas of expertise include DB2 Universal Database for System i technology, SQL and System i performance analysis and tuning, System i logical partitioning, SAP on System i Technical Infrastructure, and System i hardware architecture. He has coauthored five IBM Redbooks™ and three DB2 technical training materials from the ITSO Rochester over the past 13 years. He also teaches System i customers regularly on various product technology updates, SQL performance analysis and tuning, OS/400 for SAP deployment, and System i administration.
This IBM Redbook is based on projects that were conducted in 1994, 1997, 2000, 2001, and 2006 by the ITSO Rochester Center.

The advisors of the projects were:

Michele Chilanti
Jarek Miszczyk
ITSO Rochester Center

The authors involved in the previous editions of this IBM Redbook were:

Christophe Delponte
IBM Belgium

Cintia Marques
IBM Brazil

Thelma Bruzadin
ITEC Brazil

Hernando Bedoya
IBM Colombia

Roger H.Y. Leung
IBM Hong Kong

Oh Sun Kang
IBM Korea

Suparna Murthy
Deepak Pai
IBM India

Clarice Rosa
IBM Italy

Teresa Kan
Kent Milligan
IBM Rochester

Alex Metzler
IBM Switzerland

Claus Weiss
IBM Toronto Lab

Vijay Marwaha
IBM US
Thanks to the following people for their invaluable contributions to this project:

Mark Anderson
John Eberhard
Mietek Konczyk
Jarek Miszczyk
Kent Milligan
Kathy Passe
Jon Triebenbach
IBM Rochester

Become a published author

Join us for a two- to six-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 have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

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

To learn 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 e-mail to:
  redbooks@us.ibm.com
► Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400
Part 1

Background

This part introduces the basics concepts of DB2 Universal Database for iSeries. It describes the Order Entry application used to illustrate the use of stored procedures and triggers. Plus, it explains the sample database provided in DB2 Universal Database for iSeries in V5R1 that will be used as well for illustrating stored procedures, triggers and user-defined functions. We also dedicate one chapter to introduce the SQL Persistent Stored Module Language used in SQL stored procedures, triggers and functions.

Note: With the release of IBM i5/OS V5R4, the name of DB2 Universal Database for iSeries has changed to DB2 for i5/OS. Considering the previous editions of this book and the minimal changes in this update, we have chosen to continue using the name DB2 Universal Database for iSeries in this edition.
Introducing DB2 Universal Database for iSeries

This chapter includes:
- An introduction to DB2 Universal Database for iSeries
- An overview of the contents in this IBM Redbook
- Definition of the sample schema

Note: With the release of IBM i5/OS V5R4, the name of DB2 Universal Database for iSeries has changed to DB2 for i5/OS. Considering the previous editions of this book and the minimal changes in this update, we have chosen to continue using the name DB2 Universal Database for iSeries in this edition.
1.1 An integrated relational database

Integration has been one of the major elements of differentiation of the iSeries server in the information technology marketplace. The advantages and drawbacks of fully integrated systems have been the subject of endless disputes in the last few years. The success of the AS/400 system and iSeries server indicates that integration is still considered one of the premier advantages of this platform. Security, communications, data management, backup, and recovery: All of these vital components have been designed in an integrated way on the AS/400 system and iSeries server. They work according to a common logic with a common end-user interface. They fit together perfectly, since all of them are part of the same software, the Operating System/400® (OS/400).

The integrated relational database manager has always been one of the most significant facilities that the iSeries server provides to users. Relying on a database manager integrated into the operating system means that virtually all the user data on the iSeries server is stored in a relational database and that the access to the database is implemented by the operating system itself. Some database functions are implemented at a low level in the iSeries server architecture, while some are even performed by the hardware.

Several years ago a survey pointed out that a significant percentage of iSeries server customers did not even know that all of their business data is stored in a relational database. This might sound strange if you think that we consider the integrated database as one of the main technological advantages of the iSeries platform. This means that thousands of customers use, manage, back up, and restore a relational database every day without even knowing that they have it installed on their system. This level of transparency has been made possible by the integration and by the undisputed ease of use of this platform. These have been key elements of the success of the AS/400 and iSeries server database system in the marketplace.

During the last couple of years, each new release of OS/400 has enhanced the DB2 Universal Database for iSeries with a dramatic set of new functions. As a result of these enhancements, the iSeries server has become one of the most functionally rich relational platforms in the industry.

DB2 Universal Database for iSeries is a member of the DB2 Universal Database family of products, which includes DB2 Universal Database for OS/390 and DB2 Universal Database. The DB2 Universal Database family is the IBM proposal in the marketplace of relational database systems and guarantees a high degree of application portability and a sophisticated level of interoperability among the various platforms that are participating in the family.

1.2 DB2 Universal Database for iSeries: An overview

This section provides a quick overview of the major features of DB2 Universal Database for iSeries. You can find a full description of the functions that are mentioned in this section in several IBM manuals, for example:

- *Database Programming*, SC41-5701
- *SQL Reference*, SC41-5612
1.2.1 DB2 Universal Database for iSeries basics

As previously mentioned, the major distinguishing characteristic of the DB2 Universal Database for iSeries database manager is that it is part of the operating system. In practice, this means that a large majority of your iSeries server data is stored in the relational database. Although the iSeries server also implements other file systems in its design, the relational database on the iSeries server is the most commonly used by the customers. Your relational data is stored in the database, plus typical non-relational information, such as the source of your application programs.

Physical files and tables

Data on the iSeries server is stored in objects called physical files. Physical files consist of a set of records with a predefined layout. Defining the record layout means that you define the data structure of the physical file in terms of the length and the type of data fields that participate in that particular layout.

These definitions can be made through the native data definition language of DB2 Universal Database for iSeries, called data description specifications (DDS). If you are familiar with other relational database platforms, you are aware that the most common way to define the structure of a relational database is by using the data definition statements provided by the Structured Query Language (SQL). This is also possible on the iSeries server. The SQL terminology can be mapped to the native DB2 Universal Database for iSeries terminology for relational objects. An SQL table is equivalent to a DDS-defined physical file. We use both terms interchangeably in this book. Similarly, table rows equate to physical file records for DB2 Universal Database for iSeries, and SQL columns are synonymous with record fields.

Logical files, SQL views, and SQL indexes

By using DDS, you can define logical files on your physical files or tables. Logical files provide a different view of the physical data, allowing columns subsetting, record selection, joining multiple database files, and so on. They can also provide physical files with an access path when you define a keyed logical file. Access paths can be used by application programs to access records directly by key or for ensuring uniqueness.

On the SQL side, there are similar concepts. An SQL view is almost equivalent to a native logical file. The selection criteria that you can apply in an SQL view is much more sophisticated than in a native logical file. An SQL index provides a keyed access path for the physical data exactly the same way as a keyed logical file does. Still, SQL views and indexes are treated differently from native logical files by DB2 Universal Database for iSeries, and they cannot be considered to exactly coincide.

Database file refers to any DB2 Universal Database for iSeries file, such as a logical or physical file, an SQL table, or view. Any database files can be used by applications to access DB2 Universal Database for iSeries data.

Terminology

Since the DB2 Universal Database for iSeries server evolved from the built-in database present in the AS/400 that was born before SQL was widely-used, OS/400 uses different terminology than what SQL uses to refer to database objects. The terms and their SQL equivalents are found in Table 1-1. The terms have been interchanged throughout this book.
1.2.2 Stored procedures, triggers, and user-defined functions

The main purpose of this IBM Redbook is to describe, in detail and with practical examples, the support of stored procedures, triggers and user-defined functions (UDFs) in DB2 Universal Database for iSeries.

**Stored procedures**

A stored procedure is an ordinary program that can be called by an application with an SQL CALL statement. The stored procedure can be called locally or remotely. A remote stored procedure provides the most advantages:

- It reduces traffic across the communication line.
- It splits the application logic and encourages an even distribution of the computational workload.
- It provides an easy way to call a remote program.

DB2 Universal Database for iSeries supports two types of stored procedures:

- SQL stored procedures
- External procedures

**Database triggers**

Triggers are user-written programs that are associated with database tables. You can define a trigger for update, delete, and insert operations. Whenever the operation takes place, regardless of the interface that is changing the data, the trigger program is automatically activated by DB2 Universal Database for iSeries and executes its logic. In this way, you can implement complex rules at the database level with total independence from the application environment. You can use triggers for a variety of purposes in your database design.

There are two examples are data validation and audit trail creation. DB2 Universal Database for iSeries supports two types of triggers:

- SQL triggers
- External triggers

---

**Table 1-1 SQL terms and OS/400 terms cross-reference**

<table>
<thead>
<tr>
<th>SQL term</th>
<th>iSeries term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Physical file</td>
</tr>
<tr>
<td>View</td>
<td>Non-keyed logical file</td>
</tr>
<tr>
<td>Index</td>
<td>Keyed logical file</td>
</tr>
<tr>
<td>Column</td>
<td>Field</td>
</tr>
<tr>
<td>Row</td>
<td>Record</td>
</tr>
<tr>
<td>Schema</td>
<td>Library, Collection, Schema</td>
</tr>
<tr>
<td>Log</td>
<td>Journal</td>
</tr>
<tr>
<td>Isolation level</td>
<td>Commitment control level</td>
</tr>
</tbody>
</table>
User-defined functions and user-defined table functions

UDFs and user-defined table functions (UDTFs) are user-written programs that enrich the functionality of the database manager by adding new functions to the set of built-in functions.

UDFs are scalar functions, which means functions that receive some parameters, perform some operations, and return a unique value, such as converting farenheit to celsius degrees or calculating the net present value given the final amount, monthly payment, number of payments and interest rate.

UDTFs are functions that return a table for a given set of parameters instead of a single scalar value, such as the top \( k \) performing salesperson or the projected currency exchange rates between an initial and final date for a given pair of currencies.

DB2 Universal Database for iSeries supports three types of UDFs:
- SQL UDFs
- External UDFs
- Sourced UDFs

1.3 DB2 Universal Database for iSeries sample schema

Within the code of OS/400 V5R1M0, there is a stored procedure that creates a fully functioning database. This database contains tables, indexes, views, aliases, and constraints. It also contains data within these objects.

The database also helps with problem determination since the program is shipped with the OS/400 V5R1M0 code. By calling a simple program, you can create a duplicate of this database on any system running V5R1M0. This enables customers and support staff to work on the same database for problem determination.

Working on the same database provides the ability for customers around the world to see the new functionality at V5R1M0. It also simplifies the setup environment for the workshops that are created in the future for use by the customer.

You create the database by issuing the following SQL statement:

```
CALL QSYS.CREATE_SQL_SAMPLE('SAMPLEDBXX')
```
You can find this statement in the example pull-down box of the Run SQL Script window (Figure 1-1).

![Example display showing the schema CREATE statement](image)

**Figure 1-1  Example display showing the schema CREATE statement**

**Note:** The schema name must be in uppercase. This sample schema will also be used in future DB2 Universal Database for iSeries documentation.
As a group, the tables include information that describes employees, departments, projects, and activities. This information makes up a sample database demonstrating some of the features of DB2 Universal Database for iSeries. An entity-relationship (ER) diagram of the database is shown in Figure 1-2.

The tables are:

- Department Table (DEPARTMENT)
- Employee Table (EMPLOYEE)
- Employee Photo Table (EMP_PHOTO)
- Employee Resume Table (EMP_RESUME)
- Employee to Project Activity Table (EMPPROJACT)
- Project Table (PROJECT)
- Project Activity Table (PROJACT)
- Activity Table (ACT)
- Class Schedule Table (CL_SCHED)
- In Tray Table (IN_TRAY)

Indexes, aliases, and views are created for many of these tables. The view definitions are not included here. There are three other tables created that are not related to the first set:

- Organization Table (ORG)
- Staff Table (STAFF)
- Sales Table (SALES)

Note: Some of the examples in this book use the sample database that was just described.
Chapter 2. Stored procedures, triggers, and user-defined functions: Order entry application

This chapter describes how a simple Order Entry application can take advantage of the stored procedures, triggers and user-defined functions (UDFs) support available with DB2 Universal Database for iSeries. It describes the complete application, in terms of logical flow and database structure. You can find the actual implementation of this application in the specific chapters that exploit this application scenario to show how to use the DB2 Universal Database for iSeries stored procedures, triggers and UDFs.

By presenting an application scenario, we intend to show how the stored procedures, triggers and UDFs in DB2 Universal Database for iSeries can be applied to a real-life environment, and the technical implications of using those functions. For this reason, the application may seem simplistic in some respects (for example, the user interface or some design choices). We present a simple, easy-to-understand scenario that includes most of the aspects that are discussed throughout this IBM Redbook.

We chose to develop the various components of the application using different programming languages to show how the various languages can interact with DB2 Universal Database for iSeries.
2.1 Order Entry application overview

The Order Entry application shown in Figure 2-1 represents a simple solution for an office stationery wholesaler.

This application has the following characteristics:

- The wholesale company runs a main office and several branch offices.
- A requirement of the branch offices is their autonomy and independence from the main office.
- Data is, therefore, stored in a distributed relational database. Information about customers and orders are stored at the branch office, where the central system keeps information about the stock and suppliers.
- A main requirement of this company is the logical consistency of the database. All orders, for example, must be related to a customer, and all the products in the inventory must be related to a supplier.
- The sales representative contacts the customer over the telephone. Each sales representative is assigned a pool of customers. According to the policy of the sales division of this company, a sales representative is allowed to place orders only for a
customer of his pool. This policy is needed to guarantee a fair distribution of the commissions on the sales representative’s turnover. This requirement can be effectively enforced by means of a trigger program that automatically checks the relationship between a customer and the sales representative when the order is placed (see 11.4.2, “Audit trail trigger example programs” on page 381).

- In placing an order, the sales representative first introduces some general data, such as the order date, the customer code, and so on. This process generates a row in the Order Header table.

- The sales representative then inserts one or more items for that specific order. If the specific item is out of stock, we want the application to look in the inventory for an alternative article. The inventory is organized in categories of products and, on this basis, the application performs a search. Since the inventory table is located remotely, we use a DRDA® connection between the systems. In addition, since the process of searching the inventory may involve many accesses to the remote database, a stored procedure is called to carry out this task.

- When the item or a replacement has been found, the inventory is updated, and a row is inserted in the local order detail table.

- At this point, we want to release the inventory row to allow other people to place a new order for the same product. We commit the transaction at this time. DB2 Universal Database for iSeries ensures the consistency of the local and remote databases, thanks to the two-phase commitment control support.

- When all order items have been entered, the order is finished and a finalizing order program is called. This program can:
  - Add the total amount of the order to the Customer table to reflect the customers’ turnover.
  - Update the total revenue produced by the sales representative from this customer.
  - Update the total amount of the order in the Order Header table.

- An update event of the Order Header table starts another trigger program that writes the invoice immediately at the branch office.

- As we mentioned, the “atomic” logical transaction is completed when a single item in the order has been inserted to reduce the locking exposures. If the system or the job fails, we must be able to detect incomplete orders. This can be done when the user restarts the application. A simple restart procedure will check for orders having the total equal to zero (not “finalized”). These orders are deleted and the stock quantity of all the items is increased by the amount that we reserved during the order placement. We can also present a choice menu to the user, asking whether the incomplete orders should be finalized.

### 2.2 Order Entry database overview

The Order Entry application is based on a distributed database. Each branch office location keeps all the data related to its own customers in its local database. The information concerning the items available in the warehouse is stored in the remote database at the head office.

The local database consists of these tables:

- **CUSTOMER table**: Contains the information related to the customers
- **ORDERHDR table**: With the data related to where the Order items are stored
ORDERDTL table: Where each row represents a Detail of an Order
SALESCUS table: Keeps the relationship between a sales representative and the customers for whom that sales representative is authorized to place orders

The central database consists of two tables:
STOCK table: Contains information about the contents of the warehouse
SUPPLIER table: Contains information related to the suppliers

Table 2-1 CUSTOMER table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER_NUMBER</td>
<td>CUSBR</td>
<td>CHAR(5)</td>
<td>Customer number</td>
</tr>
<tr>
<td>CUSTOMER_NAME</td>
<td>CUSNAM</td>
<td>CHAR(20)</td>
<td>Customer name</td>
</tr>
<tr>
<td>CUSTOMER_TELEPHONE</td>
<td>CUSTEL</td>
<td>CHAR(15)</td>
<td>Customer phone number</td>
</tr>
<tr>
<td>CUSTOMER_FAX</td>
<td>CUSFAX</td>
<td>CHAR(15)</td>
<td>Customer fax number</td>
</tr>
<tr>
<td>CUSTOMER_ADDRESS</td>
<td>CUSADR</td>
<td>CHAR(20)</td>
<td>Customer address</td>
</tr>
<tr>
<td>CUSTOMER_CITY</td>
<td>CUSCTY</td>
<td>CHAR(20)</td>
<td>Customer city</td>
</tr>
<tr>
<td>CUSTOMER_ZIP</td>
<td>CUSZIP</td>
<td>CHAR(5)</td>
<td>Customer ZIP code</td>
</tr>
<tr>
<td>CUSTOMER_CRED_LIM</td>
<td>CUSCRD</td>
<td>DEC(11,2)</td>
<td>Customer credit limit</td>
</tr>
<tr>
<td>CUSTOMER_TOT_AMT</td>
<td>CUSTOT</td>
<td>DEC(11,2)</td>
<td>Customer total amount</td>
</tr>
</tbody>
</table>
# Table 2-2  ORDERHDR table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDER_NUMBER</td>
<td>ORHNBR</td>
<td>CHAR(5)</td>
<td>Order number</td>
</tr>
<tr>
<td>CUSTOMER_NUMBER</td>
<td>CUSBR</td>
<td>CHAR(5)</td>
<td>Customer number</td>
</tr>
<tr>
<td>ORDER_DATE</td>
<td>ORHTE</td>
<td>DATE</td>
<td>Order date</td>
</tr>
<tr>
<td>ORDER_DELIVERY</td>
<td>ORHDLY</td>
<td>DATE</td>
<td>Order delivery date</td>
</tr>
<tr>
<td>ORDER_TOTAL</td>
<td>ORHTOT</td>
<td>DEC(11,2)</td>
<td>Order total</td>
</tr>
<tr>
<td>ORDER_SALESREP</td>
<td>SRNBR</td>
<td>CHAR(10)</td>
<td>Sales representative number</td>
</tr>
</tbody>
</table>

# Table 2-3  ORDERDTL table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDER_NUMBER</td>
<td>ORHNBR</td>
<td>CHAR(5)</td>
<td>Order number</td>
</tr>
<tr>
<td>PRODUCT_NUMBER</td>
<td>PRDNBR</td>
<td>CHAR(5)</td>
<td>Product number</td>
</tr>
<tr>
<td>ORDERDTL_QUANTITY</td>
<td>ORDQTY</td>
<td>DEC(5,0)</td>
<td>Order detail quantity</td>
</tr>
<tr>
<td>ORDERDTL_TOTAL</td>
<td>ORDTOT</td>
<td>DEC(9,2)</td>
<td>Order detail total</td>
</tr>
</tbody>
</table>

# Table 2-4  SALESREP table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALESREP_NUMBER</td>
<td>SRNBR</td>
<td>CHAR(10)</td>
<td>Sales representative number</td>
</tr>
<tr>
<td>SALESREP_NAME</td>
<td>SRNAM</td>
<td>CHAR(20)</td>
<td>Sales representative name</td>
</tr>
<tr>
<td>SALESREP_TELEPHONE</td>
<td>SRTEL</td>
<td>CHAR(15)</td>
<td>Sales representative telephone number</td>
</tr>
<tr>
<td>SALESREP_CITY</td>
<td>SRCTY</td>
<td>CHAR(20)</td>
<td>Sales representative city</td>
</tr>
<tr>
<td>SALESREP_MANAGER</td>
<td>SRMGR</td>
<td>CHAR(10)</td>
<td>Sales representative manager, that is also a sales representative</td>
</tr>
</tbody>
</table>

# Table 2-5  SALESCUS table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALESREP_NUMBER</td>
<td>SRNBR</td>
<td>CHAR(10)</td>
<td>Sales representative number</td>
</tr>
<tr>
<td>CUSTOMER_NUMBER</td>
<td>CUSBR</td>
<td>CHAR(5)</td>
<td>Customer number</td>
</tr>
<tr>
<td>SALES_AMOUNT</td>
<td>SRAMT</td>
<td>DEC(11,2)</td>
<td>Sales representative total amount for this customer</td>
</tr>
</tbody>
</table>
Table 2-6  SUPPLIER table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUPPLIER_NUMBER</td>
<td>SPLNBR</td>
<td>CHAR(5)</td>
<td>Supplier number</td>
</tr>
<tr>
<td>SUPPLIER_NAME</td>
<td>SPLNAM</td>
<td>CHAR(20)</td>
<td>Supplier name</td>
</tr>
<tr>
<td>SUPPLIER_TELEPHONE</td>
<td>SPLTEL</td>
<td>CHAR(15)</td>
<td>Supplier phone number</td>
</tr>
<tr>
<td>SUPPLIER_FAX</td>
<td>SPLFAX</td>
<td>CHAR(15)</td>
<td>Supplier fax number</td>
</tr>
<tr>
<td>SUPPLIER_ADDRESS</td>
<td>SPLADR</td>
<td>CHAR(20)</td>
<td>Supplier address</td>
</tr>
<tr>
<td>SUPPLIER_CITY</td>
<td>SPLCTY</td>
<td>CHAR(20)</td>
<td>Supplier city</td>
</tr>
<tr>
<td>SUPPLIER_ZIP</td>
<td>SPLZIP</td>
<td>CHAR(5)</td>
<td>Supplier ZIP code</td>
</tr>
</tbody>
</table>

Table 2-7  STOCK table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCT_NUMBER</td>
<td>PRDNBR</td>
<td>CHAR(5)</td>
<td>Product number</td>
</tr>
<tr>
<td>PRODUCT_DESC</td>
<td>PRDDES</td>
<td>CHAR(20)</td>
<td>Product description</td>
</tr>
<tr>
<td>PRODUCT_PRICE</td>
<td>PRDPRC</td>
<td>DEC(7,2)</td>
<td>Product unit price</td>
</tr>
<tr>
<td>PRODUCT_AVAIL_QTY</td>
<td>PRDQTY</td>
<td>DEC(5,0)</td>
<td>Product available quantity</td>
</tr>
<tr>
<td>SUPPLIER_NUMBER</td>
<td>SPLNBR</td>
<td>CHAR(4)</td>
<td>Supplier number</td>
</tr>
<tr>
<td>PRODUCT_CATEGORY</td>
<td>PRDCAT</td>
<td>CHAR(4)</td>
<td>Product category</td>
</tr>
<tr>
<td>PROD_MIN_STOCK_QTY</td>
<td>PRDQTM</td>
<td>DEC(5,0)</td>
<td>Product minimum stock quantity</td>
</tr>
</tbody>
</table>

Table 2-8  STOCKPIC table

<table>
<thead>
<tr>
<th>Field name</th>
<th>Alias</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCT_NUMBER</td>
<td>PRDNBR</td>
<td>CHAR(5)</td>
<td>Product number</td>
</tr>
<tr>
<td>PRODUCT_PICTURE</td>
<td>PRDPIC</td>
<td>BLOB</td>
<td>Product picture</td>
</tr>
</tbody>
</table>

Our database also contains several views that are primarily used by stored procedures. These are listed in Table 2-9 through Table 2-11.

Table 2-9  SALES view

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>INTEGER</td>
<td>Order year</td>
</tr>
<tr>
<td>MONTH</td>
<td>INTEGER</td>
<td>Order month</td>
</tr>
<tr>
<td>SUPPLIER_NAME</td>
<td>CHAR(20)</td>
<td>Supplier name</td>
</tr>
<tr>
<td>SALES</td>
<td>DECIMAL(11,2)</td>
<td>Sales</td>
</tr>
</tbody>
</table>
Table 2-10  TOTALSALES view

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>INTEGER</td>
<td>Order year</td>
</tr>
<tr>
<td>MONTH</td>
<td>INTEGER</td>
<td>Order month</td>
</tr>
<tr>
<td>SUPPLIER_NAME</td>
<td>CHAR(20)</td>
<td>Supplier name</td>
</tr>
<tr>
<td>TOTALSALES</td>
<td>DECIMAL(11,2)</td>
<td>Total sales</td>
</tr>
</tbody>
</table>

Table 2-11  YEARSALES view

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>INTEGER</td>
<td>Order year</td>
</tr>
<tr>
<td>SUPPLIER_NAME</td>
<td>CHAR(20)</td>
<td>Supplier name</td>
</tr>
<tr>
<td>TOTALSALES</td>
<td>DECIMAL(11,2)</td>
<td>Total sales</td>
</tr>
</tbody>
</table>

2.3 Stored procedures and triggers in the Order Entry database

Figure 2-3 shows the Order Entry database structure and the tables to which triggers have been defined.

Figure 2-3  Order Entry application database structure
As stated in the overview of this chapter, the main objective of presenting this application scenario along with this specific database design is to show how the stored procedures and triggers provided with DB2 Universal Database for iSeries can be used and how they can work together in a single application. Let us analyze Figure 2-3 from each function standpoint.

### 2.3.1 Stored procedures

Figure 2-3 shows a stored procedure associated to the remote physical table, STOCK. The purpose of this procedure is to update the available quantity in the STOCK table and to look for a replacement when the required product is not available.

This function was implemented in a stored procedure to speed up performance. Instead of issuing several SQL statements from the local system, we call the stored procedure and wait for the result. This implementation reduces the network traffic and simplifies the logic of the client application.

### 2.3.2 Triggers

As shown in Figure 2-3, we defined three trigger programs (two of them in the ORDERHDR table and one in the CUSTOMER table). In our scenario, note these points:

- When a sales representative inserts a new order for a certain customer, we want to check that the sales representative is authorized to deal with that customer. In addition, we want to keep track of any attempts to violate the rule.
- When the order has been completed and accepted by the customer, we want to print the related invoice.
- If the total amount of an order exceeds 90 percent of the customer credit limit, a fax is sent to the customer or a message is inserted into the job log. If the customer belongs to a privileged group, recognized by a customer number starting with the digit 9, the credit limit is automatically increased by 30 percent.

Since we want these functions performed each time an ORDERHDR insertion, an ORDERHDR update, or a CUSTOMER update takes place, we associate an Insert Trigger to the ORDERHDR table and an Update trigger to the CUSTOMER table.

### 2.3.3 User-defined functions

UDFs were used to enhance the business logic illustrated with the following examples:

- A casting UDF is required to convert from DECIMAL(8) date representation to DATE: This hypothetical company has other legacy systems that have several dates stored as eight digits decimals in YYYYMMDD format. To enhance the support of DB2 Universal Database for iSeries a UDF was added to support the casting function. It is also required that when the input parameter contains an invalid date, it returns a null value and signals a user-defined warning message with SQLSTATE 01HDI.
- A table UDF is required to show a certain number of top preforming sales people.
Chapter 3. Introduction to the SQL Persistent Stored Module in DB2 Universal Database for iSeries

This chapter describes the “scripting” language for declaring and maintaining persistent database language procedures and invoking them from programs written in a standard programming language. The support of stored procedures in DB2 Universal Database for iSeries allows you to write stored procedures using extensions to the SQL language based on the ANSI/ISO Persistent Stored Module (PSM) standard specification.

DB2 Universal Database for iSeries was the first member of the DB2 family to support the SQL procedural language and make this language available for the development of SQL user-defined functions (UDFs) and SQL triggers. The greatest advantage to using the SQL PSM is portability. It is much easier to port stored procedures, triggers and UDFs to other relational database management systems (RDBMS).

This chapter covers:
- The ANSI-SQL implementation of DB2 Universal Database for iSeries SQL procedures
- Language elements of SQL PSM
- Transaction management in SQL PSM
- Debugging
3.1 Introduction

Implementation of the Persistent Stored Module extensions for SQL in DB2 Universal Database for iSeries is based on the SQL standard and supports constructors that are common to most programming languages. It supports the declaration of local variables, statements to control the flow of the procedure, assignment of expression results to variables, receiving and returning of parameters, returning result sets, and error handling.

The way in which SQL PSM programs are called depends on how they are being used. SQL PSM may be used in:

- **Stored procedures that are called using CALL statements**, as described in Part 2, “Stored procedures” on page 61
- **Triggers**: In this case, the SQL PSM implementing the trigger will be automatically called by DB2 Universal Database for iSeries when the trigger event occurs, as described in Part 3, “Triggers” on page 279.
- **UDFs**: The SQL PSM implementing the UDF will be automatically called by DB2 Universal Database for iSeries for each usage of the UDF, as described in Part 4, “User-defined functions” on page 447.

3.2 System requirements and planning

Before you start to develop stored procedures, triggers and UDFs using SQL PSM on DB2 Universal Database for iSeries, make sure that you are running on the proper OS/400 level considering:

- SQL stored procedures were introduced in V4R2.
- SQL UDFs were introduced in V4R4.
- SQL triggers were introduced in V5R1.

When you execute the CREATE PROCEDURE, CREATE TRIGGER, or CREATE FUNCTION statement for a SQL PSM procedure, trigger or UDF, DB2 Universal Database for iSeries walks through a multiphase process to create an ILE C program object (*PGM). During this process, DB2 Universal Database for iSeries generates an intermediary ILE C code with embedded SQL statements. This ILE C code is then precompiled, compiled, and linked automatically.

In V4R5 and prior releases, the following products need to be installed on the system where you plan to develop SQL PSM code:

- SQL Development Kit for iSeries
- ILE C Compiler

In V5R1, the SQL Development Kit for iSeries must be installed on the system where you plan to develop SQL PSM code.

In V5R2, no additional software is required to be installed in the system where you plan to develop SQL PSM code, other than OS/400.

After the ILE C object is created, it can be restored onto any V4R2, or later, system and run without the SQL Development Kit and ILE C compiler.
3.3 Structure of a SQL PSM program

An SQL PSM program consists of:

- A procedure, trigger or UDF name.
- On stored procedures and UDFs, a sequence of parameter declarations. SQL triggers do not have parameter declarations.
- The procedure, trigger or UDF properties. These characteristics will vary regarding the usage of the SQL PSM program, as explained in the respective chapters of this book.
- A set of options that control the way in which the stored procedure, trigger or UDF is created.
- A routine body. An SQL routine body is a single or a compound SQL statement. A compound statement is a set of single or compound statements enclosed by BEGIN and END clauses.

The general structure of an SQL procedure, trigger and UDF will be explained in each of the corresponding chapters.

For a complete syntax of the SQL procedure, refer to the “SQL procedure” chapter in SQL Reference, SC41-5612. Let us look at some examples to illustrate the basic concepts.

3.4 SQL control statements

DB2 Universal Database for iSeries provides a set of programming constructs (syntactic structures used to write procedural code). In DB2 Universal Database for iSeries, these programming constructs are called control statements. A control statement is one of the statements that can be placed in the routine body of an SQL PSM. Among these constructs, there are:

- Assignment statements
- Conditional control statements
- Iterative control statements
- Calling external procedures
- Compound statements

3.4.1 Assignment statement

In SQL PSM, the assignment statement is used to assign values to inout and output parameters and to local variables. The following rules apply for using the assignment statement:

- The assignment statements in SQL PSM must conform to the SQL assignment rules and to the SQL arithmetic operators. The data type of the target and source must be compatible.
- When a string is assigned to a variable and the string is longer than the length attribute of the variable, a negative SQLCODE is set.
- If truncation of the whole part of the number occurs on assignment to a numeric variable, a negative SQLCODE is set.

**Important:** If you use SET to assign a value to an SQL procedure parameter, it must be an output or inout parameter. It must be a parameter that is returned to the calling program. UDFs parameters are always defined as input parameters.
The following code illustrates simple assignment statements:

```sql
SET num_records = num_records + 1;
SET credit_limit = credit_limit * 1.20;
SET num_orders = NULL;
SET max_credit_limit = (SELECT max(credit_limit) FROM customer);
```

You should also notice that repeated assignment operations in a SET statement are not aware of the calculations in the preceding assignment. For example, in the following code, the value of the variable v2 is 3 after the last SET statement is run, not 4:

```sql
DECLARE v1, v2 INT;
SET v1=0, v2=0;  \<--- Repeated assignments
SET v1=1, v2=v1+3;  \<--- Repeated assignments
```

This is different from the following code in which case the value of the variable v22 is 4 after the last SET statement is run:

```sql
DECLARE v11,v22 INT;
SET v11=0, v22=0;
SET v11=1;
SET v22=v11+3;
```

Note: Assigning values to multiple variables on a single SET statement is usually a good technique for performance.

Suppose that you want to construct an SQL SELECT statement using parameters passed to the SQL procedure by the calling program. In this case, you may use the CONCATENATE operator to “glue” the strings together:

```sql
CREATE PROCEDURE MYMAX
(IN fld_name CHAR(30),
 IN file_name CHAR(128))
LANGUAGE SQL
BEGIN atomic
DECLARE sql_stmt CHAR(128);
...
SET sql_stmt = ' SELECT ' || fld_name || ' FROM ' || file_name || ' ORDER BY 1';
...
END
```

### 3.4.2 Conditional control

Many times you have to test conditions in your programs. SQL PSM has four conditional control structures:

- **IF-THEN**
- **IF-THEN-ELSE**
- **IF-THEN-ELSEIF**
- **CASE**

**IF-THEN**

The IF-THEN conditional control structure tests a simple condition. If the condition evaluates to TRUE, one or more lines of code are executed. If the condition evaluates to FALSE, the control of the program is passed to the next statement after the test. The following example illustrates this case:
IF ref_error = 1
  THEN SET o_error = 'NOT FOUND';
END IF;

**IF-THEN-ELSE**

The IF-THEN-ELSE conditional control structure is similar to the IF-THEN structure. The difference is that when the condition evaluates to FALSE, one or more statements following the ELSE keyword are executed. The following example illustrates this case:

IF ref_error = 0
  THEN SET o_error = 'FOUND';
ELSE
  SET o_error = 'NOT FOUND';
END IF;
The statement can be nested as illustrated in the following example:

IF evaluation = 100
  THEN SET new_salary = salary * 1.3;
ELSE
  IF evaluation >= 90
    THEN SET new_salary = salary * 1.2;
  ELSE
    SET new_salary = salary * 1.1;
  END IF;
END IF;

**IF-THEN-ELSEIF**

The IF-THEN-ELSEIF conditional control structure is an alternative to using the nested IF-THEN-ELSE structure. Let us rewrite the previous example:

IF evaluation = 100
  THEN SET new_salary = salary * 1.3;
ELSEIF evaluation >= 90
  THEN SET new_salary = salary * 1.2;
ELSE
  SET new_salary = salary * 1.1;
END IF;

There is no matching END IF with each ELSEIF.

**CASE**

The fourth conditional control structure is the CASE structure, which permits you to select an execution path based on multiple cases. There are two ways to code it.

Let us look at an example using the first case:

CASE evaluation
  WHEN 100 THEN UPDATE employee SET salary = salary * 1.3;
  WHEN 90 THEN UPDATE employee SET salary = salary * 1.2;
  WHEN 80 THEN UPDATE employee SET salary = salary * 1.1;
  ELSE UPDATE employee SET salary = salary * 1.05;
END CASE;

**Note:** We suggest that you use indentation, which makes the programs easier to follow and read.
Here is an example using the second case:

```sql
CASE
  WHEN evaluation = 100 THEN UPDATE employee SET salary = salary * 1.3;
  WHEN evaluation = 90 THEN UPDATE employee SET salary = salary * 1.2;
  WHEN evaluation = 80 THEN UPDATE employee SET salary = salary * 1.1;
  ELSE UPDATE employee SET salary = salary * 1.05;
END CASE;
```

If none of the conditions specified in the WHEN statement are true and an ELSE is not specified, an error is issued, and the execution of the case statement is terminated.

Case expressions can be nested. The current implementation of SQL procedures allows three levels of nesting for the first case.

### 3.4.3 Iterative control

While writing programs, you can repeat a series of instructions for a certain number of times or until a condition is satisfied. These programming structures are called *iterative control statements*. They are also known as *looping structures*. In DB2 Universal Database for iSeries, there are four iterative control statements and an exit statement:

- **LOOP**
- **WHILE**
- **REPEAT**
- **FOR**
- **LEAVE**
- **ITERATE**

**LOOP**

The first iterative control statement is **LOOP**. Before we show you the syntax, we define **LOOP** as a programming construct that permits you to execute a series of instructions repeatedly. Example 3-1 shows how a **LOOP** control statement may be used.

**Example 3-1** A **LOOP** control statement

```sql
OPEN c1;
SET at_end = 0;
SET numrec = 0;
fetch_loop: LOOP
  FETCH c1 INTO proc_cusnbr, proc_cuscrd;
  IF SQLCODE = 0 THEN
    SET proc_cuscrd = proc_cuscrd * 1.2;
    UPDATE ordapplib.customer
      SET cuscrd = proc_cuscrd
      WHERE CURRENT OF c1;
    SET numrec = numrec + 1;
  ELSE
    LEAVE fetch_loop;
  END IF;
END LOOP fetch_loop;
CLOSE c1;
```
There are two methods to get out of a loop. You can code a LEAVE statement within the loop or declare a handler. Both methods are discussed later in this chapter.

**WHILE**

With this programming structure, the statements that are between the WHILE and END WHILE are executed while the specified condition is true. It is important to note that the exit condition is checked in the WHILE condition each time an iteration is going to start. The exit condition must be set in some place of the iteration. If this is not done by mistake, you may end up with an endless execution program.

Consider Example 3-2 where the instructions between the WHILE and END WHILE are executed, while the condition that is at_end = 0 is true.

**Example 3-2  WHILE and END WHILE**

```sql
OPEN c1;
SET at_end = 0;
SET numrec = 0;
WHILE at_end = 0 DO
  FETCH c1 INTO proc_cusnbr, proc_cuscrd;
  IF SQLCODE = 0 THEN
    SET proc_cuscrd = proc_cuscrd * 1.2;
    UPDATE ordapplib.customer
    SET cuscrd = proc_cuscrd
    WHERE CURRENT OF c1;
    SET numrec = numrec + 1;
  ELSE
    SET at_end = 1;
  END IF;
END WHILE;
CLOSE c1;
```

**REPEAT**

With this programming structure, the statements that are between the REPEAT and END REPEAT are executed until the specified condition is true. It is important to note that the exit condition is checked differently than the WHILE statement. In the WHILE statement, the exit condition is checked at the beginning of the iteration. However, on the REPEAT statement, this exit condition is tested at the end of the iteration. This means that at least one iteration is executed.

**Example 3-3  REPEAT and END REPEAT**

```sql
SET numrec = 0;
fetch_loop:
REPEAT
  FETCH c1 INTO proc_cusnbr, proc_cuscrd;
  IF SQLCODE = 0 THEN
    SET proc_cuscrd = proc_cuscrd * 1.2;
    UPDATE ordapplib.customer
    SET cuscrd = proc_cuscrd
    WHERE CURRENT OF c1;
  END IF;
END REPEAT;
```

**Notes:** The following notes refer to Example 3-1:

1. A label can be defined in a LOOP statement.
2. A LEAVE statement or a handler has to be specified to avoid an endless execution.
3. Every LOOP statement ends with an END LOOP clause.
```
SET numrec = numrec + 1;
END IF;
UNTIL SQLCODE <> 0
END REPEAT fetch_loop;
```

**Notes**: The following notes refer to Example 3-3:

1. The REPEAT statement is executed at least one time because the exit condition is checked at the end of the cycle.
2. The condition is checked at the end of the cycle.
3. Every REPEAT statement ends with an END REPEAT clause.

**FOR**

The FOR statement is another type of iterative structure. It executes a statement for each row of a table. It is used for processing every row of a select statement. The important difference of this structure with WHILE, REPEAT, and LOOP is that FOR is only used for processing rows of tables, while the others can be used for general purpose functions.

**Note**: The SQL procedure statement cannot include an OPEN, FETCH, or CLOSE specifying the cursor name of the FOR statement. Also, the cursor name must not be the same as the name of another cursor declared in the SQL procedure. If it is not specified, a unique cursor is generated.

**Example 3-4 FOR statement**

```
for_loop: FOR each_record AS
  cursor1 CURSOR FOR
    SELECT cusnbr, cuscrd FROM ordapplib.customer
  DO
    UPDATE ordapplib.customer
    SET cuscrd = cuscrd * 1.1
    WHERE CURRENT OF cursor1;
  END FOR;
```

**Notes**: The following notes refer to Example 3-4:

1. The each_record is known as variable name, while for_loop is known as label. Variable name and label are used in the FOR structure to qualify variables that refers to values of the columns in the FOR associated cursor. In the example, the columns `cusnbr` and `cuscrd` may be referenced as `each_record.cusnbr` or `for_loop.cuscrd`. Previous to V5R2, only the label qualification was supported.
2. Here you define the cursor that is used for accessing the file. In this case, it is the CUSTOMER file.
3. This is the statement that is executed for each row of the CUSTOMER file.
4. The FOR cursor can be referenced in the WHERE CURRENT OF clause.
5. Every FOR statement ends with an END FOR statement.

Columns in the cursor associated with the FOR statement are referenced as variables in the stored procedure, qualifying the column name with the label or the SQL variable name, as shown in Example 3-5.
Example 3-5  Columns referenced as variables in the FOR statement

```sql
for_loop:
FOR for_loop_example AS c1 CURSOR FOR
  SELECT DEPTNO, UPPER(DEPTNAME) AS DEPTNAME,
       'REPORTS TO DEPARTMENT ' || ADMRDEPT AS REPORTS_TO
  FROM SAMPLEDB01.DEPARTMENT DO
      INSERT INTO sampledb01.deptnew (deptno, deptname, reportline)
      VALUES (for_loop.deptno, for_loop.deptname, for_loop.reports_to);
END FOR;
```

Notes: The following notes refer to Example 3-5:

1. This is a label name that corresponds to the LOOP block.
2. SQL variable name.
3. Functions and expressions may be used. Explicit names should be provided to be able to reference that column.

The following restrictions apply to V5R1 and earlier: FOR statements can only handle columns in the select list; expressions and functions are not allowed; and column values for the current cursor position may be referenced by the column name optionally qualified by the label. Column names qualified by the SQL variable name of the FOR statement are not supported.

In V5R3, WITH HOLD syntax support was added into the FOR loop construct. This syntax was available before V5R3 in several SQL statements, but not in the FOR loop.

If a cursor is not declared using the WITH HOLD syntax, the cursor will be closed at every COMMIT operation and release all program resources such as its current position and locks that it holds. Reopening a cursor puts it back to the first row of the table, and you need to readjust its position to continue from the latest transaction. This can be inefficient in several situations. The WITH HOLD syntax support in the FOR loop enables the cursor to preserve its resources at COMMIT time. This is useful in the FOR loop that often issues COMMIT.

**LEAVE**

The LEAVE statement permits you to go out of a block or loop statement. The execution continues with the first statement following the loop or block statements. As shown in the LOOP example, you may use LEAVE to go out of a loop.

Example 3-6  LOOP with a LEAVE statement

```sql
OPEN c1;
SET at_end = 0;
SET numrec = 0;
fetch_loop:
  LOOP
    FETCH c1 INTO proc cusnbr, proc cuscrd;
    IF SQLCODE = 0 THEN
      SET proc cuscrd = proc cuscrd * 1.2;
      UPDATE ordapplib .customer
      SET cuscrd = proc cuscrd
      WHERE CURRENT OF c1;
      SET numrec = numrec + 1;
    ELSE
```
ITERATE

The ITERATE statement, introduced in V5R2, forces the next iteration in a loop structure. In Example 3-7, the procedure processes all rows, except for those where the department is D11. When the department code is equal to D11, the statement in 2 sends the program to the next iteration of the loop construction identified by 1.

Example 3-7  ITERATE processing all rows with exception

```
OPEN c1;
ins_loop:
  LOOP
    FETCH c1 INTO v_dept, v_deptname, v_admdept;
    IF at_end = 1 THEN
      LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
      ITERATE ins_loop;
    END IF;
    INSERT INTO sampledb02.deptnew (deptno, deptname, admrdept)
    VALUES (v_dept, v_deptname, v_admdept);
  END LOOP;
CLOSE c1;
```

For versions of DB2 Universal Database for iSeries prior to V5R2, this code may be ported substituting the ITERATE statement by a GOTO statement, except when it is used into a FOR loop construction. In those cases, you cannot use GOTO to force the next iteration.

Example 3-8  GOTO statement

```
OPEN c1;
ins_loop:
  LOOP
    FETCH c1 INTO v_dept, v_deptname, v_admdept;
    IF at_end = 1 THEN
      LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
      GOTO ins_loop;
    END IF;
    INSERT INTO sampledb02.deptnew (deptno, deptname, admrdept)
    VALUES (v_dept, v_deptname, v_admdept);
  END LOOP;
CLOSE c1;
```
3.4.4 Calling procedures

From an SQL PSM, you can call stored procedures by using the CALL control statement. Example 3-9 shows the statements to call a stored procedure from an SQL PSM.

Example 3-9  Calling a stored procedure from an SQL PSM

```sql
SET totsales = 0;
FOR each_record AS
    cursor1 CURSOR FOR
        SELECT cusnbr, cussal FROM customer
    DO
        SET totsales = totsales + cussal;
    END FOR;
CALL PROC1 (totsales, totcust);
```

In this example, the stored procedure PROC1 is called after all the rows in the CUSTOMER table are read within the FOR statement. The PROC1 stored procedure may be an SQL or an external stored procedure (an external stored procedure is a stored procedure written in a high level language). In this particular example, it is an ILE RPG program that receives two parameters.

External stored procedures are the only way to access system APIs or CL commands. The following external call can be used to execute the CHGCURLIB command:

```sql
CALL QSYS.QCMDEXC('chgcurlib curlib(saleslib)',000000023.00000);
```

Here QSYS.QCMDEXC is an external stored procedure supplied with DB2 Universal Database for iSeries that invokes the command string specified in the first parameter, whose length is specified in the second parameter.

**Note:** Too many nested stored procedure calls within a procedure can impact performance, so use nested calls carefully.

3.5 Compound SQL statement

According to the SQL PSM specification, an SQL routine (that is, a stored procedure or a function) consists of a single SQL statement. The specification also introduces the concept of a compound statement. A compound statement consists of a BEGIN and END block and any number of SQL statements contained within the block. The body of an SQL routine is, in fact, a compound statement. Example 3-10 shows the general structure of a compound statement. Note that the various constructs must be ordered as listed.

Example 3-10  Structure of a compound SQL statement

```sql
BEGIN
    <variable declarations>
    <cursor declarations>
    <condition handler declarations>
    <SQL statement list/procedure logic>
END
```

Most of the time you need the SQL routine body of an SQL procedure to have more than one statement. In this case, it is useful to use compound statements, as we describe next.
The SQL code in Example 3-11 includes several control statements: SET, FOR, and CALL. The usual way to put multiple control statements in a procedure body is to use a compound control statement. Note that BEGIN and END are delimiters of the procedure body.

**Example 3-11  Multiple control statements**

```sql
BEGIN
    DECLARE totsales DECIMAL(11,2);
    DECLARE totcust DECIMAL(11,2);
    SET totsales = 0;
    FOR each_record AS
        cursor1 CURSOR FOR 
            SELECT cusnbr, cussal FROM customer
        DO
            SET totsales = totsales + cussal;
        END FOR;
    CALL PROC1 (totsales);
END
```

The general structure of a compound statement is:

```sql
label: BEGIN
    [ <local declaration list> ]
    [ <local cursor declaration list> ]
    [ <local handler declaration list> ]
    [ <SQL statement list> ]
END
```

When the compound statement is used, it must follow this order:

1. Local variable declarations
2. Local cursor declarations
3. Local handler declarations
4. SQL procedure logic

### 3.5.1 Nested compound statements

Since V5R2, DB2 Universal Database for iSeries has supported nested compound statements, which are typically used in complex routines to scope constructs such as variables, cursor declarations, and condition handlers. Only the constructs that were defined within the same or enclosing compound statements are visible. SQL statements within a compound statement may not be able to refer to constructs that are declared within another compound statement.

Nested compound statements are specially useful to improve flexibility in programming and readability in the source code, for example declaring error handlers, as shown in Chapter 7, “Java stored procedures” on page 173. Before the introduction of nested compound statements in V5R2, in some cases it can be circumvented using a “dummy” loop, with the use of LOOP and LEAVE.

### 3.5.2 Variable declaration

While coding the SQL compound statement, you can declare local variables that can be used for different purposes:

- Calculations
- Assignment to output parameters
- Assignment to columns for database updates
As input parameters passed by calling programs

Error handling

You can define any valid SQL data type as shown in Example 3-12.

**Example 3-12  Defining a valid SQL data type**

```sql
BEGIN
    DECLARE sqlcode             INTEGER;
    DECLARE total_sales         DECIMAL(11,2);
    DECLARE number_customer     DECIMAL(5);
    DECLARE err_msg             CHARACTER(10);
    DECLARE timestamp_order     TIMESTAMP;
    DECLARE date_order          DATE;
    DECLARE picture             BLOB(10M);

    ......
END
```

Every time you declare a variable, you define the name of the variable and the type of information that the variable holds. How values are interpreted depends on the data type they represent.

**Data types supported by SQL procedures**

Generally, all the data types supported on the CREATE TABLE statement are also supported by the SQL stored procedures. Refer to *SQL Reference*, SC41-5612, for details.

Data structures, like arrays, are not supported, which means that SQL stored procedures cannot perform blocked inserts and fetches.

### 3.5.3 Using cursors

When SQL runs a SELECT statement, the resulting rows comprise the result table. A *cursor* provides a way to access a result table. The cursor is used to maintain a position in the result table. SQL uses a cursor to work with the rows in the result table and make them available to the program. Look at the following skeleton of the SQL procedure, which employs a cursor to process the resulting table row-by-row. The code shown in Example 3-13 will run in an endless loop since it is missing the condition and handler declarations discussed later in this chapter.

**Example 3-13  Code missing the condition and handler declarations**

```sql
CREATE PROCEDURE CREDITP
    (IN i_perinc DECIMAL(3,2),
     INOUT o_numrec DECIMAL(5,0))
    LANGUAGE SQL
BEGIN atomic
    DECLARE proc_cusnbr CHAR(5);
    DECLARE proc_cuscrd DECIMAL(11,2);
    DECLARE numrec      DECIMAL(5,0);
    DECLARE at_end INT DEFAULT 0;

    DECLARE c1 CURSOR FOR
        SELECT cusnbr, cuscrd
        FROM ordapplib.customer;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET at_end = 1;
```
SET numrec = 0;
OPEN c1;
FETCH c1 INTO proc_cusnbr, proc_cuscrd;
WHILE at_end = 0 DO
  SET proc_cuscrd = proc_cuscrd +(proc_cuscrd * i_perinc);
  UPDATE ordapplib.customer
    SET cuscrd = proc_cuscrd
    WHERE CURRENT OF c1;
  SET numrec = numrec + 1;
  FETCH c1 INTO proc_cusnbr, proc_cuscrd;
END WHILE;
SET o_numrec = numrec;
CLOSE c1;
END

Notes: The following notes refer to Example 3-13:
1. The procedures name is CREDITP.
2. It defines the input parameter i_perinc.
3. It defines the input-output parameter o_numrec.
4. The compound statement starts with a BEGIN clause. If you specify atomic with the
   BEGIN clause, it means that if an unhandled error occurs in the compound statement,
   all SQL statements executed in the compound statement are rolled back.
5. Local variables are declared in this section.
6. A cursor for the SELECT statement is declared.
7. To avoid a never-ending loop, an error handler was defined to set the AT_END variable
   to 1. Condition and Handler declarations are discussed in 7.2.1, “Parameter styles” on
   page 174.
8. The cursor is opened.
9. The first row from the customer table is read.
10. The main procedure loop starts here with the WHILE control statement.
11. Every row read is updated with the new value for the cuscrd field.
12. The next row is read from the customer table.
13. The output variable is set to its value.
14. The cursor is closed, and the compound statement ends.

Avoiding orphaned locks caused by a cursors left open

The open cursor results in a read lock held over the data space. This lock is required to
preserve the integrity of the open cursor structure that points to that data space. The open
cursor is not implicitly closed upon the return from a stored procedure. Two default compile or
link time parameters (in CREATE FUNCTION, CREATE PROCEDURE, CREATE TRIGGER,
DECLARE CURSOR, PREPARE, SET OPTION) are responsible for this behavior:

- **CLOSSQLCSR(’ENDACTGRP)**: This parameter means that the open cursors are closed
  when the activation group ends.
- **ACTGRP(’CALLER)**: This parameter means that a stored procedure runs within the same
  activation group as the calling process. This is required for performance reasons.
  Consequently, the cursors that are not closed in a stored procedure are still open upon
  return to the caller process (job).

Currently, these parameters cannot be easily changed for an SQL stored procedure.
Under certain circumstances, the described behavior can result in locks that persist as long as a given job (activation group) is active. We illustrate this issue in “Using a nested error handler to avoid locks caused by cursors left open” on page 240.

### 3.6 Using scrollable cursors in SQL PSM

In Example 3-14, we show you a simple SQL PSM that implements a scrollable cursor. The procedure finds the maximum value for a given column name and a given table name passed as input parameters. The assumption is made that there is an ascending access path built over the numeric column.

**Example 3-14   Simple SQL PSM procedure that implements a scrollable cursor**

```sql
CREATE PROCEDURE MYMAX
  ( IN   fld_name CHAR(30),
  IN   file_name CHAR(128),
  INOUT  max_value INTEGER)
LANGUAGE SQL
BEGIN atomic
  DECLARE sql_stmt  CHAR(256);
  DECLARE not_found
  CONDITION FOR '02000';
  DECLARE c1 DYNAMIC SCROLL CURSOR FOR s1;
  DECLARE CONTINUE HANDLER FOR not_found
  SET max_value = NULL;
  SET sql_stmt = 'SELECT ' || fld_name || ' FROM ' || file_name ||
  ' ORDER BY 1';
  PREPARE s1 FROM sql_stmt;
  OPEN c1;
  FETCH LAST FROM c1 INTO max_value;
  CLOSE c1;
END
```

**Notes:** The following notes refer to Example 3-14:

1. The procedure defines three parameters:
   - `fld_name`: Name of the column for which the max value is to be found.
   - `file_name`: Name of the table containing the field.
   - `max_value`: The highest numeric value found for the column specified. The max value is passed back to the calling program.

2. The dynamic scrollable cursor is defined for the select statement used to find the max value.

3. The select statement is constructed using the parameters passed from the calling program.

4. The FETCH LAST statement is used to move to the last record.

The procedure returns a null value when the table is empty. Therefore, the calling program is responsible for passing the null indicator for the third parameter. To call the procedure from an ILE C program, define the indicator in the DECLARE SECTION as shown in Example 3-15.
EXEC SQL BEGIN DECLARE SECTION;
    char           fld_name[ 30 ];
    char           file_name[ 128 ];
    integer        max_value;
    short          ind3;
EXEC SQL END DECLARE SECTION;
Then the indicator variable is used in the call statement:
EXEC SQL
CALL MYMAX( :fld_name, :file_name, :max_value :ind3);

3.7 Dynamic SQL in SQL PSM

There are two ways in which SQL data manipulation statements can be used: Static SQL and
dynamic SQL. Dynamic SQL is characterized by statements that are known only at run time,
as when a statement is built on the program logic and then it is executed.

To deal with dynamic SQL, the following statements are in the procedural SQL provided by
DB2 Universal Database for iSeries:

- **PREPARE**
- **EXECUTE**
- **EXECUTE IMMEDIATE**

### 3.7.1 PREPARE statement

This statement takes a varchar variable containing an SQL statement and converts it into an
executable statement. Internally, a dynamic SQL statement must be first prepared for
execution and then it may be executed. When you expect the same SQL statement to be
executed multiple times, it is convenient to prepare it first and then use it. This makes the
program more efficient.

In Example 3-16, we prepare an UPDATE statement to be executed multiple times in an SQL
procedure.

**Example 3-16  UPDATE statement to be run multiple times in an SQL procedure**

CREATE PROCEDURE DYNSQLSAMPLE()
LANGUAGE SQL
BEGIN
    DECLARE stmt VARCHAR(256);
    SET stmt = 'UPDATE employee SET salary = salary * 1.1 WHERE empno = ?';
    PREPARE s1 FROM stmt;
    ins_loop:
    FOR each_department AS
        c1 CURSOR FOR
            SELECT mgrno FROM department WHERE mgrno IS NOT NULL
        DO
            EXECUTE s1 USING mgrno;
        END FOR;
    END;

The variable stmt is set to a string representing an SQL statement in 1. The question mark is
called a **parameter marker** and represents a value that will be substituted at execution time. In
2, the statement is prepared for execution.
3.7.2 EXECUTE statement

The EXECUTE statement allows you to execute a previously prepared statement. You can execute the same statement multiple times. For example, in Example 3-16, the statement prepared in 2 is executed, substituting the parameter marker by the content of the variable mgrno.

3.7.3 EXECUTE IMMEDIATE statement

This EXECUTE IMMEDIATE statement prepares and executes a statement represented in a string or a variable containing a string.

Consider the following statement:

EXECUTE IMMEDIATE 'UPDATE employee SET salary = salary * 1.1 WHERE empno IN (SELECT DISTINCT mgrno FROM department WHERE mgrno IS NOT NULL);

It is equivalent to this statement:

PREPARE s1 FROM 'UPDATE employee SET salary = salary * 1.1 WHERE empno IN (SELECT DISTINCT mgrno FROM department WHERE mgrno IS NOT NULL);
EXECUTE s1;

3.7.4 Cursors based on dynamic SQL

Cursor declaration can also be based on dynamic SQL, as shown in the following example:

... DECLARE stmt VARCHAR[256]; ... SET stmt = 'SELECT COLUMN1, COLUMN2, COLUMN3 FROM TBL1'; PREPARE PreparedStatement FROM stmt; DECLARE Cursor1 CURSOR FOR PreparedStatement; ...
If DB2 Universal Database for iSeries finds one or more procedures with the same name and a different signature (for example, different parms), then the restored program will be registered as a procedure with the same name (and possibly overlay the program object for the existing procedure).

When parameters have changed, it is probably best to drop the existing procedure before the restore.

Note: The SQL procedure is registered and created in the target library specified by the restore command, regardless of the CREATE PROCEDURE source.

If SPECIFIC was specified during the CREATE PROCEDURE statement, then it must be unique among the system.

The program object will be restored even if it cannot be registered as a stored procedure into the iSeries server.

3.8.2 Restore processing for V5

In V5, DB2 Universal Database for iSeries tries to automatically recognize the C program on the restore as an SQL PSM, but there are exceptions:

- If DB2 Universal Database for iSeries does not find a matching object in the catalogs, then the C program is registered as an SQL object.
- If DB2 Universal Database for iSeries finds one object with the same name (differences in parameters ignored), catalog entries for the existing object are dropped and the new program object is registered as an SQL object.
- If DB2 Universal Database for iSeries finds one or more objects with the same name and a different signature (for example, different parameters), then the restored program will be registered as an object with the same name (and possibly overlay the program object for the existing object).

When parameters have changed, it is probably best to drop the existing procedure or UDF before the restore.

3.9 Adopted authorities in SQL PSM

When an SQL PSM is created, the intermediate C program is generated using the options specified in the RUNSQLSTM.

The default value for the USRPRF attribute depends on the naming convention user. If *SQL naming convention is used, then *OWNER is the default value for USRPRF. If *SYS naming convention is used, then *USER is the default value for USRPRF. The USRPRF attribute can be controlled independently of the naming convention. In Example 3-17, a USRPRF value of *OWNER is explicitly specified.

Note: An exact match for a procedure means that the procedure name and the number of parameters coincide. The parameter types are ignored. The exact match for a UDF means that the UDF name, number of parameters, and type of parameters coincide. The result type is ignored. The exact match for a trigger means that the trigger name and table name coincide.
Example 3-17  Specifying the USRPRF value of *OWNER

CREATE PROCEDURE ITERATOR2()
LANGUAGE SQL
SET OPTION USRPRF=*OWNER
BEGIN
  ins_loop:
  FOR each_department AS
c1 CURSOR FOR
    SELECT deptno, deptname, admrdept
    FROM sampledb02.department
    WHERE deptno <> 'D11'
    ORDER BY deptno
  DO
    INSERT INTO sampledb02.deptnew (deptno, deptname, admrdept)
    VALUES (deptno, deptname, admrdept);
  END FOR;
END;

You can verify the USRPRF parameter setup used for the generated program object using the DSPPGM CL command, as shown in Figure 3-1.

Display Program Information
Program . . . . . . . : ITERATOR2   Library . . . . . . . : SAMPLEDB02
Owner . . . . . . . . : DLEMA
Program attribute . . : CLE
Detail . . . . . . . . : *BASIC

Program creation information:
  Program creation date/time . . . . . . . . . : 10/03/01 15:22:49
  Type of program . . . . . . . . . . . . . . : ILE
  Program entry procedure module . . . . . . : ITERATOR2
  Library . . . . . . . . . . . . . . . . . . : SAMPLEDB02
  Activation group attribute . . . . . . . . : *CALLER
  Shared activation group . . . . . . . . . . : *NO
  User profile . . . . . . . . . . . . . . . : *OWNER
  Use adopted authority . . . . . . . . . . : *YES
  Coded character set identifier . . . . . : 65535
  Number of modules . . . . . . . . . . . . : 1

Press Enter to continue.

F3=Exit  F12=Cancel
(C) COPYRIGHT IBM CORP. 1980, 2000.

Figure 3-1  DSPPGM for SAMPLEDB02/ITERATOR2

Every SQL stored procedure will produce a program object with USEADPAUT set to *YES. You can use the CHGPGM CL command to change the User profile (USRPRF) attribute or the Use adopted authority (USEADPAUT) attribute.
3.9.1 Authorities and adopted authorities in dynamic SQL

Authorities in dynamic SQL statements can be affected by the DYNUSRPRF parameter on the CRTSQLxxx or RUNSQLSTM CL command or in the SET OPTION clause. It is comparable to USRPRF, but it only affects dynamic SQL operations. When USRPRF is set to ‘*USER, DYNUSRPRF does not have any effect. Table 3-1 summarizes the effect of this parameter.

Table 3-1  DYNUSRPRF parameter

<table>
<thead>
<tr>
<th>USRPRF</th>
<th>DYNUSRPRF</th>
<th>Affect</th>
</tr>
</thead>
<tbody>
<tr>
<td>*USER</td>
<td>*USER or *OWNER</td>
<td>Dynamic and static SQL statements perform with the user profile authorities.</td>
</tr>
<tr>
<td>*OWNER</td>
<td>*USER</td>
<td>Dynamic SQL statements perform with the user profile authorities. Static SQL statements perform with both user and owner profile authorities.</td>
</tr>
<tr>
<td>*OWNER</td>
<td>*OWNER</td>
<td>Dynamic and static SQL statements perform with both user and owner profile authorities.</td>
</tr>
</tbody>
</table>

3.10 Testing and debugging

When you are developing any kind of software, it is important to have a debugging tool. Debugging allows you to detect, diagnose, and eliminate run-time errors in a program. This section shows you debugging alternatives to test SQL PSMs.

Remember that when you create an SQL PSM, it is really creating an ILE C program underneath. For this reason, an alternative for debugging an SP PSM is to use the ILE source debugger for testing. In V5R2, DB2 Universal Database for iSeries simplified the debug of SQL stored procedure, functions, and triggers with the SQL *SOURCE debug view and by use of the Toolbox for Java iSeries System Debugger. For more information, refer to the white paper *Graphical Debugger makes Procedural SQL Debug Even Easier – V5R3 Update*, written by Kent Milligan, on the Web at:

http://www.ibm.com/servers/eserver/iseries/db2/db2awp_m.htm

The following section refers to content in this paper.

3.10.1 Graphical debugger

DB2 Universal Database simplified the debug of SQL procedures, functions, and triggers in V5R2 with the SQL *SOURCE Debug View, but debug of SQL procedures and triggers can be even more simple by utilizing another V5R2 feature, the IBM Toolbox for Java iSeries System Debugger. This section describes these two new V5R2 enhancements and how they can be used together. In addition, PTFs have been made available to leverage these enhancements on V5R1. You can find these PTFs on the Web at:

http://ibm.com/eserver/iseries/db2/sourcedebug.html
SQL *SOURCE Debug View

SQL procedures have used C code underneath the covers since their arrival. The usage of C code behind the scenes is really not an issue for RPG and COBOL programmers until they need to debug the SQL procedure, function, or trigger. Instead of showing the original SQL procedural statements, the iSeries debugger (STRDBG - Start Debug) shows the generated C code that is being used to implement the original SQL procedural statements. Figure 3-2 shows the source of an SQL procedure statement. Figure 3-2 shows the C-based debug view that programmers were forced to use prior to V5R2.

With the arrival of the *SOURCE debug view in V5R2, programmers now have a way to work with the original SQL source in debug mode instead of the complex C code generated by DB2 Universal Database. The *SQL Source debug view can be created in one of two ways. One way is to specify the *SOURCE debug option in the source of the SQL procedure, function, or trigger. This method is demonstrated by specifying DBGVIEW=*SOURCE on the SET OPTION clause (see Example 3-18). An alternative method is to specify a *SOURCE value for the DBGVIEW parameter on the RUNSQLSTM (Run SQL Statements) CL command.
Example 3-18  DBGVIEW = "SOURCE"

CREATE PROCEDURE myschema.ship_it(IN ordnum INTEGER, IN ordtype CHAR(1),
        IN ordweight DEC(3,2))
LANGUAGE SQL
SET OPTION DBGVIEW = "SOURCE"
sp: BEGIN
DECLARE ratecalc DECIMAL(5,2);
/* Check for international order */
IF ordtype='I' THEN
    SET ratecalc = ordweight * 5.50;
    INSERT INTO wwshipments VALUES(ordnum,ordweight,ratecalc);
ELSE
    SET ratecalc = ordweight * 1.75;
    INSERT INTO shipments values(ordnum,ordweight,ratecalc);
END IF;
END

With the *SOURCE debug view specified, DB2 Universal Database creates an extra SQL
source-level debug view as it generates the C program. If the procedure name on the
CREATE PROCEDURE statement is qualified with a schema or library name, then the debug
view is created in the same library as the C program object. For Example 3-18, both the
debug view and C program for the procedure are created in the myschema library. If the
procedure name is unqualified, the SQL source-level debug view is stored in the QTEMP
library and only the job that created the SQL procedure, function, or trigger will be able to use
the *SOURCE debug view.

There are several other nuances with the SQL source-level debug that one needs to be aware
of. One is that comments are not saved in the *SOURCE debug view. Another is that when
stepping through the *SOURCE debug view in debug mode, the step may stay on the same
SQL statement for several steps, if that SQL statement required multiple lines of generated C
code to implement. The final nuance is accessing the value of SQL variables and parameters
during the debug session. The EVAL command is still used, but the variable and parameter
name must be prefixed with a label and be entered in capital letters. For parameters, the label
name is the procedure name; EVAL SHIP_IT.ORDNUM will display the value of the first input
parameter of the procedure (see Example 3-18). The variables use the label specified on the
BEGIN statement, so EVAL SP.RATECALC would be the command to execute in the debug
session to display the variable value in Example 3-18. If the variable contains a character
string, then the variable name must be prefixed with an asterisk (*), for example EVAL
*PROC.CHARVAR.

Graphical iSeries System Debugger
The new graphical iSeries System Debugger, that is part of the IBM Toolbox for Java, lets you
debug programs that run on an iSeries server. This state-of-the-art debugger includes an
integrated call stack window, breakpoint groups, variable monitors, and a local variables
display. In addition to the new capabilities, the graphical debugger can be used to debug
practically any scenario that a developer can debug using the green-screen interface
(STRDBG) without retraining.

The graphical debugger supports the ILE languages listed below. Since SQL procedures,
functions, and triggers are implemented by DB2 Universal Database as generated ILE C
code, they are also supported by the graphical debugger, including support for the *SOURCE
SQL debug view.
The following ILE languages are supported:

- ILE C & C++
- ILE RPG
- ILE COBOL
- ILE CL
- Java

Note: Original Program Model (OPM) RPG and COBOL programs can also be used with the graphical debugger if *LSTDBG or *SRCDBG are specified on the compile and the debug session is started with OPMSRC(*YES) specified.

The graphical debugger is comprised of four components:

- Client-based Debug Manager
- Client-based Debug Interface
- Host-based Debug Hub
- Host-based Debug Server

The Debug Manager is used to register a user with the Debug Hub for graphical debug on a particular iSeries server. It functions as a convenient launch point to start debug sessions. There is only one Debug Manager for a given client.

The Debug Interface also runs on the client and provides the actual debug interface. This is the interface used to set breakpoints, step through programs, inspect variables, and so on. There is one Debug Interface instance for every job that is being debugged on an iSeries server. The Debug Interface can debug programs that are running in existing jobs on the system or use the System Debugger to launch and then debug programs in a system batch job. You can set up the System Debugger to start automatically, manually from a workstation command prompt, or by using the Debug Manager interface.

As mentioned earlier, the Debug Hub (QTESDBGHUB) is the server-side component responsible for keeping track of which user's are currently registered for debug and starting debug server jobs on their behalf. Using the Start Debug (STRDBG) CL command from an emulation session contacts the Debug Hub to see if the user executing the command is registered with the Debug Manager. It also checks to see if the command being executed is from the same TCP/IP address as the Debug Manager. When these qualifications are met, the graphical iSeries System Debug application is started instead of the traditional debug environment.

The Debug Server is a TCP/IP server job that is started by the Debug Hub when a request to start debugging has been issued. The server job then services the job that is being debugged and issues the appropriate debugger APIs and commands. If the user is registered through graphical debug through the debug manager and enters the STRDBG command, the debug server runs in the same job that the STRDBG command was issued in. Otherwise, the debug server runs its own job and then uses the STRSRVJOB command to service another job to be debugged.

Debug interface components
Figure 3-4 shows the debug interface window with a series of tabbed areas. Each tab provides a different view of the overall debug environment. The debugger interface is a multiple-document window, so more than one source file can be viewed at once. Most compilers are capable of presenting several source code views. Each debug view (that is, compiler listing, statement listing, and so on) can be used during the debug session.
The Programs tab lists the current program or Java class files that are currently under debug. The program objects and breakpoint information can be saved to an environment file. To eliminate repetitive setup of breakpoints for a program, the saved environment file can be loaded into a future debug session. The Breakpoints tab enables the manipulation and listing of all program breakpoints. Conditions and thread-specific information can be specified when defining a breakpoint. In addition, breakpoints can be logically grouped together, and these groups can be assigned a name and color. The breakpoints in a groups can all be disabled and re-enabled with a single mouse click.

The Monitors tab keeps track of variables or expressions that are not found in the local program scope. A monitor can be established by just highlighting a variable or expression in a source code window. The value of each monitor is updated at each breakpoint or debug step. The value of each monitor is updated at each breakpoint or debug step.

The Locals tab (see Figure 3-5) displays the local variables associated with the program currently being debugged. Structured variables have their children organized in a tree hierarchy. For SQL procedures, functions, and triggers; the SQL variables and parameters are organized under each label. Like monitored variables, locals can be edited directly from this panel.
The Console tab (see Figure 3-6) allows users to directly enter the same debug commands (that is, EVAL) that they use on the green-screen debugger interface. The graphical debugger supports all of the green-screen debug commands.

The Call Stack tab displays the call stack associated with the program in debug mode. Any call frame in the call stack can be visited using the mouse or a pull-down menu.

The Thread tab displays all active threads in the program. If the code being executed by the thread is in debug mode, the program information including line numbers are displayed. Any source code being executed in a thread can be accessed by a mouse click or a pull-down menu.
Debugger installation and configuration

To run the graphical iSeries system debugger, your client system must meet the following hardware and software requirements:

- **Hardware:**
  - CPU: 400 - 500 MHZ
  - Memory: 128 MB (minimum) - 256 MB (recommended)

- **Software (one of the following):**
  - Java 2 Platform, either the Standard Edition (J2SE™) or the Enterprise Edition (J2EE™), Version 1.3 or later
  - Java 2 Runtime environment (JRE™), Standard Edition, Version 1.3.1 or later

**Note:** You can download the software from the Sun Microsystem’s Java Web site at: http://java.sun.com/downloads/index.html

To install the graphical iSeries system debugger:

1. Load V5R2 SI05473 onto your iSeries server. On a V5R1 iSeries server, load the following PTFs:
   - SI05799
   - SI02871
   - SI02849

2. The IBM Toolbox for Java is required by the graphical debugger, so the IBM Toolbox for Java must be on the client. IBM Toolbox for Java is a component of iSeries Navigator. Instead of installing iSeries Navigator on the client, copy the jt400.jar file from the /QIBM/ProdData/HTTP/Public/jt400/lib/ directory on the iSeries server.

3. Copy the Graphical System Debugger file, tes.jar, from the /QIBM/ProdData/HTTP/Public/jt400/lib/ directory on the iSeries server.

4. Modify the CLASSPATH variables to include access to the jt400.jar, tes.jar, and jhall.jar files on your client. A CLASSPATH setting for a typical client is C:\Program Files\IBM\Client Access\jt400\lib\jt400.jar, C:\Program Files\IBM\Client Access\jt400\lib\tes.jar, and C:\Program Files\IBM\Client Access\JRE\Lib.

5. Start the host debug server on your iSeries server:

   STRTCPVR *DBG

   On V5R1, the debug server is started by issuing the following program call:

   CALL QSYS/QTESSTRSVR

To get the graphical debugger up and running for an SQL stored procedure:

1. Create the SQL procedure with the *SOURCE debug view. In this case, the source code for the SQL procedure in Example 3-18 on page 40 has been copied into a source physical file member called MYLIB/MYSRC(SPTEST). The following RUNSQLSTM command is executed to create the SQL procedure with the debuggable source-level debug:

   RUNSQLSTM SRCFILE(MYLIB/MYSRC) SRCMBR(SPTEST) COMMIT(*NONE) NAMING(*SQL)

2. Start the Debug Manager on the client by issuing the following command from an MS-DOS® prompt:

   Java utilities.DebugMgr

   This command launches the graphical debug manager.
3. From the graphical debug manager, click the **Edit** to register your ID and iSeries server for a graphical debug session. After registration is completed, your DebugMgr view should look like the example in Figure 3-7.

![DebugMgr view](image)

**Figure 3-7  DebugMgr view**

4. Return to your iSeries session and start debug mode for the newly created SQL procedure with the following command:

   STRDBG PGM(SHIP_IT) UPDPROD(*YES)

   This command causes the iSeries Graphical Debugger to be started on your client and load the SQL source-level debug view for the SHIP_IT stored procedure.

5. Click Line 5 (IF ORDTYPE='I') to set a breakpoint. An enabled breakpoint is indicated by the red arrow demonstrated in Figure 3-4 on page 42. Now that a breakpoint has been set, click the green resume arrow on the tool bar.

6. Return to your iSeries session and issue the following SQL CALL statement using either RUNSQLSTM command or interactive SQL:

   CALL SHIP_IT(33, 'I', 5.1)

   The Debug Client then takes control at the breakpoint specified in the previous step. Figure 3-5 on page 43 shows how the yellow highlighting is used to indicate where execution was stopped for the breakpoint.

7. To view the contents of the ORDTYPE input parameter to determine which leg of the IF statement will be executed, click the **Console** tab in the lower left corner. Enter the following EVAL statement in the Command window:

   EVAL *SHIP_IT.ORDTYPE

   The contents of this variable are displayed in the Console window, as demonstrated in Figure 3-6 on page 43. You can display all of the procedure parameter values by entering EVAL SHIP_IT. When using the graphical debugger over the normal debug views, variable values are automatically displayed when just moving the cursor over a variable name in a line of code. This flyover display of variables is not available with the *SOURCE debug view for SQL procedures, functions, and triggers.
8. To view the calculated shipping rate prior to the stored procedure ending, right-click **Line 11** and select the **Run to Cursor** task. This allows the debugger to execute all of the code up to Line 11. When Line 11 is reached, enter the following command in the Console window to display the computed shipping rate:

```
EVAL  SP:RATECALC
```

9. To complete execution of the **SHIP_IT** stored procedure, click the green **resume arrow** on the tool bar.

### 3.10.2 The ILE source debugger

The other tool available used to detect errors in, and eliminate errors from, program objects and service programs is the ILE source debugger. By using debug commands with any ILE program, you can:

- View the program source or change the debug view.
- Set and remove conditional and unconditional breakpoints.
- Step through a specified number of statements.
- Display or change the value of fields, structures, and arrays.
- Equate a shorthand name with a field, expression, or debug command.

Many debug commands are available for use with the ILE source debugger. These debug commands and their parameters are entered on the debug command line shown at the bottom of the Display Module Source and Evaluate Expression® displays. These commands can be entered in uppercase, lowercase, or mixed case.

**Note:** The debug commands on the debug command line are not CL commands.

The debug commands are:

- **ATTR:** Permits you to display the attributes of a variable. The attributes are the size and type of the variable.
- **BREAK:** Permits you to enter an unconditional or conditional breakpoint at a position in the program being tested. Use the **BREAK line-number WHEN** expression to enter a conditional breakpoint.
- **CLEAR:** Permits you to remove conditional and unconditional breakpoints.
- **DISPLAY:** Allows you to display the names and definitions assigned by using the **EQUATE** command.
- **EQUATE:** Allows you to assign an expression, variable, or debug command to a name for shorthand use.
- **EVAL:** Allows you to display or change the value of a variable or to display the value of expressions, records, structures, or arrays.
- **QUAL:** Allows you to define the scope of variables that are displayed in subsequent **EVAL** commands.
- **STEP:** Allows you to run one or more statements of the procedure being debugged.
- **FIND:** Searches forward or backward in the module currently displayed for a specified line number or string or text.
- **UP:** Moves the displayed window of source toward the beginning of the view for the number of lines entered.
- **DOWN:** Moves the displayed window of source toward the end of the view for the number of lines entered.
3.10.3 Preparing the SQL procedure for debugging

A program or module must have debug data available if you are to debug it. Debug data is created during compilation. You have to specify this to the iSeries server using one of the following approaches:

- Starting with V5R1, you can specify parameters in the CREATE PROCEDURE, CREATE TRIGGER, or CREATE FUNCTION statement using the SET OPTION as in this example:

```sql
CREATE PROCEDURE G8()
LANGUAGE SQL
SET OPTION OUTPUT = *PRINT, DBGVIEW = *STMT
BEGIN
...
END
```

In the previous example, we specify that a listing must be generated into a spool file and the generated C module is to contain debug data.

- In the RUNSQLSTM command, specify whether the generated C module contains debug data. This is done by using the DBGVIEW parameter on the RUNSQLSTM command as shown in Figure 3-8. You can also specify if a listing output is desired by using the OUTPUT parameter.
Run SQL Statements (RUNSQLSTM)

Type choices, press Enter.

Source file . . . . . . . . . . > QSQLSRC Name
Library . . . . . . . . . . . > ORDAPILIB Name, *LIBL, *CURLIB
Source member . . . . . . . > CASEPROC Name
Commitment control . . . . > *NONE *CHG, *ALL, *CS, *NONE.
Naming . . . . . . . . . . . > *SQL *SYS, *SQL

Additional Parameters

Severity level . . . . . . . 10 0-40
Date format . . . . . . . . . *JOB *JOB, *USA, *ISO, *EUR.
Date separator character . . *JOB *JOB, /,, ,", ",
Time format . . . . . . . . . *HMS *HMS, *USA, *ISO, *EUR,
Default Collection . . . . . *NONE Name, NONE
IBM SQL Flagging . . . . . . *NOFLAG *NOFLAG, *FLAG
ANS flagging . . . . . . . . *NONE *NONE, *ANS

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this disp
F24=More keys

Figure 3-8  RUNSQLSTM command

Press the Page Down key to view the DBGVIEW parameter, as shown in Figure 3-9.

Run SQL Statements (RUNSQLSTM)

Type choices, press Enter.

Sort Sequence . . . . . . . *JOB Name, *HEX, *JOB...
Library . . . . . . . . . . . Name, *LIBL, *CURLIB
Language id . . . . . . . . . *JOB *JOB, *JOBRUN...
Print file . . . . . . . . . . QSYSPRT Name
Library . . . . . . . . . . . *LIBL Name, *LIBL, *CURLIB
Statement processing . . . . *RUN *RUN, *SYN
Allow copy of data . . . . . *OPTIMIZE *YES, *NO, *OPTIMIZE
Close SQL cursor . . . . . . *ENDACTGRP *ENDMOD, *ENDACTGRP
Allow blocking . . . . . . . *READ *READ, *NONE, *ALLREAD
Delay PREPARE . . . . . . . *NO *YES, *NO
Debugging view . . . . . . . *LIST *LIST, *NONE
User profile . . . . . . . . *NAMING *NAMING, *USER, *OWNER
Dynamic User Profile . . . *USER *USER, *OWNER
Listing output . . . . . . . *NONE *NONE, *PRINT
Target release . . . . . . . *CURRENT *CURRENT, VxRxMx

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this displ
F24=More keys

Figure 3-9  Creating an SQL procedure with additional debugging information
The Debugging view parameter specifies the type of source debug information to be provided by the SQL precompiler. The possible values are:

- **NONE**: The debug view is not generated.
- **STMT**: Allows the compiled module object to be debugged using program statement numbers and symbolic identifiers.
- **LIST**: Generates the listing view for debugging the compiled module object.

You must specify **STMT** or **LIST** if you want debugging data to be saved in the program. After RUNSQLSTM successfully creates the SQL PSM, you are ready to test it.

### 3.10.4 Testing the SQL PSM

When testing the SQL PSM, if it updates, inserts, or deletes records from files, you can use an interactive SQL session to verify that it works properly. Another useful tool is to use query for verifying the results.

There are cases in which you use local variables:

- Passing information to other programs
- Calculating totals or arithmetic operations
- Checking conditions
- Assignments

SQL control statements do not include a PRINT or DISPLAY statement, and that is the main reason that some programmers insert variable values into debug tables that can be reviewed after the procedure is executed. Another way to determine the content of a variable in a procedure is by using the ILE C code debugger.

While debugging and testing your program, ensure that your library list is changed to direct the programs to a test library containing test data so that any existing real data is not affected.

To start a debugging session, use the STRDBG command as shown in Figure 3-10.

![Start Debug session](image)
It is important to note that when your session is in debug mode, the job log of the session saves a lot of information related to the SQL statements being executed. The application developer can use this information for problem detection and performance tuning.

After starting the debug, you are ready to call the procedure by issuing the SQL CALL statement as shown in Figure 3-11.

```
>>> call ordapplib.caseproc ('00002', 0)
```

After the SQL procedure is called, the ILE Source debugger is invoked, as shown in Figure 3-12.

```
void main(int argc, char* argv[]) {
  SQLP_IND = (short int*) argv[3];
  sqlcap = (SQLCA*) argv[4];
  SQLInitSQLCA((SQLCA*)&sqlca);
  CASEPROC.SQLP_I1 = *(SQLP_IND+0);
  if (CASEPROC.SQLP_I1 != SQLP_NULLIND)
    strcpy(CASEPROC.CUSNBR,argv[1]);
  CASEPROC.SQLP_I2 = *(SQLP_IND+1);
  if (CASEPROC.SQLP_I2 != SQLP_NULLIND)
    CASEPROC.EVALUA = * (decimal(2,0)*) argv[2];
  if (CASEPROC.SQLP_I2 != SQLP_NULLIND & &
    CASEPROC.EVALUA>90)
    CASEPROC.EVALUA=90;
}
```

On the debugging line, you can enter any of the debug commands. This way, you can display the content of any variable. You can also go through the program step-by-step using the F10 key.
Since your session is in debug mode, the job log has all the messages related to the execution of the procedure. If you type the DSPJOBLOG command, you see the information shown in Figure 3-13.

![Figure 3-13 Job log in debug mode](image)

From the job log, you can see that 17 records from the CUSTOMER file were updated. You can also see that there seems to be a data conversion. If you place the cursor on that line and ask for help, you see the information that is displayed in Figure 3-14.

![Figure 3-14 Additional information in the job log](image)
We encourage you to always test your SQL procedures in debug mode so you can see what the database manager is doing underneath. This allows you to solve some of the problems and deliver better quality software. Figure 3-13 shows that there is a data conversion that eventually affects the performance of the procedure.

**Note:** If you do not feel comfortable debugging a C program, you can always create a temporary table where you can write the content of the local variables for each iteration. Then with a simple SELECT statement, you can see the values of the different fields.

### 3.10.5 Testing the SQL PSM in a distributed environment

Testing and debugging SQL PSMs in a distributed environment may be a little bit more tricky than in the traditional iSeries server environment. This section outlines the steps required to debug an SQL stored procedure called from a Visual Basic client. The combination of Visual Basic running on the client and SQL running on a powerful database server, such as the iSeries server, can result in a highly scalable and robust software solution.

In our test scenario, we code a Visual Basic client that uses ADO on top of the iSeries ODBC driver to send the SQL request to DB2 Universal Database for iSeries. In the iSeries server distributed architecture, an ODBC client communicates with a corresponding iSeries server job, which runs the SQL requests on behalf of this client. When we call a stored procedure from the Visual Basic client, an iSeries server job invokes the stored procedure on the server and then passes back the results to the client. The iSeries server jobs associated with the database access are named QZDASOINIT and run in the QSERVER subsystem. At any given time, a large number of database server jobs may be active in the QSERVER subsystem. The first step in the debug procedure is to find the server job that serves the client:

1. Open the Visual Basic project and set a break point at the line just below the invocation of the connection open method, as shown in Figure 3-15.

![Figure 3-15 Adding a break point in a Visual Basic client](image-url)
2. Switch to the iSeries session. To find the QZDASOINIT job serving your client program, run the following CL command:

```
WRKOBJLCK OBJ(XXXXXXXX) OBJTYPE(*USRPRF)
```

Here, XXXXXXXX is the user profile you use to log into the iSeries server.

The Work with Object Locks screen is displayed. There should be one job named QZDASOINIT listed. Type 5 in the Option field next to this job as shown in Figure 3-16, and press Enter.

![Work with Object Locks](image)

On the Work with Job display, type option 10 (Display job log, if active or on job queue).

In the Display Job Log screen, find the first message in the job log and write down the fully qualified job name for your database server job as shown in Figure 3-17.

![Display Job Log](image)

In this case, the fully qualified name is 064728/QUSER/QZDASOINIT.
3. Return to the command prompt, and run the following CL command:

```
STRSRVJOB JOB(064728/QUSER/QZDAS0INIT)
```

**Note:** The Start Service Job (STRSRVJOB) command starts the remote service operation for a specified job so that other service commands can be entered to service the specified job. Any dump, debug, and trace commands can be run in that job until the service operation ends.

4. Start the ILE C Source Debugger for your server job with the following CL command:

```
STRDBG PGM(LEUNGRHY/GETCUSNAME) UPDPROD(*YES)
```

The ILE source debugger loads the ILE C source created for your SQL stored procedure. Set the breakpoint, and return to the command line.

5. Return to the Visual Basic client session. Set the breakpoint at the statement that calls the stored procedure on the iSeries server as shown in Figure 3-18.

![Figure 3-18 Calling the stored procedure from Java](image)

Run the statement at the breakpoint. The client code execution is suspended since the control was passed to the stored procedure on the iSeries server.

6. Switch to the iSeries server session. The ILE C Source Debugger was activated, so you may step through your stored procedure on the server. Run the procedure to completion. The control returns to the client, and you may continue to work with the Visual Basic code.

### 3.11 Reverse engineering and Generate SQL

Reverse engineering is one of the major changes that have been included in V5R1M0. This function allows you to create SQL for a given schema, table, stored procedure, function index, view, and so on, and all related objects to them if that option is selected. This enables database administrators to recreate, create duplicates, and port to other iSeries servers entire databases or particular parts of a database.
The Generate SQL function is often referred to as “reverse engineering for iSeries Navigator” because it provides a GUI interface that allows you to reverse engineer several types of database objects. The results are SQL create statements (often referred to as Data Definition Language (DDL) statements).

The Generate SQL function of iSeries Navigator allows you to reconstruct SQL statements used to create existing database objects. With this function, you can reverse engineer database objects and then have the option to display the resulting SQL in the Run SQL Scripts window or save the output to a file. Using the existing Run SQL Scripts functions, you can then edit, run, and save the SQL statement to a file on the PC.

The new Generate SQL Database Objects support the following objects:
- Aliases
- Distinct types
- Functions
- Indexes
- Procedures
- Schemas (collections) and libraries
- Tables and physical files
- Views and logical files

### 3.11.1 Generate SQL

Reverse engineering (Generate SQL) allows you, through the Database Navigator map and the Libraries display of iSeries Navigator, to re-engineer an SQL database or an iSeries database that was not created using SQL.

One of the uses of Generate SQL is to generate the SQL statements of tables, procedures, functions, views, indexes, and constraints that were created using the iSeries Navigator interface. For example, when you create a stored procedure using iSeries Navigator, there is no method for saving the SQL statement that is behind the interface. In this case, Generate SQL provides a way to reverse engineer this object and obtain the SQL statement.

The Generate SQL function of Database Navigator also creates the SQL statements of databases created by DDS (physical and logical files). You must be aware that keyed-logical files are converted to SQL views.

When the Generate SQL process creates the Run SQL script for the selected object, it either marks any problem objects with SQL messages or it does not create the SQL for the object if it is not supported. You can create a Run SQL Script from object context or from schema context.

With Generate SQL, there is an option from your library in the Operations Navigator window to generate the SQL DDL statement for some objects. To generate this statement:

1. Start iSeries Navigator. Click the iSeries server that you want to access. After you enter your user ID and password, expand the Databases option.
2. Under Databases, double-click your database and then click Libraries. Then select the library name, which in our case is SAMPLEDB04, for your iSeries server connection.
3. Click the SAMPLEDB04 library to display the current content in the right window panel. In the right panel, press the Ctrl key, and locate and select the objects that you want to generate the SQL from.
**Important:** When the Generate SQL function is invoked, the new Generate SQL window opens as shown in Figure 3-19. This window provides a list of the objects initially selected and three tabs that specify Output, Format, and Options that are used in the Generate SQL.

![Generate SQL window](image)

*Figure 3-19  Generate SQL window*
4. Click **Generate** to accept the default values as shown in Figure 3-20.

**Important:** The initial list of objects in the Generate SQL window can be modified using the Add and Remove buttons to add new objects or remove objects from the initial list.
5. Switch to the new Run SQL Scripts window to see the generated SQL statement.

Figure 3-21  SQL generated in the Run SQL Scripts window
6. Click **File -> Save As** to save the SQL script as shown in Figure 3-22.

![Figure 3-22   Saving the SQL script](image)

**Important:** You can use the SQL file to replicate your database files on another system (for example, a development system).

7. Click **Save** to save the SQL script file.

This is a useful tool to recreate the SQL statements for database objects. For a complete description of this tool, refer to *Advanced Functions and Administration on DB2 Universal Database for iSeries*, SG24-4249.
This part describes stored procedures and its benefits. It also examines the different types of stored procedures that are available in DB2 Universal Database for iSeries. This part dedicates separate chapters to SQL stored procedures, external stored procedures, and Java stored procedures. Another chapter describes error handling in stored procedures, as well as examples on the client code.
Stored procedures

This chapter explains how you can take advantage of stored procedures in developing a distributed application. Stored procedures provide a standard way to call an external procedure from within an application by using an SQL statement.

This chapter describes:

- The concepts and benefits of stored procedures
- The types of stored procedures
- The CREATE PROCEDURE command
- System catalog tables
- Deleting or replacing a stored procedure
- Procedure overloading
4.1 Introduction

The *invocation* of a stored procedure is treated as a regular external call. The application waits for the stored procedure to terminate, and parameters can be passed back and forth. Stored procedures can be called locally (on the same system where the application runs) and remotely on a different system. However, stored procedures are particularly useful in a distributed environment since they may considerably improve the performance of distributed applications by reducing the traffic of information across the communication network.

For example, if a client application needs to perform several database operations on a remote server, you can choose between issuing many different database requests from the client site and calling a stored procedure. In the first case, you start a window with the remote system every time you issue a request. If you call a stored procedure instead, only the call request and the parameters flow on the line. In addition, the server system executes some of the logic of your application with potential performance benefits at the client site.

Your programming productivity can be improved by using stored procedures when you develop distributed applications. Stored procedures are the easiest way to perform a remote call and to distribute the execution logic of an application program.

Stored procedures can be used for many different application purposes such as:
- Distributing the logic between a client and a server
- Performing a sequence of operations at a remote site
- Combining results of query functions at a remote site
- Controlling access to database objects
- Performing non-database functions

Let us look at a typical example where stored procedures can be effective. A company runs its business on a server located at the headquarters and on client systems located at every branch office. A user at a branch office is working with an invoice clearance application, which has to update three tables on the server:
- The invoice table is named INVOICE.
- The customer table is named CUSTOMER.
- The account receivable balance table is named ARBLNCE.

An invoice record is flagged with a “clearance” marker, and after that the corresponding CUSTOMER record is updated by deducting the invoice amount from the current account receivable total amount. Finally, the account receivable balance record must be updated as well.

Figure 4-1 shows a distributed application for the invoice clearance process that was implemented without resorting to stored procedures. The client system has to access the server database several times for every update event, sending and receiving data across communication lines for every request. In addition, all the application logic is implemented at the client site.
Figure 4-1  Distributed application without stored procedures

Figure 4-2 shows how we can take advantage of stored procedures in developing this application.

Figure 4-2   Distributed application with stored procedures

The same application functions can be carried out by calling a single stored procedure that runs at the server site. The communications window is greatly reduced, and the network resources are better balanced by splitting the application logic.
Modularity in application development is also encouraged by using stored procedures. This makes application maintenance easier and improves code reusability.

It is useful to compare stored procedures to other tools and techniques for distributed application development such as DRDA SQL, DDM Submit Remote Command (SBMRMTCMD), and triggers:

- With DRDA SQL, the application logic is fully implemented at the application requester site. Stored procedures are the natural extension for DRDA applications, since they allow you to easily split the application logic.

- The Submit Remote Command (SBMRMTCMD) command submits a CL command using Distributed Data Management (DDM) support to run on the target system. The SBMRMTCMD command allows a user at a client system to perform some object management operations rather than running remote applications. You may also want to submit user-written commands or programs to run on the target system, but you face the following restrictions:
  - The target (server) system cannot send any parameters to the source (client) system. Only a generic return code is sent back to signal whether the remote execution completed successfully.
  - Any changes in database tables made by the server application on the server system cannot be committed or rolled back by the client application.

- Triggers are user-written programs associated with a table. Unlike stored procedures, they are almost independent from applications because they are automatically executed either before or after a database change. Stored procedures need to be called explicitly by the SQL CALL statement.

  Triggers receive from the database manager a standard parameter list, which is input only, and they cannot pass any information back to the application through the parameter list. Therefore, when the trigger ends abnormally, the application must receive an error message or an SQLCODE and handle it. Stored procedures can receive input/output parameters and use them to communicate with the client application.

  Triggers can be used to enforce business rules. Stored procedures are used mainly to improve the performance of distributed applications and productivity of application development. Stored procedures are capable of returning result sets, which makes them very flexible and efficient in client/server environments.

  See Part 3, “Triggers” on page 279, for more information about triggers.

Table 4-1 summarizes the comparison of stored procedures, triggers, UDFs and DRDA SQL for distributed applications.

<table>
<thead>
<tr>
<th>Stored procedure</th>
<th>Trigger</th>
<th>UDFs</th>
<th>DRDA SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation</td>
<td>Execution of SQL CALL statement</td>
<td>Database I/O</td>
<td>As a function in an SQL statement</td>
</tr>
<tr>
<td>Environment</td>
<td>Distributed or non-distributed applications</td>
<td>Distributed or non-distributed applications</td>
<td>Distributed or non-distributed applications</td>
</tr>
<tr>
<td>Language</td>
<td>Any HLL program including Java (may include SQL)</td>
<td>Any HLL program (may include SQL) - No Java support</td>
<td>Any HLL program including Java (may include SQL)</td>
</tr>
</tbody>
</table>
### 4.2 Stored procedure types

There are two categories into which stored procedures can be divided:

- SQL stored procedures
- External stored procedures

#### 4.2.1 SQL stored procedures

SQL stored procedures are written in the SQL language. This makes it easier to port stored procedures from other database management systems (DBMS) to the iSeries server and from the iSeries server to other DBMS. Implementation of the SQL stored procedures is based on procedural SQL standardized in SQL99. For more details about SQL stored procedures, refer to Chapter 5, “SQL stored procedures” on page 81.

#### 4.2.2 External stored procedure

An external stored procedure is written by the user in one of the programming languages on the iSeries server. You can compile the host language programs to create *PGM objects or Service Program. To create an external stored procedure, the source code for the host language must be compiled so that a program object is created. Then the CREATE PROCEDURE statement is used to tell the system where to find the program object that implements this stored procedure. The stored procedure registered in the following example returns the name of the supplier with the highest sales in a given month and year. The procedure is implemented in ILE RPG with embedded SQL:

```sql
EXEC SQL
CREATE PROCEDURE HSALE
(IN YEAR INTEGER    ,
 IN MONTH INTEGER   ,
 OUT SUPPLIER_NAME CHAR(20) ,
 OUT HSALE DECIMAL(11,2))
EXTERNAL NAME SPROCLIB.HSALES
LANGUAGE RPGLE
PARAMETER STYLE GENERAL
END_EXEC
```

The following SQL CALL statement calls the external stored procedure, which returns a supplier name with the highest sales:

```sql
EXEC SQL
CALL HSALE(:PARM1, :PARM2, :PARM3, :PARM4)
END-EXEC
```
An external stored procedure may contain no SQL statements. For example, you may create a stored procedure that uses the native interface to access the DB2 Universal Database for iSeries data.

The iSeries server implementation of external stored procedures is discussed in detail in Chapter 6, “External stored procedures” on page 123.

**Java stored procedures**

Java stored procedures were first introduced in the iSeries server starting with V4R5, as a particular case of external stored procedures limited to not being able to return result sets. Starting with V5R1, the support of result sets was added to Java stored procedures. There is a growing interest in Java and its portability across platform that makes Java stored procedures a very interesting option to consider. The iSeries server implementation of Java stored procedures is discussed in detail in Chapter 7, “Java stored procedures” on page 173.

### 4.3 Registering stored procedures

Before a stored procedure can be called by a client program, it must be registered with the database using the DECLARE PROCEDURE or the CREATE PROCEDURE statement. The stored procedure can also be defined using either of these statements. The CREATE PROCEDURE statement differs from the DECLARE PROCEDURE since it adds procedure and parameter definitions to the system catalog tables (SYSROUTINES and SYSPARMS). This way, a stored procedure becomes available for any client program running on the local or the remote system. Since the information about the stored procedure is stored in the system catalog tables, the CREATE PROCEDURE needs to be performed only once in the lifetime of a stored procedure. Use the DROP PROCEDURE statement to delete the stored procedure catalog information entry. The DECLARE PROCEDURE statement is not frequently used. It is mainly for temporary registration of stored procedures. For a detailed discussion on the CREATE PROCEDURE, refer to *SQL Reference*, SC41-5612.

#### 4.3.1 CREATE PROCEDURE

The CREATE PROCEDURE statement can be used to create any of the two types of stored procedures. This statement can be issued interactively, or it can be embedded in an application program. After a procedure is registered, it can be called from any interface supporting the SQL CALL statement.

During stored procedure creation, you can control characteristics that affect the way the stored procedure is identified in DB2 Universal Database for iSeries or its behavior. This section explains some of them.

**SPECIFIC specific-name**

DB2 Universal Database for iSeries identifies each stored procedure with a specific name that, combined with the specific schema, must be unique in the system. This gains importance because multiple stored procedures with the same name but different signatures must have different specific names. If you do not provide a specific name, DB2 Universal Database for iSeries generates one automatically. If the SQL procedure name is longer than 10 characters, this name can be used to specify the C program name instead of having DB2 generate one automatically, as shown in the following example:

```
CREATE PROCEDURE SAMPLE.ALLOCATECOSTS(...)
...
SPECIFIC ALLOCATECOSTS_3PARMS
...
```
CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL

These options allow you to set some limits in regard to what the stored procedure can use. The different options are described in Table 4-2.

Table 4-2  SQL statements in stored procedures

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTAINS SQL</td>
<td>The stored procedure contains SQL. It can only contain:</td>
</tr>
<tr>
<td></td>
<td>▶ Non-executable statements (such as DECLARE statements)</td>
</tr>
<tr>
<td></td>
<td>▶ CALL statements to procedures with NO SQL or CONTAINS SQL attribute</td>
</tr>
<tr>
<td></td>
<td>▶ FREE LOCATOR</td>
</tr>
<tr>
<td></td>
<td>▶ SET RESULT SET</td>
</tr>
<tr>
<td></td>
<td>▶ SET assignment and VALUES INTO as long as only variables or constants</td>
</tr>
<tr>
<td></td>
<td>are referenced</td>
</tr>
<tr>
<td></td>
<td>▶ COMMIT, ROLLBACK, or SET TRANSACTION</td>
</tr>
<tr>
<td></td>
<td>▶ CONNECT, DISCONNECT, RELEASE, or SET CONNECTION</td>
</tr>
<tr>
<td>NO SQL</td>
<td>The stored procedure does not contain SQL statements.</td>
</tr>
<tr>
<td>READS SQL DATA</td>
<td>The stored procedure possibly reads data using SQL. It can contain SQL</td>
</tr>
<tr>
<td></td>
<td>statements other than:</td>
</tr>
<tr>
<td></td>
<td>▶ COMMIT, ROLLBACK, or SET TRANSACTION</td>
</tr>
<tr>
<td></td>
<td>▶ CONNECT, DISCONNECT, RELEASE, or SET CONNECTION</td>
</tr>
<tr>
<td></td>
<td>▶ DELETE, INSERT, or UPDATE</td>
</tr>
<tr>
<td></td>
<td>▶ ALTER TABLE, COMMENT ON, any CREATE, DROP, GRANT, LABEL ON, RENAME, or</td>
</tr>
<tr>
<td></td>
<td>REVOKE statement</td>
</tr>
<tr>
<td>MODIFIES SQL DATA</td>
<td>The stored procedure possibly modifies data using SQL. It can contain SQL</td>
</tr>
<tr>
<td></td>
<td>statements other than:</td>
</tr>
<tr>
<td></td>
<td>▶ COMMIT, ROLLBACK, or SET TRANSACTION</td>
</tr>
<tr>
<td></td>
<td>▶ CONNECT, DISCONNECT, RELEASE, or SET CONNECTION</td>
</tr>
</tbody>
</table>

SET OPTION

The SET OPTION clause gives you more control over the way in which SQL stored procedures are created. Some of the options that you can control with SET OPTION are:

- OUTPUT: Specifies whether a listing file is required. If set to *PRINT, it generates two listing spool files, one for the intermediate C code and another for the corresponding precompiled C code.
- DBGVIEW: Controls the level of debug information contained in the program object. Its default is none, but you can set *STMT, *LIST or *SOURCE values. *STMT allows the compiled module object to be debugged using program statement numbers and symbolic identifiers. *LIST generates the listing view for debugging the compiled module object. *SOURCE lets you debug the generated program or module at the SQL statement level.
- TGTRLS: Defines the target release of the operating system in which you intend to use the stored procedure.
- DATFMT: Specifies the format for dates.
- DATSEP: Specifies the date separator.
- TIMFMT: Specifies the format for times.
- TIMSEP: Specifies the time separator.
- **DECMPT**: Specifies the decimal point value (can be *POINT, *COMMA, *SYSVAL, or *JOB).
- **SRTSEC**: Specifies the sort sequence table to be used for string comparisons in SQL statements.
- **LANGID**: Specifies the language identifier to be used when SRTSEQ(*LANGIDUNQ) or SRTSEQ(*LANGIDSHR) is specified.
- **ALWCPYDTA**: Specifies whether a copy of the data can be used in a SELECT statement. Possible values are *YES, *NO and *OPTIMIZE (default). It influences the optimizer access plan for select statements.
- **ALWBLK**: Specifies whether the database manager can use record blocking and the extent to which blocking can be used for read-only cursors.
- **DLYPRP**: Specifies whether the dynamic statement validation for a PREPARE statement is delayed until an OPEN, EXECUTE, or DESCRIBE statement is run. Delaying validation improves performance by eliminating redundant validation.
- **USRPRF**: Specifies the user profile that is used when the compiled program object and SQL package object are run, including the authority that the program object or SQL package has for each object in static SQL statements. The profile of either the owner or the user is used to control access to objects.
- **DYNUSRPRF**: Specifies the user profile that is used for dynamic SQL statements.

The following example shows using some of the options in a stored procedure creation:

```sql
CREATE PROCEDURE ITERATOR2()
LANGUAGE SQL
SET OPTION TGTRLS = V4R5M0, OUTPUT = *PRINT, SRTSEQ=*LANGIDUNQ, LANGID=ESP
BEGIN
    ins_loop:
    FOR each_department AS
c1 CURSOR FOR
        SELECT deptno, deptname, admrdept
        FROM sampledb02.department
        WHERE deptno <> 'D11'
        ORDER BY deptno
    DO
        INSERT INTO sampledb02.deptnew (deptno, deptname, admrdept)
        VALUES (deptno, deptname, admrdept);
    END FOR;
END;
```

You will find that OUTPUT and DBGVIEW are of great value when debugging an application. For more information about these values, you can go to the online help for the Run SQL Statement (RUNSQLSTM) command.

### 4.3.2 DECLARE PROCEDURE

DECLARE PROCEDURE is a kind of temporal procedure definition in which the stored procedure is declared in a program. Originally it could only be embedded in an application program as a static SQL statement. However starting in V5R1 (and we advise that you to look for the latest iSeries Access ODBC driver available), it can be used as a dynamic SQL statement in an ODBC program.

With the DECLARE PROCEDURE, you avoid catalog lookup for procedure information, which in some cases, represents a performance improvement.
There are some differences in the syntax of DECLARE PROCEDURE compared with CREATE PROCEDURE, as you can see in Example 4-1.

**Example 4-1 DECLARE PROCEDURE**

```sql
EXEC SQL
DECLARE GETSUPPLIERSDB2GENERAL PROCEDURE (  
    IN YEAR INTEGER,
    IN MONTH INTEGER,
    IN RANK INTEGER)
PARAMETER STYLE DB2GENERAL
RESULT SETS 2
LANGUAGE JAVA
EXTERNAL NAME 'GetSupplierResultSetDB2GENERAL!GetSupplierRS';
```

Notice the difference in the order of the tokens (line highlighted in bold) compared to the equivalent CREATE PROCEDURE GETSUPPLIERSDB2GENERAL.

The C++ program sample in Example 4-2 shows how to use DECLARE PROCEDURE from an ODBC program.

**Example 4-2 DECLARE PROCEDURE from an ODBC program**

```cpp
#include <windows.h>
#include <sqlext.h>
#include <stdio.h>
#include <iostream.h>

// Define The DeclarExample Class
class DeclarExample {
  // Attributes
  public:
    SQLHANDLE  EnvHandle;
    SQLHANDLE  ConHandle;
    SQLHANDLE  DclStmtHandle;
    SQLHANDLE  SpStmtHandle;
    SQLRETURN  rc;
  // Operations
  public:
    DeclarExample();                        // Constructor
    ~DeclarExample();                       // Destructor
    SQLRETURN declareSP();
    SQLRETURN executeSP();
    SQLRETURN printError( SQLHDBC, SQLHSTMT);
};

// Define The Class Constructor
DeclarExample::DeclarExample() {
  // Initialize The Return Code Variable
  rc = SQL_SUCCESS;
  // Allocate An Environment Handle
  rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &EnvHandle);
  // Set The ODBC Application Version To 3.x
  if (rc == SQL_SUCCESS)
    rc = SQLSetEnvAttr(EnvHandle, SQL_ATTR_ODBC_VERSION,  
                       (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
  // Allocate A Connection Handle
  if (rc == SQL_SUCCESS)
    rc = SQLAllocHandle(SQL_HANDLE_DBC, EnvHandle, &ConHandle);
```
// Define The Class Destructor
DeclarExample::~DeclarExample()
{
    // Free SQL Statements Handle
    if (SpStmtHandle != NULL)
        SQLFreeHandle(SQL_HANDLE_STMT, SpStmtHandle);
    if (DclStmtHandle != NULL)
        SQLFreeHandle(SQL_HANDLE_STMT, DclStmtHandle);
    // Free The Connection Handle
    if (ConHandle != NULL)
        SQLFreeHandle(SQL_HANDLE_DBC, ConHandle);
    // Free The Environment Handle
    if (EnvHandle != NULL)
        SQLFreeHandle(SQL_HANDLE_ENV, EnvHandle);
}

SQLRETURN DeclarExample::declareSP()
{
    // Declare The Local Memory Variables
    SQLRETURN rc;
    SQLCHAR SQLStmt[512];

    // Declare the procedure to avoid catalog lookups
    strcpy((char *)SQLStmt, "DECLARE dlema.dosomething PROCEEDURE ();");
    strcat((char *)SQLStmt, "PARAMETER STYLE DB2GENERAL LANGUAGE JAVA ");
    strcat((char *)SQLStmt, "EXTERNAL NAME 'ClassName!MethodName' ");
    cout << "Procedure to prepare:" << endl;
    cout << SQLStmt << endl;
    rc = SQLPrepare(DclStmtHandle, SQLStmt, SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    {
        printError(ConHandle, DclStmtHandle);
    }
    return(rc);
}

SQLRETURN DeclarExample::executeSP()
{
    // Declare The Local Memory Variables
    SQLRETURN rc;
    SQLCHAR SQLStmt[256];

    // Prepare the statement to call the procedure
    strcpy((char *)SQLStmt, "CALL dlema.dosomething();");
    rc = SQLPrepare(SpStmtHandle, SQLStmt, SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    {
        printError(ConHandle, SpStmtHandle);
    }
    return(rc);

    // calling stored procedure
    rc = SQLExecute(SpStmtHandle);

    if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO)){
        printError(ConHandle, SpStmtHandle);
    }
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {

rc = SQLEndTran(SQL_HANDLE_DBC, ConHandle, SQL_ROLLBACK);
}
else {
    rc = SQLEndTran(SQL_HANDLE_DBC, ConHandle, SQL_COMMIT);
    cout << "Stored procedure call completed successfully." << endl;
}
return(rc);
}

SQLRETURN DeclarExample::printError (SQLHDBC hdbc, SQLHSTMT hstmt) {
    SQLCHAR buffer[SQL_MAX_MESSAGE_LENGTH + 1];
    SQLCHAR sqlstate[SQL_SQLSTATE_SIZE + 1];
    SQLINTEGER sqlcode;
    SQLSMALLINT length;
    SQLRETURN rc;

    while ((rc = SQLError(SQL_NULL_HENV, hdbc, hstmt, sqlstate, &sqlcode, buffer, SQL_MAX_MESSAGE_LENGTH + 1, &length) == SQL_SUCCESS) || rc == SQL_SUCCESS_WITH_INFO) {
        cout << "SQLSTATE: " << sqlstate << endl;
        cout << "SQLCODE : " << sqlcode << endl;
        cout << "Error msg : " << buffer << endl;
        cout << "----------------------------- " << endl << endl;
    }
    return(SQL_ERROR);
}

/*-----------------------------------------------------------------*/
/* The Main Function                                              */
/*-----------------------------------------------------------------*/
int main()
{
    // Declare The Local Memory Variables
    SQLRETURN   rc = SQL_SUCCESS;
    SQLCHAR     ConnectStr[128] = "DSN=QDSN_AS23;UID=TEAM01;PWD=PWDTEAM1;";

    // Create An Instance Of The DeclarExample Class
    DeclarExample declarExample;

    // Connect to the sample database
    if (declarExample.ConHandle != NULL)
    {
        rc = SQLDriverConnect(declarExample.ConHandle, NULL, ConnectStr, SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        {
            declarExample.printError(declarExample.ConHandle, declarExample.SpStmtHandle);
            return(rc);
        }

        // set autocommit off
        rc = SQLSetConnectAttr(declarExample.ConHandle, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER) SQL_AUTOCOMMIT_OFF, SQL_IS_UINTEGER);

        // Allocate An SQL Statement Handlers
        rc = SQLAllocHandle(SQL_HANDLE_STMT, declarExample.ConHandle, &declarExample.DclStmtHandle);
        rc = SQLAllocHandle(SQL_HANDLE_STMT, declarExample.ConHandle, &declarExample.SpStmtHandle);
// Now declare the stored procedure to be used
declarExample.declareSP();

// Execute the previously declared stored procedure
rc = declarExample.executeSP();
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
{
    declarExample.printError(declarExample.ConHandle,
        declarExample.SpstmtHandle);
}

// Return To The Operating System
return(rc);

Notes: The following notes refer to Example 4-2:
1. We assemble the DECLARE PROCEDURE statement. Be aware that its syntax is different from the CREATE PROCEDURE syntax.
2. The statement assembled in 1 is then prepared. Note that it is not executed, just prepared.
3. Now we are ready to create and use statements that call to the stored procedure declared in 2.

Keep in mind that DECLARE PROCEDURE is a nonstandard SQL command that is not being enhanced. If you are interested in portability, you must avoid using it.

4.4 System catalog tables

The database manager maintains a set of tables containing information about the data in each relational database. These tables are collectively known as the catalog. The catalog tables contain information about tables, user-defined functions, distinct types, parameters, procedures, packages, views, indexes, aliases, constraints, triggers, and languages supported by DB2 Universal Database for iSeries.

Every CREATE PROCEDURE statement, including the registration with iSeries Navigator, generates entries in the SYSROUTINES and SYSPROCS catalog tables. This section shows how to view the stored procedure information using the SYSROUTINES catalog, the SYSPROCS view, and the SYSPARMS view.

4.4.1 SYSROUTINES catalog

All stored procedures that are registered with the CREATE PROCEDURE statement or using iSeries Navigator are stored in the SYSROUTINES catalog. Refer to SQL Reference, SC41-5612, for a detailed description of the catalog views.

Note: The SYSROUTINES catalog contains information pertaining to both stored procedures and user-defined functions (UDF). You may use the SYSPROCS catalog view to work with stored procedures. The SYSFUNCS catalog view contains the information for the UDFs.
The following SQL statement displays the content of SYSROUTINES:

```sql
select * from qsys2.sysroutines;
```

The result includes information about all procedures and functions registered on DB2 Universal Database for iSeries. The number of returned rows may be very large on a busy production system. You should always try to narrow the scope of your query. The following SELECT statement retrieves relevant information about stored procedures registered in the SPROCLIB library:

```sql
select specific_schema, routine_name, routine_type, routine_body, parameter_style from qsys2.sysroutines where routine_schema = 'SPROCLIB' and routine_type = 'PROCEDURE';
```

If you run this statement in the Run SQL Scripts utility, the query results viewer displays the stored procedure details, as shown in Figure 4-3.

```
<table>
<thead>
<tr>
<th>SPECIFIC_SCHEMA</th>
<th>ROUTINE_NAME</th>
<th>ROUTINE_TYPE</th>
<th>ROUTINE_BODY</th>
<th>PARAMETERSTYLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPROCUS</td>
<td>HSALRES</td>
<td>PROCEDURE</td>
<td>EXTERNAL</td>
<td>GENERAL</td>
</tr>
<tr>
<td>SPROCUS</td>
<td>RANKTOT1</td>
<td>PROCEDURE</td>
<td>EXTERNAL</td>
<td>GENERAL</td>
</tr>
<tr>
<td>SPROCUS</td>
<td>RANKTOT2</td>
<td>PROCEDURE</td>
<td>EXTERNAL</td>
<td>GENERAL</td>
</tr>
<tr>
<td>SPROCUS</td>
<td>RANKTOT3</td>
<td>PROCEDURE</td>
<td>EXTERNAL</td>
<td>GENERAL</td>
</tr>
<tr>
<td>SPROCUS</td>
<td>RANKTOT4</td>
<td>PROCEDURE</td>
<td>EXTERNAL</td>
<td>GENERAL</td>
</tr>
<tr>
<td>SPROCUS</td>
<td>DELPGM3</td>
<td>PROCEDURE</td>
<td>EXTERNAL</td>
<td>NULLS</td>
</tr>
<tr>
<td>SPROCUS</td>
<td>DELPGM2</td>
<td>PROCEDURE</td>
<td>EXTERNAL</td>
<td>NULLS</td>
</tr>
</tbody>
</table>
```

Figure 4-3 Stored procedures in SPROCLIB library

If the stored procedure of interest is located at a specific schema or collection, the schema or collection catalog can be used instead, as follows:

```sql
select specific_schema, routine_name, routine_type, routine_body, parameter_style from myschema.sysroutines where routine_type = 'PROCEDURE';
```

### 4.4.2 SYSPARMS catalog

The SYSPARMS catalog contains one row for each parameter of a stored procedure created by the CREATE PROCEDURE statement. Refer to SQL Reference, SC41-5612, for the detailed layout of this catalog. The SYSPARMS catalog contains parameters for both UDFs and stored procedures.

Let us suppose that you want to retrieve the parameter details for all instances of the SELPGMRES stored procedure located in the SPROCLIB library. You can run the following SQL statement to display the required information:

```sql
select * from qsys2/sysparms where Specific_schema='SPROCLIB'
```

### 4.5 Procedure signature and procedure overloading

DB2 Universal Database for iSeries supports the concept of procedure overloading. This means that you can have two or more procedures with the same name in the same library, schema, or collection, provided they have different signatures. The signature of a procedure can be defined as a combination of the qualified name and the number of parameters in the procedure.

No two procedures in the library can have the same signature. Therefore, no two procedures with the same name and the same number of parameters can coexist in the same library.
For example, the following two stored procedures can coexist in the same library:

MyStorProc( char(5), int)
MyStorProc( int)

However, these two procedures cannot exist in the same library:

MyStorProc( char(5))
MyStorProc( int)

Procedure overloading is the reason why the RESTORE commands avoid overlaying existing stored procedures. If you try to restore a stored procedure to a library, where the same named procedure already exists, the system registers a new procedure instance rather than overlaying the existing one. Refer to 6.6, “Moving into production (save and restore)” on page 156, for more details.

**Important:** The stored procedure signature differs from the UDF signature. The UDF signature consists of a name, number, and types of parameters. The following two UDFs can coexist in the same library:

myUDF( char(5) )
myUDF( int )

Refer to Chapter 13, “User-defined functions” on page 449, for a detailed discussion on UDFs.

### 4.6 Deleting or replacing a stored procedure

When you create a procedure, its signature must be unique to register the procedure in the catalog. As described in 4.5, “Procedure signature and procedure overloading” on page 75, the signature of a procedure is defined based on the combination of the qualified name and the number of the parameters of the procedure. Note that the CREATE PROCEDURE statement does not have a replace option. For this reason, if you want to re-create or delete an existing procedure, use the DROP PROCEDURE statement.

#### 4.6.1 Using a command line to drop a procedure

There are several ways to drop a stored procedure from the iSeries server:

- In the traditional green-screen environment, start the Interactive SQL session with the command:

  `STRSQL NAMING(*SQL)
  `At the ISQL prompt, type following SQL statement:

  `DROP PROCEDURE library.procedure-name`

  Figure 4-4 shows the message that is issued after the procedure is successfully deleted.
In the iSeries Navigator environment, in the right panel of the main iSeries Navigator window, right-click the procedure you want to drop, and select the **Delete** option, as shown in Figure 4-5. A window opens that shows the stored procedure object that is selected for deletion. Confirm that this is the procedure that you want to delete, and click the **Delete** button.

In the Run SQL Scripts utility, insert the **DROP PROCEDURE library.procedure-name** statement in the workable area. Then select **Run -> All** from the menu bar.

The system catalog tables, **SYSROUTINES** and **SYSPARMS**, are updated when a **DROP PROCEDURE** statement is executed. In the **SYSROUTINES** table, a row is deleted corresponding to the information of the deleted procedure. In the **SYSPARMS** table, the number of rows deleted depends on the number of parameters defined in the procedure.

### 4.6.2 Dropping overloaded procedures

Dropping overloaded procedures can be tricky. Since the procedure name is overloaded, it is not sufficient to supply its name on the **DROP PROCEDURE** statement. There are two methods that can be used to properly resolve the overloaded name.
Let us suppose that you created the following two stored procedures:

create procedure myStoredProc(p1 int)
language sql
specific spint
BEGIN
  IF ( P1 = 0 OR P1 = 1 ) THEN UPDATE DUMMY SET COL1 = P1 ;
  END IF ;
END;

create procedure myStoredProc(p1 int, p2 char)
language sql
specific spintchar
BEGIN
  IF ( P1 = 0 OR P1 = 1 ) THEN UPDATE DUMMY SET COL1 = P2;
  END IF ;
END;

To drop the second procedure, you need to use one of the methods listed here:

- Specify the specific procedure name:
  drop specific procedure spintchar;
- Include the parameter types on the DROP PROCEDURE statement:
  drop procedure myStorproc( int, char );

### 4.7 Authorization and adopted authority

When a stored procedure is called by the client program, the statements in the stored procedure are executed with the authorities of the calling user or the authorities of the user, plus the authorities of the owner of the program object corresponding to that stored procedure, depending on how it was defined in the USRPRF attribute for that program object.

When USRPRF is set to *OWNER, the statements inside the program object use only the invoking user authorities. When USRPRF is set to *OWNER, the statements are executed with the authorities of the calling user, plus the authorities of the owner of the program object.

As a complement, there is a mechanism called adopted authorities. Adopted authority is whether a program inherits the authorities of its caller program, depending on the Use Adopted Authorities (USEADPAUT) parameter for the program object. Adopted authorities only has affect if USRPRF is set to *OWNER. Table 4-3 summarizes the effects of authorization and adopted authorities.

<table>
<thead>
<tr>
<th>Authorization (USRPRF)</th>
<th>Adopted authorities (USEADPAUT)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*OWNER</td>
<td>*YES</td>
<td>The program uses authorization from both the user and the program owner profiles. In addition, it inherits the authorities of the caller program.</td>
</tr>
<tr>
<td>*NO</td>
<td></td>
<td>The program uses authorization from both the user and the program owner profiles. But it does not inherit the authorities of the calling program.</td>
</tr>
<tr>
<td>*USER</td>
<td>*YES or *NO</td>
<td>The program only uses the user profile authorities.</td>
</tr>
</tbody>
</table>
Authorities and adopted authorities provide mechanisms for improving security. This may include giving access to sensitive objects to just a user or maybe a controlled set of users, and then making those users be the owners of the programs (including stored procedure programs, except for Java stored procedures) that access and modify them. Then end users can be granted execution authorities only to those programs, without giving them access to data objects such as application tables. For example, in a bank application, you do not want to grant access to account tables for each cashier (which can be very risky). Instead you grant them execution on the clerk frontend banking application.

4.8 Returning result sets from stored procedures

An SQL stored procedure can call another procedure, which in turn calls another procedure in a chain. This is called a nested SQL procedure. A facility is available for you to specify to which calling procedure the result sets of a specific called procedure are returned. This is called the returnability attribute.

Starting in V5R3, two new syntaxes were added to DB2 Universal Database for iSeries SQL. These syntaxes provide flexibility in designing whether the result set of a stored procedure, which is called in nested procedures, are to be returned to the immediate calling procedure or to the calling procedure that is at the beginning of the calling chain as shown in Figure 4-6.

![Figure 4-6  Returning result sets to a caller versus to a client]

You do this by adding either of the following returnability attributes to the DECLARE CURSOR or SET RESULT SET statement:

- WITH RETURN TO CALLER, which returns result sets to the immediate caller

Consider the following examples:

- DECLARE c1 CURSOR FOR WITH RETURN TO CALLER SELECT * FROM t1
- SET RESULT SETS WITH RETURN TO CALLER FOR ARRAY :array1 FOR :hv1 ROWS
WITH RETURN TO CLIENT, which returns result sets to the procedure at the beginning of the calling chain

The result sets are invisible to all the intermediate procedures in the chain. Consider the following examples:

- DECLARE c1 CURSOR FOR WITH RETURN TO CLIENT SELECT * FROM t1
- SET RESULT SETS WITH RETURN TO CLIENT FOR CURSOR x1

Before V5R3, a stored procedure always returns its result sets to its immediate caller. Starting in V5R2, RETURN TO CALLER is still a default returnability attribute if DECLARE CURSOR or SET RESULT SET does not have the returnability attribute syntax explicitly specified.

**Note:** You must have the latest Database Group PTF installed.
SQL stored procedures

DB2 Universal Database for iSeries was the first member of the DB2 family to support the SQL procedural language and make this language available for the development of SQL user-defined functions (UDFs) and SQL triggers. The greatest advantage to using the SQL stored procedure language is portability. You can often use the same stored procedure with other relational database management systems (RDBMS).

This chapter covers:

- Transaction management and SQL procedures
- Calling from client applications
- Stored procedures and commitment control
- Considerations regarding SQL stored procedures
5.1 Introduction

From the iSeries server point-of-view, there are two ways to implement stored procedures. One way is to write the procedure in any high-level language program. This kind of procedure is described as an external procedure. This approach gives you the flexibility to use a language you are familiar with, such as C, CL, RPG, COBOL, and so on. The second approach, widely used by other DBMS providers, is to make available SQL-only stored procedures, which are described as SQL procedures. This second approach makes it easier to port stored procedures from other DBMSs to the iSeries server and vice versa.

The SQL CALL statement is used for the stored procedure invocation. The application waits for the stored procedure to terminate. Parameters can be passed back and forth. Stored procedures can be called locally (on the same system where the application runs) or remotely on a different system.

Note: The CL CALL command cannot be used to invoke an SQL stored procedure.

All the benefits of stored procedures that are discussed in 4.1, “Introduction” on page 64, also apply to SQL stored procedures.

5.2 Structure of an SQL stored procedure

An SQL procedure consists of:

- A procedure name
- A sequence of parameter declarations
- The procedure properties (defining number of result sets, whether the procedure is deterministic, and the kind of SQL access that is included in the stored procedure)
- A set of parameters that control the way in which the stored procedure is created
- A routine body

The general structure of an SQL procedure is shown in the following example. The numbers are explained in detail in the note box that follows.

```
CREATE PROCEDURE name-of-procedure
   (List of the input or output parameters of the procedure)
   Procedure properties
   Generation options
   Routine body of the procedure.
```
Chapter 5. SQL stored procedures

5.2.1 Example of a single SQL statement stored procedure

This SQL procedure receives a customer number and customer name, and updates the customer file with the new name:

```sql
CREATE PROCEDURE UPDCUST
(IN i_cusnbr CHARACTER(5),
 IN i_cusnam CHARACTER(20))
LANGUAGE SQL
UPDATE ordapplib.customer SET CUSNAM = i_cusnam
 WHERE cusnbr = i_cusnbr;
```
An SQL stored procedure with a single statement is not very common since the real value of a stored procedure is bundling together multiple database operations.

5.2.2 Example of a compound SQL statement

Here is a more complicated example in which the procedure receives an input parameter, which is the percentage of increase for the customer credit limit. This increase is applied to all customers. The procedure returns the number of records that were updated to the calling program:

```sql
CREATE PROCEDURE CREDITP
(IN i_perinc DECIMAL(3,2),
 OUT o_numrec DECIMAL(5,0))
LANGUAGE SQL
BEGIN ATOMIC
DECLARE proc_cusnbr CHAR(5);
DECLARE proc_cuscrd DECIMAL(11,2);
DECLARE numrec DECIMAL(5,0);
DECLARE at_end INT DEFAULT 0;
DECLARE not_found CONDITION FOR '02000';
DECLARE c1 CURSOR FOR
SELECT cusnbr, cuscrd
FROM ordapplib.customer;
DECLARE CONTINUE HANDLER FOR not_found
SET at_end = 1;
SET numrec = 0;
OPEN c1;
FETCH c1 INTO proc_cusnbr, proc_cuscrd;
WHILE at_end = 0 DO
  SET proc_cuscrd = proc_cuscrd +(proc_cuscrd * i_perinc);
  UPDATE ordapplib.customer
  SET cuscrd = proc_cuscrd
  WHERE CURRENT OF c1;
  SET numrec = numrec + 1;
  FETCH c1 INTO proc_cusnbr, proc_cuscrd;
END WHILE;
SET o_numrec = numrec;
CLOSE c1;
END
```

Notes: The following notes refer to the previous example.

1 This defines the name of the procedure, which is UPDCUST.
2 There are two input parameters: i_cusnbr and i_cusnam.
3 Since this is an SQL procedure and not an external one, LANGUAGE SQL is specified.
4 The last two lines are the SQL procedure body, which consists of a single SQL UPDATE statement.
5.3 Creating an SQL stored procedure

Now that you know the general structure of an SQL procedure, you are ready to create one. This section documents the steps required to edit and compile an SQL procedure. There are many ways to build your SQL stored procedure. You can use the following methods:

- iSeries Navigator GUI
- iSeries Navigator SQL script utility
- DB2 Universal Database Version 8 Development Center
- Traditional 5250 interactive SQL interface using the STRSQL CL command
- Traditional 5250 interactive SQL interface using SEU and RUNSQLSTM utilities

5.3.1 Creating an SQL stored procedure with iSeries Navigator

iSeries Navigator provides an attractive graphical interface that allows you to perform typical database administration tasks. It allows easy access to all server administration tools, gives a clear overview of the entire database system, enables remote database management, and provides assistance for complex tasks.

In this section, you learn how to efficiently use the GUI administration tools offered by iSeries Access Express to work with SQL stored procedures on the iSeries server. Prior to reading this section, you should already know how to set up the iSeries Navigator connection to your iSeries server.

The following steps show how to create an SQL stored procedure using the Create New SQL Procedure window:

1. Double-click the iSeries Navigator icon on your desktop. In the main panel, right-click the library that contains your database objects. In our case, the name of the library is ORDAPPLIB. Select New -> Procedure -> SQL.

2. The New SQL Procedure window opens:
   a. Enter CASEPROC for the stored procedure name.
   b. For the description, type:
      Update customer credit depending on the evaluation.
   c. For the maximum number of result sets, type 0 (zero).

Notes: Note the following points:

1. The name of the procedure is defined as CREDITP.
2. It has an input parameter, which is i_perinc (the percentage of increase for all customers).
3. It has an output parameter, which is o_numrec (the number of records that were updated).
4. The rest of the lines are the SQL procedure body that consists of a single compound SQL control statement. This compound statement permits you to group other SQL control statements and SQL procedure statements together. Every compound statement starts with the clause BEGIN.
5. Every compound statement ends with an END clause.

This example is explained in detail later in this chapter.
d. Click the **Parameters** tab.

e. Click the **Insert** button. For the first parameter name, type:

   cust_no

   From the type-drop down list, select **CHARACTER**. In the parameter length box, enter the number 5. Change the parameter style to IN.

f. Insert the second parameter as shown in Figure 5-1.

---

*Figure 5-1  Parameter definition for the SQL stored procedure*
g. Click the **SQL Statements** tab. Type the procedure body as shown in Figure 5-2.

![Figure 5-2  Entering SQL statements](image)

**Note:** Two syntax errors are intentionally inserted in the code:

1. The variable name does not coincide with the parameter name.
2. The CASE statement must finish with END CASE.

h. Click the **OK** button to create the stored procedure.

3. You see an error message like the example in Figure 5-3. For more details, click the **Job Log** button.

![Figure 5-3  Syntax error on SQL stored procedure](image)
4. You see the job log messages as shown in Figure 5-4. If you look carefully in the job log, you see the first syntax error detected by DB2 Universal Database for iSeries.

To view a second level message in the job log, double-click the item that you want to view. A window opens that shows all of the information for that message.

![Job Log error messages](image)

5. Correct the first syntax error by changing customer_nbr to cust_no in the code. Then retry the generation by clicking the OK button again.

6. A second error message that corresponds to the second syntax error is introduced in the code. Note that with this approach, you fix one error at a time. Correct the second syntax and retry the generation. If everything goes well, you will have a new stored procedure.

### 5.3.2 Creating an SQL stored procedure with the Run SQL Scripts utility

The Run SQL Scripts utility is yet another interface that you can use on the iSeries server to create a stored procedure. The script utility is available through the iSeries Navigator GUI. It allows you to create, edit, run, and troubleshoot scripts of SQL statements. You can also save the scripts with which you work on your PC.

To create an SQL stored procedure using the SQL script utility:

1. Double-click the iSeries Navigator icon on your desktop. In the main panel, right-click the Databases icon, and select Run SQL Scripts.
2. In the Run SQL Scripts window, type the procedure body as shown in Figure 5-5.

```
CREATE PROCEDURE ORDAPPLIB.CASEPROC2(
    IN CUST_NO CHARACTER(5),
    IN EVALUA DECIMAL(3,0))
BEGIN
    LANGUAGE SQL
    CASE
    WHEN EVALUA > 90 THEN UPDATE ORDAPPLIB.CUSTOMER
        SET CUSORD = CUSORD * 3
        WHERE CUSORD = CUSTOMER.NBR;
    WHEN EVALUA > 80 THEN UPDATE ORDAPPLIB.CUSTOMER
        SET CUSORD = CUSORD * 2
        WHERE CUSORD = CUSTOMER.NBR;
    WHEN EVALUA > 70 THEN UPDATE ORDAPPLIB.CUSTOMER
        SET CUSORD = CUSORD * 1.2
        WHERE CUSORD = CUSTOMER.NBR;
    ELSE UPDATE ORDAPPLIB.CUSTOMER
        SET CUSORD = CUSORD * 0.5
        WHERE CUSORD = CUSTOMER.NBR;
    END;
END;
```

*Figure 5-5  Creating an SQL stored procedure with the Run SQL Scripts utility*

**Note:** Two syntax errors are intentionally inserted in the code:

1. The variable name does not coincide with the parameter name.
2. The CASE statement must finish with END CASE.
3. To run the CREATE PROCEDURE statement, select Run -> All. You see an error message in the Message tab, as shown in Figure 5-6.

![Syntax error on the SQL stored procedure (RUN SQL Scripts utility)](image)

As explained in 5.3.2, “Creating an SQL stored procedure with the Run SQL Scripts utility” on page 88, syntax errors can be corrected one by one. When the procedure is free of errors, the last message in the Messages frame of the Run SQL Script window should read:

Statement ran successfully

If the run history panel does not supply sufficient information about the execution of the SQL statement, you can view the iSeries server job log to obtain additional, more specific messages. From the View drop-down menu, select Job Log. A job log window similar to that shown in Figure 5-4 on page 88 opens.

4. Correct the error messages and generate the stored procedure.

To save the script that contains the source code for the CASEPROC2 stored procedure, select File -> Save As from the script utility menu bar. The Save As window is displayed. In the Save in list combo, open the directory you want to use as your SQL script repository. In our case, we used the d:\sg24_6503\WIP directory. Enter caseproc in the file name input field. Then, click Save to return to the Run SQL Script window.
The Run SQL Scripts utility proved to be very useful when we ported the SQL stored procedures from other DB2 Universal Database platforms to the iSeries server system. We simply copied the scripts to our working directory and changed the file extension from .stp to .sql. Then we can double-click a stored procedure file from the Windows Explorer window to load the script into the Run SQL Scripts utility.

5.3.3 Creating an SQL stored procedure with DB2 Universal Database Development Center

DB2 Universal Database for UNIX®, Windows, and Others (UWO) Version 8 introduced a new development tool known as Development Center. This new tool improved the functionality of the old Stored Procedure Builder available as part of the DB2 Universal Database for UWO Version 7, allowing the developer to develop stored procedures, triggers, and UDFs in SQL, C, C++ and Java for UWO. This tool has limited support for DB2 Universal Database for iSeries, supporting SQL stored procedures only.

To use this tool, you need IBM DB2 Universal Database for UWO Enterprise Server Edition (ESE) Version 8 or IBM DB2 Connect™ Version 8 or later installed. The DB2 Universal Database for iSeries databases in which you are interested should be cataloged.

To create SQL stored procedures with DB2 Universal Database Development Center:

1. From the Windows Start menu, select Programs -> Programs -> IBM DB2 -> Development Tools -> Development Center.

2. If you are running the Development Center for the first time, it will open the Project Creation window for you. Otherwise, you can create a new project if you need it by selecting Project -> New Project from the main menu. The Project Creation window is shown in Figure 5-7. Projects are used to put together related source code.

![Figure 5-7 Development Center’s Open Project window](image)

In our example, a project called SC24-6503 was created.
3. After a new project had been created, connections to databases should be declared. In the upper left frame of the DB2 Development Center, expand the project and right-click **Database Connections** and click **Add Connection** from the pop-up menu. A wizard is provided to help in the connection definition, as shown in Figure 5-8 on page 92. In the first step select the desired type of connection. If you select online, the connection will be established when you start working with the project. If you select offline, the connection will be established when the Development Center requires to access the database for accomplishing a task.

4. After the connection type has been established, the characteristics for the connection should be established. Select the proper driver for the connection. Under the covers, Development Center is a Java application and has several different drivers that can be used to connect to iSeries databases. In general, all three options work. Select also the Database alias (the name by which the DB2 Universal Database for UWO or DB2 Universal Database Connect knows the iSeries database, not the iSeries database actual name), and the user ID and password to be used by this connection.

![Figure 5-8 Adding a new database connection for a project](image)

5. Define the default schema. Schema and collection are synonymous in DB2 Universal Database for iSeries. The package and build owner is irrelevant to DB2 Universal Database for iSeries. In Figure 5-9, a default SQL schema has been defined as ORDAPPLIB. At this point, click **Finish** to complete the task or click **Next** to review the selected options.
Now that the desired connections have been defined, stored procedures can be developed.

Expand your project. Inside your project, expand the database connection folder and expand the database in which the stored procedure will reside. Right-click the **Stored Procedure** folder. Placing the cursor over the New option will open a side menu offering two ways to create stored procedures. No matter what you select the final result will be equivalent, but you will get more help using the Wizard. That is our choice while you are not familiarized with the SQL stored procedure.
By choosing the wizardized path, we get the menu of options that can be developed using DB2 Universal Database Development Center. That means that it will offer you only one option: To build a SQL stored procedure, which is the only option supported at the time we were writing this book.

Choosing the only available option and clicking **OK** drives us to the Create SQL stored procedure Wizard. This wizard allows us to create a store procedure in five steps:

1. Define a name for the stored procedure, as shown in Figure 5-11.

![Figure 5-11: Defining a name for the SQL stored procedure](image)

*Figure 5-11: Defining a name for the SQL stored procedure*
2. We can set up stored procedure characteristics such as SQL statements for the body, number of result sets the stored procedure is going to return, and the way in which errors are going to be treated, as shown in Figure 5-12.

By clicking the three stars at the right of the value box for Statement, an SQL builder assistant window opens, as shown in Figure 5-13.
You can choose between single statement, multiple statement or no statements stored procedure. The SQL Assist button will provide you with additional assistance for creating select, insert, update, and delete SQL statements. It uses the same assistant present in the DB2 Universal Database Control Center and that was included in the Run SQL Script since V5R2.

In our case, we were interested in a single statement stored procedure, but our statement is not an insert, select, update or delete. Then we choose generate no SQL statement.

You also have to specify the number of result sets that the stored procedure will return, the way in which errors are going to be treated and the fragment you want to be used as header, variable declaration, error handlers and pre-return. Those fragments are text files that the wizard will insert into the stored procedure. In a large project, this will reinforce standardization for stored procedure source coding.

3. After you define the SQL statement or statements for the stored procedure, it is time to specify parameters. As shown in Figure 5-14, you can add, change, remove and reorder the parameters required by the stored procedure.

![Figure 5-14 Parameters specification](image)

For each parameter, a parameter name, mode, SQL type, length or precision and scale, and comments can be supplied.

4. After the parameters are defined, a couple of options can be set up:
   - A specific name, which is the name of the C program that the iSeries creates for the stored procedure
   - A check box indicating the wizard if we want the stored procedure to be built as soon as we finish or we want to built it later in the process
If a specific name is not specified, a C program with the same name of the stored procedure will be generated in the library associated to the schema or collection of the stored procedure.

We choose not to build the stored procedure, since we need to edit and add some code in it before. We finish the wizard and our new stored procedure is created in the project.

With those steps, a SQL stored procedure without statements was created. Now we need to edit to complete and build the stored procedure. In the upper left corner of the project view, expand the stored procedure folder for our database connection, right-click it, and select the Edit option from the pop-up menu. We edit the stored procedure introducing some errors as in our previous examples. To build the stored procedure, use the wrench icon in the edit view.

As the DB2 Universal Database for iSeries detects errors in the stored procedure, you will see in the output view, the execution history at the left and the log messages on the right side, as shown in Figure 5-16. Note that one error is detected at a time.

After all errors are corrected, the output view shows the stored procedure building step successfully accomplished.
5.3.4 Creating an SQL stored procedure with traditional 5250 tools

The steps required to create SQL stored procedures with traditional 5250 tools are summarized here:

1. Create a library if you do not have one already.
2. Create a source physical file. This is the file where the SQL source members are going to be stored.
3. Start a Source Entry Utility (SEU) editing session.
4. Enter the SQL procedure source code.
5. Create the SQL procedure using the Run SQL Statement (RUNSQLSTM) command to issue a CREATE PROCEDURE command. This creates a C program object that runs when the procedure is called. If there are problems generating the procedure, there is a listing that shows the syntax errors of the source.
6. Invoke the stored procedure through the SQL CALL statement passing the parameter list.
7. Check for the completion status of the SQL procedure.

Let's see how to implement this scenario. First, create a library, a source file, and start an editing session. Follow these steps:

1. To create a library called ordapplib, type the following CL command at the 5250 emulation prompt:
   ```plaintext
   CRTLIB LIB(ORDAPPLIB)
   ```
2. To create a source physical file called QSQLSRC, type the command:
   ```plaintext
   CRTSRCPF FILE(ORDAPPLIB/QSQLSRC) RCDLEN(112)
   TEXT('Source physical file for SQL Procedures')
   ```
The CRTSRCPF command creates a source physical file QSQLSRC in the library ORDAPPLIB.

3. To start an editing session and create a source member, CASEPROC, type the command:

```
STRSEU SRCFILE(ORDAPPLIB/QSQLSRC) SRCMBR(CASEPROC)
   TYPE(TXT) OPTION(2)
```

Entering OPTION(2) indicates that you want to start a session for a new member. The STRSEU command creates a new member, CASEPROC, in the QSQLSRC file in the ORDAPPLIB library and starts an edit session.

After typing the source, you see a display similar to the example in Figure 5-17.

```
Columns . . . :  1  71            Edit                     ORDAPPLIB/QSQLSRC
SEU==>                                                                CASEPROC
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
*************** Beginning of data ****************************************
0001.00 CREATE PROCEDURE CASEPROC
0002.00     (IN i_cusnbr CHARACTER(5,
0003.00      IN i_evalua DECIMAL(2))
0004.00      LANGUAGE SQL
0005.00  CASE
0006.00    WHEN i_evalua > 90
0007.00    THEN UPDATE proclibnn/customer SET cuscrd = cuscrd * 1.3
0008.00         WHERE cusnbr = i_cusnbr;
0009.00    ELSE UPDATE proclibnn/customer SET cuscrd = cuscrd * 1.1
0010.00         WHERE cusnbr = i_cusnbr;
0011.00    END CASE;
0012.00  END CASE;
*************** End of data *******************************************
```

Notes: The following notes refer to Figure 5-17:

1 Syntax error to produce the error listing (missing closing parenthesis).

2 If you adopt "SQL as your naming convention in RUNSQLSTM, you should use "." as the library delimiter. In case you adopt "SYS, you should use "/" instead. Make your choice carefully since the "SYS convention is not supported on other platforms.
4. Use the RUNSQLSTM command to create the procedure (Figure 5-18). We recommend that you use the Debugging view option *LIST and Listing output *PRINT. It is useful for debugging and testing purposes.

5. If there are syntax errors in your source, a message similar to the example shown in Figure 5-19 is issued.

---

**Notes:** The following notes refer to Figure 5-18:

1. Type the name of the source file (QSQLSRC) and the library (ORDAPPLIB) that you created before.

2. The name of the member source file that you typed.

3. This parameter specifies the type of source debug information to be provided by the SQL pre-compiler. The possible values are:
   - *STMT*: Allows the compiled module object to be debugged using program statement numbers and symbolic identifiers.
   - *NONE*: The debug view is not be generated.
   - *LIST*: Generates the listing view for debugging the compiled module object.

4. If you want the pre-compiled listing, type *PRINT in this parameter.

---

**Command Entry**

Previous commands and messages:
> RUNSQLSTM SRCFILE(ORDAPPLIB/QSQLSRC) SRCMBR(CASEPROC) COMMIT(*NONE)

RUNSQLSTM command failed.

Type command, press Enter.
===> RUNSQLSTM SRCFILE(ORDAPPLIB/QSQLSRC) SRCMBR(CASEPROC) COMMIT(*NONE)

---

**Figure 5-18** Creating the SQL procedure by using the RUNSQLSTM command

**Figure 5-19** RUNSQLSTM command fails
It is important to note that when you are creating an SQL procedure, SQL creates a temporary source file that contains C source code with embedded SQL statements. The actual procedure is created as a *PGM object using the CRTSQLCI command and the CRTPGM command.

Three listings are generated as highlighted in Figure 5-20:

1. This first one is the source listing of the generated SQL ILE C program.
2. This is the listing of the pre-compiled SQL ILE C program.
3. This is the listing of the source of the SQL procedure. You have to look at this listing in case of a failure to create the procedure. You can look at it by typing 5 next to the listing. If the SQL procedure has syntax errors, the first two listings are not generated.

Note: If the RUNSQLSTM command fails (see the message in bold in Figure 5-19) to create the SQL procedure, go to the listing of the program by typing the WRKSPLF command at the command prompt (Figure 5-20).
6. In the preceding listing (Figure 5-21), there is a syntax error that probably generated the other ones. Correct the errors by going to an editing session and execute the RUNSQLSTM command again.

After the procedure is successfully created, two system catalog tables are updated: SYSROUTINES and SYSPARMS. The SYSROUTINES table contains one row for each procedure created by the CREATE PROCEDURE statement. The SYSPARMS table contains one row for each parameter of a procedure created by the CREATE PROCEDURE statement.

After the procedure is created, it can be invoked with the SQL call statement using any interface that supports SQL (embedded SQL, ODBC, JDBC, SQLJ, CLI, and so on).

5.3.5 Verifying the stored procedure properties

After the stored procedure is successfully created, verify its properties by using the iSeries Navigator interface:

1. In iSeries Navigator, double-click the ORDAPPLIB library icon. The right panel displays all DB2 Universal Database for iSeries objects in this library.
2. Find the CASEPROC stored procedure icon, right-click, and select Properties.
3. The CASEPROC Properties window opens. It has three tabs:
   - General page: Specifies the name by which the procedure is known to SQL programs and the number of result sets it should return. If you want to call an external program as a procedure, you need to define the program as a procedure before you can call it from an SQL program.
   - Parameters page: Specifies the parameters that the procedure uses.
   - SQL Statements page: Contains the code for the external SQL program that you are defining as a procedure. You can use the SQL statement examples and fill in the...
necessary information to make coding SQL easier. After an SQL procedure is created, the SQL statements cannot be changed.

**Note:** iSeries Navigator has a feature called Generate SQL that allows you to reconstruct SQL statements used to create existing database objects. With this function, you can reverse engineer database objects and then have the option to display the resulting SQL in the Run SQL Scripts window or save the output to a file. Using the existing Run SQL Scripts functions, you can then edit, run, and save the SQL statement to a file on the PC. Refer to 3.11, “Reverse engineering and Generate SQL” on page 54, for more information.

### 5.4 System catalog tables

On DB2 Universal Database for iSeries, system catalog tables register details of stored procedures, as described in 4.4, “System catalog tables” on page 74. These tables keep track of all the stored procedures. For an SQL stored procedure, the source code is stored in the SYSPROCS view in the ROUTINE_DEFINITION column. If the SQL stored procedure source is more than 24 KB, the source code is not stored in SYSPROCS, and ROUTINE_DEFINITION is then NULL.

### 5.5 SQL procedures returning result sets

An SQL stored procedure, similarly to the external stored procedures, can return one or multiple result sets to the calling process.

#### 5.5.1 Creating result sets in an SQL stored procedure

To return a result set to a calling application, an SQL stored procedure needs to declare a cursor on the selected rows. Multiple result sets can be returned, but each requires an independent DECLARE CURSOR statement. Here is an example:

```sql
/* Enter one or more SQL statements separated by semicolons */
CREATE PROCEDURE GetCusName()
    RESULT SETS 1
    LANGUAGE SQL
BEGIN
    DECLARE c1 CURSOR WITH RETURN FOR
        SELECT cusnam FROM customer ORDER BY cusnam;
    OPEN c1;
END
```

The caller program has the responsibility for closing those cursors that the stored procedure returns as result sets. Otherwise, an SQLSTATE 24502 (SQLCODE 0502) error may be fired by the stored procedure in subsequent calls because it will find them already opened.

Notes:

1. When a result set returns in your procedure, you should include the clause RESULT SETS in your CREATE PROCEDURE declaration. RESULT SETS specifies the maximum number of result sets that can be returned from the procedure.
2. This statement declares a cursor for a SELECT statement. The WITH RETURN clause indicates that the cursor is intended for use as a result set from a stored procedure.
3. You must open the cursor before it can be returned to the caller.
In V5R2, scrollable result sets were introduced. In previous releases, only read-only nonscrollable result sets were supported.

The SET RESULT SET support allows the same thing to be done with a proprietary syntax shown in the next example:

```
CREATE PROCEDURE GetCusName()
RESULT SETS 1
LANGUAGE SQL
BEGIN
    DECLARE c1 CURSOR FOR SELECT cusnam FROM customer ORDER BY cusnam;
    OPEN c1;
    SET RESULT SETS CURSOR c1;
END;
```

**Note:** For portability across different DBMS providers, the WITH RETURN clause for defining result sets is preferred.

### 5.5.2 Retrieving result sets in the caller

You may use ODBC, JDBC, or CLI to retrieve result sets. Software vendors, such as Microsoft, offer object models conforming to the Microsoft Component Object Model (COM) that mask the complexity of the ODBC calls. For example, Microsoft's MSDASQL OLE DB provider is a wrapper or a bridge to the ODBC data sources. Any ADO or OLE DB application can use the MSDASQL provider to access the databases using an object-oriented programming approach. The MSDASQL provider translates the ADO/OLE DB requests into ODBC API calls, which are then passed to the ODBC driver for processing. The iSeries Access ODBC driver may be used by the MSDASQL provider, so an ADO/OLE DB application can manipulate iSeries server data through ODBC.

The iSeries Access package also offers the native OLE DB provider called IBMDA400, which bypasses the MSDASQL wrapper layer. IBMDA400 does not use the iSeries Access ODBC driver to access SQL data. Note that the iSeries Access OLE DB provider has less functionality than the iSeries Access ODBC driver. Refer to *A Fast Path to AS/400 Client/Server Using AS/400 OLE DB Support*, SG24-5183, for a detailed discussion on different implementations of the OLE DB specification available for the iSeries server.

The following example shows how an application can specify these providers:

```
"Provider=IBMDA400; Data Source=<Client Access configured system name>"
"Provider=MSDASQL; Data Source=<ODBC data source name>"
```

Note that the current implementation of the IBMDA400 provider does not support multiple result sets. In this case, you should use MSDASQL as your OLE DB provider.

**Notes:** Some traditional programming languages, such as RPG and COBOL, cannot retrieve result sets returned by a stored procedure, unless they use the SQL Call Level Interface for the stored procedure call.

### 5.5.3 Using ADO in the Visual Basic client to retrieve result sets

The following Visual Basic program contains only one list box, lstCustomer, and one button, cmdGetCustName. When you click the button, the stored procedure, getcusName, on the iSeries server is invoked. The returned result set is passed to an ADO recordset object, Rs. A while-loop is used to look through each item in the recordset and add the items to the list box.
Const cnsSource = <ODBCSource>  
Const cnsUserID = <UserID>  
Const cnsPasswd = <Password>

Option Explicit
Dim Cnn As ADODB.Connection
Dim Cmd As ADODB.Command
Dim Rs As ADODB.Recordset
Dim strSQL As String
Private Sub cmdGetCusName_Click()
    Set Rs.Source = Cmd
    Rs.Open  
    While Not Rs.EOF
        lstCustomer.AddItem Rs(0)
        Rs.MoveNext
    Wend
    Rs.Close
End Sub
Private Sub Form_Load()
    Set Cnn = New ADODB.Connection
    strCnn = "PROVIDER=MSDASQL;dsn=" & cnsSource & " ;uid=" & cnsUserID & " ;pwd=" & cnsPasswd & ";"
    With Cnn
        .ConnectionString = strCnn
        .CursorLocation = adUseClient
        .Open
    End With
    strSQL = "{call library.getCusName()}
    Set Cmd = New ADODB.Command
    Set Cmd.ActiveConnection = Cnn
    Cmd.CommandText = strSQL
    Cmd.CommandType = adCmdText
    Set Rs = New ADODB.Recordset
    Rs.CursorType = adOpenStatic
    Rs.LockType = adLockReadOnly
End Sub
Private Sub Form_Unload(Cancel As Integer)
    Cnn.Close
    Set Cnn = Nothing
    Set Cmd = Nothing
    Set Rs = Nothing
End Sub

Notes: The following notes refer to the previous example.

1. You should replace ODBCSource with your ODBC source name, and do the same for UserID and Password.

2. The stored procedure is not executed until the result set is opened.

3. In this case, the MSDASQL OLE DB provider is employed.

4. You can either invoke the stored procedure by using the adCmdText command type or the adStoredProcedure command type. The adCmdText command type is more general in nature. You can pass any valid SQL statement through this command type. You can find an example of using the adStoredProcedure command type in 5.7.3, “Calling an SQL procedure from the Visual Basic client application” on page 115.
5.6 Global Temporary Table: A result set alternative

The SQL procedural language makes it easy for you to create stored procedures. The constructs in this language enable you to code conditional checks, to perform iterative processing with various looping constructs, to call other stored procedures, and to efficiently return sets of data via result sets.

Result sets are commonly used because they can contain data from multiple tables. They can also improve the runtime performance of a procedure by returning a data set in blocks instead of one row at a time. However, result sets of an stored procedure are not always universally accessible (or consumed) by all other stored procedures. We discuss a mechanism that helps address this issue in this section.

5.6.1 Purpose of a Global Temporary Table

In DB2 Universal Database for iSeries, result sets of stored procedures can only be accessed by client programs using the ODBC, JDBC, or CLI programming interfaces but not by a direct SQL procedure call that is initiated by a program running in OS/400. If an SQL procedure within OS/400 calls another stored procedure (also running in OS/400) that returns a result set, the calling SQL procedure cannot directly access the contents of the result sets returned by the called stored procedure.

This limitation also applies to embedded SQL codes. The techniques discussed in this section are also applicable to embedded SQL. This limitation only exists in DB2 Universal Database for iSeries. The other DB2 Universal Database products support additional SQL statements that eliminate this restriction. You can use Global Temporary Tables, an SQL feature introduced in V5R2, to address this limitation.

A Global Temporary Table is used to hold temporary data for a database connection (job) or application. Instead of a procedure returning data via result sets, the data is placed into a Global Temporary Table object instead. Then, with a prior knowledge of the agreed-upon name of the temporary table, the invoking procedure reads from the temporary table to access the result set data.

You may wonder if Global Temporary Tables are different from creating temporary tables in a specific library (or schema). They are different because the SQL Global Temporary Tables are always automatically created in the QTEMP library by DB2 Universal Database for iSeries. Despite the usage of the QTEMP library, SQL always requires the qualifier, whether explicit or implicit, for the temporary table name to be the syntax SESSION. DB2 Universal Database for iSeries does not create a library named SESSION. It uses the SESSION qualifier as an alias for the QTEMP library. Like any objects created in QTEMP library, DB2 Universal Database for iSeries automatically deletes all the temporary tables when the connection (or a job that runs the procedure) ends.

You the DECLARE GLOBAL TEMPORARY TABLE statement to create a temporary table. This statement can contain the following useful clauses:

- **WITH REPLACE clause**
  
  This clause provides an ability to address the situation where there are multiple executions of the same procedure (that declares the temporary table) within the same connection (or job). If a declared Global Temporary Table already exists with the specified name, the existing table is replaced with the temporary table defined by the statement and all rows of the existing table are deleted.

  If this clause in not specified, the statement fails if a Global Temporary Table (with the same name as that specified by the statement) already exists within the connection.
ON COMMIT and ON ROLLBACK clauses

This option enables you to treat the data in the temporary table in accordance with the policy of your database transactions. You use the following additional syntax to control this:

- **DELETE ROWS**

  You use this syntax to automatically delete all the existing rows in the temporary table to prepare for the next transaction at every COMMIT or ROLLBACK operation. You use this syntax when you do not declare a WITH HOLD cursor on the temporary table.

- **PRESERVE ROWS**

  You use this syntax to keep the existing data in the temporary table if your business rules dictate that the existing data is needed for the next transaction.

The ON COMMIT and ON ROLLBACK clauses can be used only if a commitment control or isolation level is set to a value other than "NONE.

5.6.2 Storing a result set into a Global Temporary Table

Let us look at an example that demonstrates the utility of the Global Temporary Table. In our example, a stored procedure named Get_Free_Employees returns a result set that contains those employees who are currently free from any project engagements. An input parameter is used to limit the search for free employees to a specific department.

The procedure header includes the count of result sets returned by the procedure (RESULT SETS 1). Within the procedure itself, result sets are returned by including the WITH RETURN clause on the cursor definition and then leaving that cursor open before exiting the procedure. See Example 5-1.

Example 5-1  Get_Free_Employees procedure that returns a result set

```sql
CREATE PROCEDURE Get_Free_Employees (IN dept CHAR(3))
LANGUAGE SQL
RESULT SETS 1
BEGIN

IF dept NOT IN ('D11','D21','ALL') THEN
    SIGNAL SQLSTATE VALUE '75001' SET MESSAGE_TEXT = 'Invalid department input';
END IF;

DECLARE free_emp CURSOR WITH RETURN TO CALLER FOR
    SELECT distinct firstname AS fname, lastname AS lname, workdept AS deptnum
    FROM employee e
    WHERE workdept=dept AND
    /* Check that the current employee has completed all of their projects */
    (CURRENT DATE > ALL (SELECT emp_enddate FROM emp_activeprojects p
        WHERE e.empno=p.empno AND emp_enddate IS NOT NULL)
    /* And that the employee does not have an active project (ie, no end date value) */
    AND empno NOT IN (SELECT empno FROM emp_activeprojects WHERE emp_enddate IS NULL));

OPEN free_emp;
END;
```
Figure 5-22 shows a sample result set of this procedure.

<table>
<thead>
<tr>
<th>FNAME</th>
<th>LNAME</th>
<th>DEPTNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irving</td>
<td>Stern</td>
<td>D11</td>
</tr>
<tr>
<td>Bruce</td>
<td>Adams</td>
<td>D11</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>Pianka</td>
<td>D11</td>
</tr>
<tr>
<td>David</td>
<td>Brown</td>
<td>D11</td>
</tr>
<tr>
<td>William</td>
<td>Jones</td>
<td>D11</td>
</tr>
<tr>
<td>Jennifer</td>
<td>Lutz</td>
<td>D11</td>
</tr>
<tr>
<td>Reba</td>
<td>John</td>
<td>D11</td>
</tr>
</tbody>
</table>

Figure 5-22  A sample result set returned by Get_Free_Employees procedure in Example 5-1

Example 5-2 shows a modified version of Example 5-1 that uses the Global Temporary Table technique for storing its result set.

Example 5-2   Get_Free_Employees_G procedure that uses a temporary table to store the result set

```
CREATE PROCEDURE Get_Free_Employees_G (IN dept CHAR(3))
LANGUAGE SQL
BEGIN
IF dept NOT IN ('D11','D21','ALL') THEN
SIGNAL SQLSTATE '75001' SET MESSAGE_TEXT = 'Invalid department input';
END IF;
CASE WHEN dept='D11' THEN
DECLARE Global Temporary Table freeemp_results AS
(SELECT distinct firstname as fname, lastname as lname, workdept as deptnum
FROM employee e
WHERE workdept='D11'
AND (CURRENT DATE > ALL (SELECT EMP_ENDDATE FROM emp_activeprojects p
WHERE e.empno=p.empno AND emp_enddate IS NOT NULL)
AND e.empno NOT IN (SELECT empno FROM emp_activeprojects WHERE emp_enddate is null))
WITH DATA WITH REPLACE ON COMMIT DELETE ROWS ;
END CASE;
WHEN dept='D21' THEN
DECLARE Global Temporary Table freeemp_results AS
(SELECT distinct firstname as fname, lastname as lname, workdept as deptnum
FROM employee e
WHERE workdept='D21'
AND (CURRENT DATE > ALL (SELECT EMP_ENDDATE FROM emp_activeprojects p
WHERE e.empno=p.empno AND emp_enddate IS NOT NULL)
AND e.empno NOT IN (SELECT empno FROM emp_activeprojects WHERE emp_enddate is null))
WITH DATA WITH REPLACE ON COMMIT DELETE ROWS ;
END CASE;
ELSE /* They want to search all departments */
DECLARE Global Temporary Table freeemp_results AS
(SELECT distinct firstname as fname, lastname as lname, workdept as deptnum
FROM employee e
WHERE (CURRENT DATE > ALL (SELECT EMP_ENDDATE FROM emp_activeprojects p
WHERE e.empno=p.empno AND emp_enddate IS NOT NULL)
AND e.empno NOT IN (SELECT empno FROM emp_activeprojects WHERE emp_enddate is null))
WITH DATA WITH REPLACE ON COMMIT DELETE ROWS ;
END CASE;
END;
```
In the modified codes of Get_Free_Employees_G procedure in Example 5-2, you notice that:

1. Compared with Example 5-1, the RESULT SETS clause in the procedure header is no longer needed.

2. The cursor declarations and open statement of the original Get_Free_Employees procedure are replaced with the DECLARE GLOBAL TEMPORARY TABLE statement. The temporary table named FreeEmp_Results is created in SESSION (the QTEMP library) by DB2 Universal Database for iSeries using the values in the SELECT list to determine column definitions of the temporary table. Similar to the CREATE TABLE statement, the DECLARE GLOBAL TEMPORARY TABLE statement allows column definitions to be individually defined or duplicated from another table or viewable with the addition of the AS or LIKE clause.

3. The WITH DATA clause directs DB2 to run the defined query and place the results of that query into the Global Temporary Table.

4. There are multiple versions of this DECLARE GLOBAL TEMPORARY TABLE statement because it cannot contain host variable references. Therefore, the different department values (D11, D21, ALL) had to be hardcoded into different versions of this statement.

The DECLARE GLOBAL TEMPORARY TABLE statement can also be dynamically prepared. You can have another option of building the DECLARE... statement dynamically and then using the PREPARE and EXECUTE statements to run it. This dynamic approach is demonstrated in Example 5-3, so you can compare the coding styles and see what works best for your applications.

Example 5-3  Get_Free_Employees_D procedure that uses dynamic declaration

```sql
CREATE PROCEDURE Get_Free_Employees_D (IN dept CHAR(3))
LANGUAGE SQL
BEGIN
DECLARE s1 VARCHAR(1000);
DECLARE deptsearch VARCHAR(24);

IF dept NOT IN ('D11','D21','ALL') THEN
  SIGNAL SQLSTATE '75001' SET MESSAGE_TEXT = 'Invalid department input';
END IF;

CASE
  WHEN dept='D11' THEN
    SET deptsearch = ' WORKDEPT=''D11''  AND ';
  WHEN dept='D21' THEN
    SET deptsearch = ' WORKDEPT=''D21''  AND';
  ELSE
    SET deptsearch = ' ';
END CASE;

SET s1 = 'DECLARE Global Temporary Table freeemp_results AS (SELECT distinct firstname as fname,lastname as lname,workdept as deptnum FROM employee e WHERE ' || deptsearch || ' (CURRENT DATE > ALL (SELECT EMP_ENDDATE FROM emp_activeprojects p WHERE e.empno=p.empno AND emp_enddate IS NOT NULL) AND empno NOT IN (SELECT empno FROM emp_activeprojects WHERE emp_enddate is null)) ) WITH DATA WITH REPLACE ON COMMIT DELETE ROWS';
```
5.6.3 Accessing a result set from a Global Temporary Table

Now we look at the logic needed in the invoking SQL stored procedure to access the result set data contained in a Global Temporary Table. The Staff_Project procedure shown in Example 5-4 calls the Get_Free_Employees_D procedure of Example 5-3 to find all employees who are currently free from project engagements.

Example 5-4 Staff_Project procedure that calls Get_Free_Employees_D procedure

```sql
CREATE PROCEDURE staff_project
(IN start_date date, IN requestor_name CHAR(30), OUT empct INT)
LANGUAGE SQL
SET OPTION DBGVIEW=*SOURCE
sp: BEGIN
   DECLARE emp_name CHAR(30);
   DECLARE mgr_name CHAR(30);

   IF (start_date < current date) THEN
      SIGNAL SQLSTATE VALUE '75002' SET MESSAGE_TEXT = 'Invalid start date input';
   END IF;

   -- Search all the departments for employee's just wrapping up from a project
   CALL Get_Free_Employees_D('ALL');
   SET empct=0;

   FOR loop_var AS free_list CURSOR FOR
      SELECT fname,lname,deptnum FROM session.freeemp_results
   DO
      SET emp_name = fname|| ' ' || lname;
      SET mgr_name = (SELECT fname|| ' ' || lname
                      FROM employee WHERE job='MANAGER' AND workdept=deptnum);

      CALL send_enquiry_email(requestor_name, emp_name, mgr_name, start_date);
      SET empct=empct+1;
   END FOR;
END sp;
```

From Example 5-4, the Staff_Project procedure knows (from the design step) that the results of its inquiry are to be stored in a Global Temporary Table called FreeEmp_Results. To process the returned list of employees in the FreeEmp_Results table, the FOR statement is used to open a cursor for the temporary table. Notice that the Global Temporary Table name must be qualified with the syntax SESSION so that DB2 can locate the declared Global Temporary Table returned from the invocation of the Get_Free_Employees_D stored procedure.

Important: If the Staff_Project procedure is called by two different database connections (or jobs) simultaneously, then each connection has its own unique instance of FreeEmp_Results Global Temporary Table to store the list of free employees. If any non-SQL program (in the same job) must access this FreeEmp_Results temporary table, normal native read and write requests can be performed against this table since a Global Temporary Table is considered a regular DB2 table (physical file) object. Therefore, a Global Temporary Table is global to all program codes within a database connection or job.
5.7 GetSuppliers example

The Order Entry database contains the ORDERHDR table. The primary key of the ORDERHDR table is the ORHNBR (order number) column. For every order number in the ORDERHDR table, there are one or more rows in the ORDERDTL table, for the same order number. The ORDERDTL table contains the order number (ORHNBR) of the order already present in the ORDERHDR table and a product number (PRDNBR) that identifies the products to be supplied for that order. Every product has a name, price, and a supplier number. The product details are in the STOCK table. Every supplier has a supplier name, supplier number, and the supplier address. The supplier details are in the SUPPLIER table. Refer to 2.2, “Order Entry database overview” on page 13, for details on the database layout.

Now suppose that you want to find \( n \) suppliers with the highest sales in a given year and month. At the same time, you also want to retrieve \( n \) suppliers with the lowest sales in a given year and month. You can implement this business logic coding an SQL stored procedure that returns multiple result sets.

You can pass three parameters: Year, month, and rank. If the value of the rank parameter is 10, for example, the stored procedure will return a list of ten suppliers with the highest sales in a given month and year as the first result set. You also see a list of ten suppliers with the lowest sales in a given month and year as the second result set. The first two parameters, year and month, are input (IN) parameters. The third parameter, rank, is an input/output (INOUT) parameter. On the stored procedure invocation, it contains the number of suppliers to be returned in the two result sets. On the return, the stored procedure sets this parameter to a value that indicates the actual number of rows available in the result sets (may be less than the requested number). If the month is not specified, the program returns two supplier lists (the best and the worst) for the whole year, rather than for a given month in a year.

To implement this scenario, we used three views: SALE, TOTALSALE, and YEARSALE created on the ORDERHDR, SUPPLIER, and the STOCK tables. The SQL scripts needed to create the views are available for download. Refer to “Locating the Web material” on page 561 for more details.

5.7.1 Creating the SQL stored procedure

The CREATE PROCEDURE statement in Example 5-5 creates the SQL stored procedure, which returns two result sets.

Example 5-5 CREATE PROCEDURE

```
CREATE PROCEDURE GETRANK
  (IN proc_year DECIMAL(4,0),
   IN proc_month DECIMAL(2,0),
   INOUT proc_rank INTEGER)
  RESULT SETS 2
  LANGUAGE SQL
BEGIN
  DECLARE highsales DECIMAL(11,2);
  DECLARE lowsales DECIMAL(11,2);
  DECLARE rank1 INTEGER;
  DECLARE rank2 INTEGER;
  DECLARE SQLStmt CHAR(512);
  DECLARE c1  DYNAMIC SCROLL CURSOR WITH RETURN FOR s1;
  DECLARE c2  DYNAMIC SCROLL CURSOR WITH RETURN FOR s2;

  --- Check if it is Null for Month
  IF proc_month IS NULL or proc_month = 0 THEN
```
--Get Highest Rank Suppliers
SET SQLStmt='SELECT yearsales FROM yearsale WHERE year=?
    ORDER BY yearsales DESC';
PREPARE s1 FROM SQLStmt;
OPEN c1 USING proc_year;
FETCH RELATIVE proc_rank FROM c1 INTO highsales;
IF highsales IS NULL THEN
    FETCH LAST FROM c1 INTO highsales;
END IF;
CLOSE c1;
SET SQLStmt='SELECT count(*) FROM yearsale WHERE year=? AND yearsales>=?';
PREPARE s1 FROM SQLStmt;
OPEN c1 USING proc_year, highsales;
FETCH c1 INTO rank1;
CLOSE c1;
SET SQLStmt='SELECT supplier_name,yearsales FROM yearsale WHERE year=?
    AND yearsales>=? ORDER BY yearsales DESC';
PREPARE s1 FROM SQLStmt;
OPEN c1 USING proc_year, highsales;
--Get Lowest Rank Suppliers
SET SQLStmt='SELECT yearsales FROM yearsale WHERE year=?
ORDER by yearsales ASC';
PREPARE s2 FROM SQLStmt;
OPEN c2 USING proc_year;
FETCH RELATIVE proc_rank FROM c2 INTO lowsales;
IF lowsales IS NULL THEN
    FETCH LAST FROM c2 INTO lowsales;
END IF;
CLOSE c2;
SET SQLStmt='SELECT count(*) FROM yearsale WHERE year=? AND yearsales<=?';
PREPARE s2 FROM SQLStmt;
OPEN c2 USING proc_year, lowsales;
FETCH c2 INTO rank2;
CLOSE c2;
SET SQLStmt='SELECT supplier_name,yearsales FROM yearsale WHERE year=?
    AND yearsales<=? ORDER BY yearsales ASC';
PREPARE s2 FROM SQLStmt;
OPEN c2 USING proc_year, lowsales;
ELSE
    --Get Highest Rank Suppliers
    SET SQLStmt='SELECT totalsales FROM TOTALSALE WHERE year=?
    AND month=? ORDER BY totalsales DESC';
    PREPARE s1 FROM SQLStmt;
    OPEN c1 USING proc_year, proc_month;
    FETCH RELATIVE proc_rank FROM c1 INTO highsales;
    IF highsales IS NULL THEN
        FETCH LAST FROM c1 INTO highsales;
    END IF;
    CLOSE c1;
    SET SQLStmt='SELECT count(*) FROM totalsale WHERE year=? AND month=?
    AND totalsales>=?';
    PREPARE s1 FROM SQLStmt;
    OPEN c1 USING proc_year, proc_month, highsales;
    FETCH c1 INTO rank1;
    CLOSE c1;
    SET SQLStmt='SELECT supplier_name,totalsales FROM totalsale WHERE year=?
    AND month=? AND totalsales>=?
ORDER BY totalsales DESC';
    PREPARE s1 FROM SQLStmt;
ELSE
    --Get Highest Rank Suppliers
    SET SQLStmt='SELECT totalsales FROM TOTALSALE WHERE year=?
    AND month=? ORDER BY totalsales DESC';
    PREPARE s1 FROM SQLStmt;
    OPEN c1 USING proc_year, proc_month;
    FETCH RELATIVE proc_rank FROM c1 INTO highsales;
    IF highsales IS NULL THEN
        FETCH LAST FROM c1 INTO highsales;
    END IF;
    CLOSE c1;
    SET SQLStmt='SELECT count(*) FROM totalsale WHERE year=? AND month=?
    AND totalsales>=?';
    PREPARE s1 FROM SQLStmt;
    OPEN c1 USING proc_year, proc_month, highsales;
    FETCH c1 INTO rank1;
    CLOSE c1;
    SET SQLStmt='SELECT supplier_name,totalsales FROM totalsale WHERE year=?
    AND month=? AND totalsales>=?
ORDER BY totalsales DESC';
    PREPARE s1 FROM SQLStmt;
OPEN c1 USING proc_year, proc_month, highsales;
--Get Lowest Rank Suppliers
SET SQLStmt='SELECT totalsales FROM totalsale WHERE year=? AND month=?
ORDER by totalsales ASC';
PREPARE s2 FROM SQLStmt;
OPEN c2 USING proc_year, proc_month;
FETCH RELATIVE proc_rank FROM c2 INTO lowsales;
IF lowsales IS NULL THEN
    FETCH LAST FROM c2 INTO lowsales;
END IF;
CLOSE c2;
SET SQLStmt='SELECT count(*) FROM totalsale WHERE year=? AND month=?
AND totalsales<=?';
PREPARE s2 FROM SQLStmt;
OPEN c2 USING proc_year,proc_month, lowsales;
FETCH c2 INTO rank2;
CLOSE c2;
SET SQLStmt='SELECT supplier_name,totalsales FROM totalsale WHERE year=?
AND month=? AND totalsales<=?
ORDER BY totalsales ASC';
PREPARE s2 FROM SQLStmt;
OPEN c2 USING proc_year,proc_month, lowsales;
END IF;
IF rank1< rank2 THEN
    SET proc_rank=rank2;
ELSE
    SET proc_rank=rank1;
END IF;
END

CREATE PROCEDURE statement explanation
The following notes refer to Example 5-5.

1. We do not qualify the procedure name. Special care must be taken, because depending on
the naming convention, the behavior can change. If we are working with the "SQL naming
convention, the procedure will be located in the implicit or explicit qualifier. If we are
working with the "SYS naming convention, it will be located in the current library or explicit
qualifier. The procedure accepts two input parameters and one INOUT parameter.

2. This is the number of result sets returned from the procedure.

3. This specifies that this is an SQL procedure.

Now let us examine the body of this SQL stored procedure:

4. Starting on V5R1, result sets are specified by declaring cursors using the WITH RETURN
option. Multiple result sets can be returned. Notice also that cursors c1 and c2 will remain
open at the end of the procedure.

5. The first SELECT statement is used to find the totalsales value for the n-th relative supplier.
The second SELECT statement uses the retrieved totalsales value to select the desired
result set. We ORDER BY total sales in descending (ascending) order. Therefore, we have
the highest (lowest) total sales value in the first row of the resultant table when the cursor is
opened.

6. To use FETCH RELATIVE, you have to declare the cursor as scrollable.

7. If the FETCH RELATIVE moves the cursor to an out-of-range position, the lowsales
variable is not modified. In this example, the value of the lowsales variable before the
FETCH statement executes is NULL. Therefore, we can use the IS NULL statement to
check whether the FETCH was successful.
Before V5R1, the WITH RETURN syntax for returning result sets were not supported. An equivalent SQL stored procedure looks like Example 5-6.

**Example 5-6  Equivalent SQL stored procedure for WITH RETURN**

```sql
CREATE PROCEDURE GETRANKV4R5
  (IN proc_year DECIMAL(4,0),
   IN proc_month DECIMAL(2,0),
   INOUT proc_rank INTEGER)
RESULT SETS 2
LANGUAGE SQL
BEGIN
  ...
  DECLARE c1  DYNAMIC SCROLL CURSOR FOR s1;
  DECLARE c2  DYNAMIC SCROLL CURSOR FOR s2;
  ...
  SET RESULT SETS CURSOR c1, CURSOR c2;
END
```

The preferred syntax is to use WITH RETURNS because it is more portable across different platforms.

### 5.7.2 Displaying the result sets with iSeries Navigator

If a stored procedure does not use the INOUT, you can test it using the Run SQL Scripts utility. In the getRank example described in the previous section, we defined one INOUT parameter, which now is modified to a couple of parameters: one as INPUT and the other as OUTPUT. Now if a stored procedure returns a result set, the utility displays the result set data in a spreadsheet format, as shown in Figure 5-23.

![Figure 5-23 Using the SQL Script utility to retrieve result sets](image)
5.7.3 Calling an SQL procedure from the Visual Basic client application

This section introduces a Visual Basic application, which can invoke the SQL stored procedure described in the previous section. You may also use this client program to invoke other implementations of the subject procedure documented in Chapter 6, “External stored procedures” on page 123, and Chapter 7, “Java stored procedures” on page 173. The complete program source of the client is available for download from the Web. Refer to “Locating the Web material” on page 561 for details.

The client reads the year, month, and required rank from the text boxes. The Query button is used to invoke a procedure implementation named in the txtProcedure text box. The highest sales suppliers’ result set and the lowest sales suppliers’ result set are assigned to two Flexgrids, as shown in Figure 5-24.

See Example 5-7.

Example 5-7 Invoking the procedure implementation

Private Sub cmdStoredProc_Click()
    On Error GoTo ResolveError
    Set cmd = New ADODB.Command
    Call clearFlexGrid
    cmd.ActiveConnection = Cn
    cmd.CommandText = txtProcedure
    cmd.CommandType = adCmdStoredProc
    'Store Procedure Input Parameter
    Set prm1 = cmd.CreateParameter("proc_year", adDecimal, adParamInput, 4, Val(txtYear.Text))
    cmd.Parameters.Append prm1
    'Store Procedure Input Parameter
    If txtMonth.Text = "" Then
        Set prm2 = cmd.CreateParameter("proc_month", adDecimal, adParamInput, 2, Null)
    Else
        Set prm2 = cmd.CreateParameter("proc_month", adDecimal, adParamInput, 2, Val(txtMonth.Text))
    End If
    cmd.Parameters.Append prm2
    'Store Procedure Output Parameter

Figure 5-24 Visual Basic application retrieving multiple result sets
Set prm3 = cmd.CreateParameter("proc_rank", adInteger, adParamInputOutput, , Val(txtRank.Text))
cmd.Parameters.Append prm3
'cmd.Execute
If OptYes.Value Then
    Set Rs.Source = cmd
    Rs.Open
    MSHFlexGrid1.Rows = Rs.RecordCount + 1
    MSHFlexGrid1.Cols = Rs.Fields.Count
    'Set column names in the grid
    For i = 0 To Rs.Fields.Count - 1
        MSHFlexGrid1.TextMatrix(0, i) = Rs.Fields(i).Name
    Next
    Set MSHFlexGrid1.DataSource = Rs
    Rs.Close
    txtRank.Text = CStr(cmd.Parameters(2))
Else
    Set prm4 = cmd.CreateParameter("suppliers", adVarChar, adParamOutput, 1000)
cmd.Execute
    strSupplier = CStr(cmd.Parameters(3))
txtRank.Text = CStr(cmd.Parameters(2))
    MSHFlexGrid1.Rows = Val(txtRank.Text)
    MSHFlexGrid2.Rows = Val(txtRank.Text)
    For i = 0 To Val(txtRank.Text) - 1
        For j = 0 To 1
            intDelPos = FindDelimitor(strSupplier)
            MSHFlexGrid1.TextMatrix(i, j) = Left(strSupplier, intDelPos - 1)
            strSupplier = Right(strSupplier, Len(strSupplier) - intDelPos)
        Next
    Next
    For i = 0 To Val(txtRank.Text) - 1
        For j = 0 To 1
            intDelPos = FindDelimitor(strSupplier)
            MSHFlexGrid2.TextMatrix(i, j) = Left(strSupplier, intDelPos - 1)
            strSupplier = Right(strSupplier, Len(strSupplier) - intDelPos)
        Next
    Next
End If
Exit Sub
ResolveError:
    Call ResolveError(Cn)
End Sub

Private Sub Form_Unload(Cancel As Integer)
    If cmdDisconnect.Enabled Then Call cmdDisconnect_Click
End Sub

Function ResolveError(ErrCn As ADODB.Connection)
    Dim strErrMsg As String
    Dim i As Integer
For i = 0 To ErrCn.Errors.Count - 1
    strErrMsg = "Collection Element " + CStr(i) + vbCrLf
    strErrMsg = strErrMsg & "Error Number" & CStr(ErrCn.Errors(i).NativeError) & vbCrLf
    strErrMsg = strErrMsg & "Source: " & ErrCn.Errors(i).Source & vbCrLf
    strErrMsg = strErrMsg & ErrCn.Errors(i).Description & vbCrLf
    strErrMsg = strErrMsg & "SQLSTAT: " & ErrCn.Errors(i).SQLState
    MsgBox strErrMsg
Next i
End Function

**Notes:** The following notes refer to Example 5-7:

1. You can specify the name of the target stored procedure and the library using the SQL naming convention library.proc_name. The library name is optional.
2. To invoke a stored procedure, set the CommandType property to adCmdProc.
3. The parameter name and the parameter type should be declared as in the target stored procedure.
4. For the SQL style parameter type, pass a NULL value to the target stored procedure.
5. The FlexGrid Visual Basic control can implement a result set of various row and column sizes with a few lines of code.
6. This is a general error handling routine. You may have a more specific routine to handle a specific SQLSTATE.

### 5.8 Implicit object qualification and authorization resolution

In this section, we discuss the default schema resolution mechanism and runtime authorization ID resolution of stored procedures.

#### 5.8.1 Implicit schema qualification for static and dynamic SQL in procedures

In DB2 Universal Database for iSeries, the unqualified alias, constraint, external program, index, node group, package, sequence, table, trigger, and view names are implicitly qualified by the default schema. Before V5R2, there were no compile-time options to indicate a specific value of a default schema for any object name in the program codes. The system used the rules listed in Table 5-1 to resolve unqualified names in an SQL stored procedure.

**Table 5-1 Default schema resolution for unqualified object in OS/400 PSM**

<table>
<thead>
<tr>
<th>Default schema resolution method</th>
<th>SQL naming (&quot;SQL&quot;)</th>
<th>System naming (&quot;SYS&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static SQL</td>
<td>Set to the authorization ID (user profile name) in effect when the stored procedure is <em>created</em></td>
<td>Set to the job's library list (<em>LIBL</em>)</td>
</tr>
<tr>
<td>Dynamic SQL</td>
<td>Set to the authorization ID (user profile name) in effect when the stored procedure is <em>executed</em></td>
<td>Same as above</td>
</tr>
</tbody>
</table>
To illustrate the default OS/400 behavior, consider the simple stored procedure shown in Example 5-8.

**Example 5-8**  A sample CREATE PROCEDURE for TestDynSQL() procedure

```sql
CREATE PROCEDURE TestDynSQL ( )
LANGUAGE SQL
SPECIFIC testdynsql
MODIFIES SQL DATA
MainBody: BEGIN
DECLARE STMT VARCHAR ( 128 ) ;
DECLARE SQLERRM VARCHAR ( 256 ) DEFAULT '';
DECLARE ErrorIndicator CHAR(1) DEFAULT 'N';

Static_Stmt: BEGIN
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION 2
BEGIN
GET DIAGNOSTICS EXCEPTION 1 SQLERRM = MESSAGE_TEXT ;
INSERT INTO jm_debug ( SQLTEXT )
VALUES ( 'Static Statement Failed: ' || SQLERRM ) WITH NC;
SET ErrorIndicator = 'Y';
END;
INSERT INTO jm_debug VALUES ( 'Static Statement Succeeded.' );
END Static_Stmt;

Dynamic_Statement: BEGIN
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION 4
BEGIN
GET DIAGNOSTICS EXCEPTION 1 SQLERRM = MESSAGE_TEXT ;
INSERT INTO jm_debug ( SQLTEXT )
VALUES ( 'Dynamic Statement Failed: ' || SQLERRM ) WITH NC;
SET ErrorIndicator = 'Y';
END;
SET STMT = 'INSERT INTO jm_debug VALUES(''Dynamic Statement Succeeded.'');
PREPARE S1 FROM STMT ;
EXECUTE S1;
END Dynamic_Stmt;

IF ErrorIndicator = 'Y' THEN
SIGNAL SQLSTATE '70000'  SET
message_text='There were errors. Check jm_debug for details.';
END IF;
END MainBody;
```

**Note:** The JM_DEBUG table used in Example 5-8 is a simple table that contains the following columns:
- SQLTEXT, data type = CHAR, data length = 384
- T1, data type = timestamp

The sample stored procedure inserts two rows into the JM_DEBUG table:
- At line 1 a static INSERT is used. Since the table name is unqualified, the task to resolve it to a table object rests on OS/400. The errors in the Static_Stmt compound statement are intercepted and handled by the continue handler at 2.
- A dynamic INSERT at line 3 is used to insert the second row. The errors are handled by the continue handler in the Dynamic_Stmt compound statement at line 4.

How does DB2 resolve the unqualified reference to the JM_DEBUG table if no default schema is set? Consider the scenario in the following section.
Test scenario 1

In this scenario, we presume the following points:

- The CREATE PROCEDURE statement of Example 5-8 is run under the authorization ID (user profile) DB2ADMIN.
- The JM_DEBUG table resides in the DB2ADMIN schema.
- The naming convention in effect is *SQL.

Another user named DB2GURU invokes the stored procedure. For simplicity, we can assume that the user DB2GURU has all object (*ALLOBJ) authority. Before the stored procedure call is executed, DB2GURU sets the PATH to DB2ADMIN so that the stored procedure can be called:

```sql
SET PATH = DB2ADMIN;
CALL TESTDYNSQL();
```

The stored procedure returns the following error messages:

- SQL State: 70000
- Vendor Code: -438
- Message: [SQL0438] There were errors. Check jm_debug for details.

The JM_DEBUG contains the entries shown in Figure 5-25.

![Debug messages for test scenario 1](image)

As expected, the static statement succeeds, because DB2 Universal Database for iSeries uses the authorization ID (user profile DB2ADMIN) in effect at creation time of the procedure to resolve the unqualified JM_DEBUG table name. However, the dynamic statement fails. This time, the runtime authorization ID (user profile DB2GURU) is used to resolve the unqualified name. Since no JM_DEBUG table is in a schema named DB2GURU, DB2 signals an SQL exception.

To eliminate the error, DB2GURU can use the SET SCHEMA statement (new in V5R2) to set a default schema for all the dynamic statements that run in a given session:

```sql
SET SCHEMA = DB2ADMIN;
```

Another approach is to hardcode the SET SCHEMA statement in the procedure. In some environments, you may want to determine the default schema for both static and dynamic statements at runtime rather than hardcoding it in the application code or runtime settings. This requirement becomes especially important for applications that support heterogeneous databases. For example, with Oracle, unqualified names in both static and dynamic statements are, by default, resolved to the creator's authorization ID, which is different from DB2 Universal Database for iSeries.

In V5R2, through a set of PTFs, SQL stored procedure support on iSeries was enhanced to accommodate for implicit qualification. In the CREATE PROCEDURE statement, you can use the two new options that were added to the SET OPTION statement:

- Default collection (DFTRDBCOL) specifies the name of the schema identifier used for the unqualified names of the tables, views, indexes, and SQL packages. This parameter applies only to static SQL statements.
Dynamic default collection (DYNDFTCOL) specifies whether the default schema name specified for the DFTRDBCOL parameter is also used for dynamic statements.

By default, these two attributes are set to the following values:

- DFTRDBCOL(*NONE)
- DYNDFTCOL(*NO)

These default values correspond to the default DB2 behavior described in this scenario. Specifically, DFTRDBCOL(*NONE) means that no default schema is in effect. Therefore, the rules outlined in Table 5-1 on page 117 are in effect.

Now consider the use of the SET OPTION statement as explained in the following scenario.

**Test scenario 2**

In the header of Example 5-8 on page 118, you can add the following SET OPTION statement to the CREATE PROCEDURE statement as shown in Example 5-9.

**Example 5-9**  CREATE PROCEDURE for TestDynSQL() procedure with DYNDFTCOL

```sql
CREATE PROCEDURE TestDynSQL ( )
    LANGUAGE SQL
    SPECIFIC testdynsql
    MODIFIES SQL DATA
    SET OPTION DYNDFTCOL=*YES
BEGIN
    ...
END
```

To avoid hardcoding, you should deliberately omit the DFTRDBCOL keyword in the procedure header. The DFTRDBCOL can be inherited from the current environment. At runtime, when the target default schema is determined, the user DB2ADMIN sets the default schema to, for example, DB2ADMIN, and executes the script that creates the stored procedure. As a result, the program object associated with the stored procedure has the attributes DFTRDBCOL=DB2ADMIN and DYNDFTCOL=*YES. This means that, at runtime, both static and dynamic unqualified statements will be implicitly qualified to the DB2ADMIN schema.

How does the modified stored procedure work? After opening a new connection, DB2GURU calls the stored procedure:

```
CALL TESTDYNSQL();
```

This time, there is no need to use SET SCHEMA or to change the default schema setting in the connection environment. The stored procedure completes successfully. The JM_DEBUG table contains the message entries shown in Figure 5-26.

<table>
<thead>
<tr>
<th>SQLTEXT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Statement Succeeded</td>
<td></td>
</tr>
<tr>
<td>Dynamic Statement Succeeded</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5-26**  Debug messages for test scenario 2
5.8.2 Dynamic resolution of authorization ID at runtime

Sometimes, even experienced iSeries developers tend to confuse the ability to resolve unqualified names to database objects with authorization required to access them. As described in this section, the implicit qualification is controlled by DFTRDBCOL and DYNDFTCOL, while access authority is managed by USRPRF and DYNUSRPRF attributes.

- **User profile** (USRPRF) specifies the user profile that is used when the compiled routine (or SQL package) is run, including the authority to each object in static SQL statements. The profile of either the *OWNER or the *USER is used to control access to objects.
- **Dynamic user profile** (DYNUSRPRF) specifies the user profile used for dynamic SQL statements. The profile of either the program's user or the program's owner is used to control access to objects.

By default, these parameters are set to USRPRF(*NAMING) and DYNUSRPRF(*USER).

We use the "SQL naming convention in “Test scenario 2" on page 120, which means that the *OWNER profile (DB2ADMIN) was used for the static statement and the *USER profile (DB2GURU) was used for the dynamic statement. Test scenario 2 runs successfully only because DB2GURU had *ALLOBJ authority as we assumed earlier. Typically, application users run under user profiles with limited authority. Here, DYNUSRPRF can be convenient.

We can specify DYNUSRPRF(*OWNER) on the CREATE PROCEDURE statement so that the dynamic statement also runs under the *OWNER user profile (DB2ADMIN). By using this method, we can ensure that the statement will not fail due to a lack of access authority.

Based on the procedure header shown in Example 5-9, Example 5-10 shows the modified CREATE PROCEDURE statement that takes care of both implicit qualification and access authority.

**Example 5-10  CREATE PROCEDURE for TestDynSQL() procedure with DYNDFTCOL, DYNUSRPRF**

```
CREATE PROCEDURE TestDynSQL ( )
  LANGUAGE SQL
  SPECIFIC testdynsql
  MODIFIES SQL DATA
  SET OPTION DYNDFTCOL=*YES, DYNUSRPRF=*OWNER
BEGIN
  ...
END
```

You must not forget to grant the execute privilege on the stored procedure to the user who needs to invoke it, for example:

```
GRANT EXECUTE ON PROCEDURE testdynsql TO db2usr;
```

In addition, user DB2USR needs at least *USE authority to the library (schema) that contains the stored procedure’s *PGM object. Here’s a sample CL command:

```
GRTOBJAUT OBJ(DB2ADMIN) OBJTYPE(*LIB) USER(DB2USR) AUT(*USE)
```
Chapter 6. External stored procedures

Stored procedures can be written in two ways on DB2 Universal Database for iSeries. One approach is described as *SQL stored procedures*. This is based on procedural extensions to the SQL language, highly used by other DBMS providers, as described in Chapter 5, “SQL stored procedures” on page 81. The other approach is based on high-level languages that you are familiar with, such as C, CL, RPG, COBOL, and so on. They are described as *external stored procedures*.

External stored procedures are coded in one of the high-level languages available on the iSeries server. If you want to perform complex sophisticated processing, or plan to re-use code that already exists, external stored procedures are the best choice for you.

This chapter describes external stored procedures written in high-level languages other than Java. It explains how to register and code external store procedures. It also reviews the difference in coding external store procedures regarding the different parameter styling supported by DB2 Universal Database for iSeries and how to invoke external stored procedures and deal with error handling.

External stored procedures can also be written in Java. Due to intrinsic differences in this approach, they are described as *Java stored procedures*, which are discussed in Chapter 7, “Java stored procedures” on page 173.

All the benefits of the stored procedures discussed in Chapter 4, “Stored procedures” on page 63, also apply to external stored procedures.
6.1 Registering external stored procedures

Before using an external stored procedure, it must be registered within the database. You can use the CREATE PROCEDURE statement or iSeries Navigator to register an external stored procedure. When an external stored procedure is registered with the database, entries are made into the system catalog tables. These tables store information about every routine (procedure or function) that is registered with the database. The information that is recorded in these tables is discussed in 4.4, “System catalog tables” on page 74.

When you register an external stored procedure, you should specify the name of the procedure, the number of parameters, and the data type and length of the parameters. In most cases, you also specify the input/output type of the parameter and the parameter passing style. This chapter discusses the different parameter passing styles in 6.2, “Parameter styles in external stored procedures” on page 129. Apart from accepting input parameters and returning output parameter values, a stored procedure can return a number of rows to the calling program in the form of a result set. The external program can implement the result set as an array of values, or it can open an SQL cursor and return it as a result set. This chapter discusses the different coding techniques for result sets in 6.4, “Returning result sets from external procedures” on page 144.

The external program that is executed when the external stored procedure is called by the CALL statement should be a *PGM object compiled with the Activation Group parameter *CALLER. The external program can contain host language statements and SQL statements. Note that a service program cannot be registered as an external stored procedure.

Note: Some iSeries interfaces, such as STRSQL and iSeries Navigator, allow programs to be called without having the program to be registered, but to be on the safe side we recommend that you always register the programs as external stored procedures.

An external stored procedure can be called by an application program running on the same iSeries server, where the stored procedure resides. Or it can be called across the network by a client program. The client program may run on a workstation and communicate with the server through programming interfaces such as ODBC, ADO, JDBC, and SQLJ. It may also run on another server machine and communicate through DRDA.

Examples of client programs that can call the external stored procedure are discussed in 6.5, “CLI client program that calls a procedure that returns multiple result sets” on page 151.

6.1.1 Registering an external procedure with iSeries Navigator

As an example, we discuss the creation of the High_Sales external stored procedure. This procedure accepts year of type INTEGER and month of type INTEGER as input parameters. It returns Supplier_Name of type CHAR(20) and H_Sales of type DECIMAL(11,2). The returned values contain data for the supplier with the highest total sales in a given year and month. The parameter passing style used is GENERAL. These input and output parameters are based on the columns of the TOTALSALE view. Refer to 2.2, “Order Entry database overview” on page 13, for the detailed structure of this view.
This section shows how to create an external stored procedure using the iSeries Navigator’s New External Procedure window. The required steps are listed here:

1. In iSeries Navigator, expand **Databases** and the **database folder** for which the external procedure is going to be registered. Expand the **Libraries** object. You see all the libraries in your library list. Right-click the library in which you want to create the external stored procedure, and select **New -> Procedure -> External** as shown in Figure 6-1.

![Figure 6-1 Creating an external stored procedure using the Create procedure window](image-url)
2. The New External Procedure window (Figure 6-2) opens in which you perform the following actions:

   a. On the **General** tab, type the name of the procedure, a description, and the specific name of the external stored procedure. If you do not enter the specific name, it defaults to the name of the procedure. The specific name is used by the database manager to uniquely identify a stored procedure within a library.

   Other parameters may be defined at this time. For example, you may define whether an automatic commit should be performed when the stored procedure returns control to the callers and if the stored procedure should run at an inner savepoint level, and the maximum number of result sets the stored procedure is going to return.

   ![Figure 6-2 New external procedure](image)
b. Click the **Parameters** tab.
   i. Click the **Insert** button. Type the name of the parameter.
   ii. Choose the data type of the parameter from the pull-down menu. Enter the length of the parameter if it is required. Select the input/output type of the parameter as shown in Figure 6-3.
   iii. Choose the parameter style. Click **Simple, no null values allowed**. If you do not choose the parameter style, the default value is SQL.

![New External Procedure in ORDAPI1B - Monaco (Monaco)](image)

**Figure 6-3  Defining input/output parameters**
c. Click the **External Program** tab.

Type the name of the external program that should be executed when this external stored procedure is called using an SQL CALL statement. If you do not enter the external program name, the default value is the name of the external stored procedure. Choose the library name and the language of the external program, as shown in Figure 6-4. If you leave the language field empty, the system tries to guess the implementation language. The default value for this field is ILE C.

d. Click the **OK** button to register the stored procedure.

![New External Procedure in ORDAPPLIB - Monaco(Monaco)](image)

*Figure 6-4  External program name, library, and language*

The corresponding SQL CREATE PROCEDURE statement is shown here:

```sql
CREATE PROCEDURE ORDAPPLIB.Hsales(
  IN Year INTEGER,
  IN Month INTEGER,
  OUT Supplier_Name CHAR(20),
  OUT Hsale DECIMAL(11,2)
) LANGUAGE RPGLE EXTERNAL NAME ORDAPPLIB.HSALES MODIFIES SQL DATA PARAMETER STYLE GENERAL
```
After successful completion of the window, refresh the contents of the ORDAPPLIB library by clicking the **Refresh** icon in the toolbar. You should now see the Hsale stored procedure icon in the list of the objects, as shown in Figure 6-5.

![Figure 6-5   A new stored procedure](image)

### 6.2 Parameter styles in external stored procedures

You can specify several different parameter styles for an external stored procedure. On the invocation of the external stored procedure, DB2 Universal Database for iSeries passes a number of parameters to the procedure in addition to those specified on the parameter list. The number and type of additional parameters passed depends on the parameter style. You can specify the required parameter style when the procedure is created. DB2 Universal Database for iSeries supports four parameter styles:

- SQL parameter style
- DB2SQL parameter style
- GENERAL WITH NULLS parameter style
- GENERAL parameter style

Prior to V4R4, only general and GENERAL WITH NULLS parameter styles were supported. The SQL parameter style was added in V4R4, and the DB2SQL parameter style was added in V4R5. Now the SQL parameter style is the default parameter style in the DB2 Universal Database family for compatibility purposes.

This section discusses the number and type of parameters passed with each parameter style. Later in this chapter we provide examples for each of these parameter styles.
6.2.1 SQL parameter style

The list of parameters received by the external stored procedure, when the SQL parameter style is specified in the procedure definition, is shown here:

- IN | OUT | INOUT argument (repeated),
- INOUT argument indicator variables,
- OUT SQLSTATE,
- IN procedure name
- IN specific name
- OUT diagnostic message

The parameters are explained in the following list:

- **Arguments**: The input, output, and input/output parameters passed from the calling program to an external stored procedure. The order in which you specify the argument types (IN | OUT | INOUT) is not relevant.

- **Argument indicator**: The NULL indicator for each input argument and output argument. If a NULL value was passed for an argument, the corresponding indicator variable contains -1. If a valid value is passed, the indicator variable contains 0. The function can test the value of an argument indicator. Before using the input parameter in the external stored procedure, check the null indicator. If the corresponding argument contains a null, be sure to take corrective action. For every output parameter, there is a corresponding null indicator that is passed back to the calling program. In the calling program, you can check whether a null value has been returned in the output parameter.

- **SQLSTATE**: Output parameter, defined as CHAR(5), that corresponds to the SQLSTATE in SQL. This value is set by the external stored procedure, to signal a successful execution, warning, or error to the calling program. If the SQLSTATE is not set to any of the defined values shown in the following list, the calling program receives the SQLSTATE 39001, which indicates an invalid SQLSTATE. The external program can set this output parameter to one of the following values:
  - 00000: Successful execution, no errors.
  - 01Hxx: Warning. The trailing xx value is any two digits or uppercase letters. It results in SQLCODE 462 from SQL.
  - 38yxx: Error condition y can be any letter or number. The next two characters xx are any two-digit or uppercase letters to indicate the error. It results in SQLCODE -443 from SQL.

- **Procedure name**: A fully qualified procedure name. This is an input parameter, defined as VARCHAR(517).

- **Specific name**: The specific name of the function. This is an input parameter, defined as VARCHAR(128).

- **Diagnostic message**: The message text that can contain a customized error message. You can set the diagnostic message only when you set the SQLSTATE parameter. For system errors, such as record locked, referential constraint violation, and so on, it is set to the first 70 characters of the system message. This is an output parameter defined as VARCHAR(70).

**Note**: In V4R5, the diagnostic message is a character array that has the length in the first position. In V5R1, this changed and the message is a null terminated string.
6.2.2 DB2SQL parameter style

The DB2SQL style is a super-set of the SQL parameter style. When DBINFO is specified in the CREATE PROCEDURE, it indicates to DB2 Universal Database for iSeries to pass the DBINFO structure containing the following fields:

- Relational database name
- Authorization ID
- CCSID
- Version & Release
- Platform

If NO DBINFO is specified in the CREATE PROCEDURE, this style is equal to SQL parameter style. For more information about the parameters passed, see the include sqludf in the appropriate source file. For example, for C, sqludf can be found in QSYSINC/H.

6.2.3 GENERAL WITH NULLS parameter style

The list of parameters received by the external stored procedure for the GENERAL WITH NULLS parameter style is listed here:

IN | OUT | INOUT argument [repeated],
INOUT argument indicator variables,

The parameters are explained in the following list:

- Arguments: The input, output, and input/output (both) parameters that are passed from the calling program to an external stored procedure.
- Argument indicator: The NULL indicator for each argument. If a NULL value was passed for the corresponding argument, the indicator variable contains -1. If a valid value is passed, the indicator variable contains 0. The function can test the value of an argument indicator. Before using the input parameter in the external stored procedure, check the null indicator. If the corresponding argument contains a NULL value, take corrective action.

6.2.4 GENERAL parameter style

The list of parameters is received by the external stored procedure, when the GENERAL style is specified. See the following example:

IN | OUT | INOUT argument [repeated]

The argument parameters consists of input, output, and input/output parameters that are passed from the calling program to an external stored procedure.

Note: The maximum number of parameters allowed in the CREATE PROCEDURE statement is limited by the programming language used to implement the stored procedure. For a procedure created with the SQL parameter style, the additional implicit parameters are included into the calculation.

6.3 Coding external stored procedures

An external stored procedure does not differ significantly from any other high-level language program you already write. What can make it different is that it is registered to the DB2 Universal Database for iSeries, as described in 6.1, “Registering external stored procedures” on page 124, and in the way it receives and returns results both as parameters and as result sets.
A result set is an open cursor returned by a stored procedure. Result sets are the mechanism stored procedures use for returning multirow results. One stored procedure can return none, one, or multiple result sets, but at any time a calling program can only have 100 procedures with result sets waiting to be fetched.

This section explains the differences in stored procedures code related to the parameter style used as well as the details about returning result sets and error handling.

### 6.3.1 Coding for SQL parameter style

This section looks at examples on how to code external stored procedures with the SQL parameter style. It also demonstrates how the parameters passed by DB2 Universal Database for iSeries to the external stored procedure can be used within the procedure.

In the Order Entry database used throughout this IBM Redbook, we define the ORDERHDR and the CUSTOMER tables. There is a referential constraint established between the CUSTOMER table and the ORDERHDR table. The CUSTOMER table is the parent table, with the parent key CUSNBR. The ORDERHDR table is the dependent table with the foreign key CUSNBR. The delete rule is *RESTRICT.

Let us suppose that you want to delete a particular customer from the CUSTOMER table. To accomplish this task, you implement an external stored procedure, called CusNumDel, using the SQL parameter style. This external stored procedure accepts one input parameter, the customer number (CUSNBR) of type CHAR(5). The external stored procedure executes an SQL DELETE statement to delete the record for the passed customer number from the CUSTOMER table. If the customer with this customer number has an order in the ORDERHDR table, or the customer number to be deleted has a dependency in the ORDERHDR table, an error occurs. Otherwise, the deletion is successful. If an error occurs, it should be returned to the calling program, so it is aware that the customer record deletion failed.

Let us examine the CREATE PROCEDURE statement for the CusNumDel external stored procedure in Example 6-1. The numbered sections are explained further in the list that follows.

**Example 6-1 CREATE PROCEDURE statement for the CusNumDel external stored procedure**

```sql
CREATE PROCEDURE PROCLIB/CUSNUMDEL(
    IN CUSNBR CHAR(5))
  SPECIFIC  CUSNUMDEL
  LANGUAGE  C
  EXTERNAL  NAME SPROCLIB/CUSNUMDEL
  MODIFIES  SQL DATA
  PARAMETER STYLE SQL
```

**CREATE PROCEDURE statement explanation**

The following explanation refers to Example 6-1:

1. We qualify the procedure name with the library name, SPROCLIB in this case. We use the system naming convention. If you do not qualify the procedure name in the CREATE PROCEDURE statement, the procedure is created in the current library. The procedure takes one input parameter CUSNBR of type CHARACTER(5). If there is an existing procedure with the same name, but a different number of parameters in the destination library, collection, or schema, the CREATE PROCEDURE statement will execute successfully. In this case, we say that the stored procedure name was overloaded.

2. This is the SPECIFIC NAME clause of the CREATE PROCEDURE statement. Every procedure created on the iSeries server must have a specific name. This name must be unique in the given library. This is an optional clause. If you do not specify a specific name...
for the procedure, the system will generate a specific name. Normally the specific name is the same as the procedure’s name. However, if a procedure with the specific name already exists, the system generates a unique name.

3 This is the LANGUAGE clause of the CREATE PROCEDURE statement. The LANGUAGE clause specifies the language that was used to implement the external stored procedure. In our case, it is ILE C. This information helps the database to pass parameters to the external stored procedure in the format required by the programming language. External stored procedures can be written in any of the following languages:

- CL
- COBOL
- COBOLLE
- FORTRAN
- JAVA
- PLI
- RPG
- RPGLE
- C
- C++
- REXX

The LANGUAGE clause is optional. If it is not specified, the system tries to retrieve the attributes of the program object specified in the EXTERNAL NAME clause and set the clause accordingly. If the program object does not exist, or if the attribute is not present, the language is defaulted to ILE C.

4 This is the EXTERNAL NAME clause of the CREATE PROCEDURE statement. It is the name of the external program that is called when the external stored procedure is called from the calling program with SQL CALL. In this example, SPROCLIB is the name of the library in which the program resides. CUSNUMDEL is the name of the program that is to be executed. The program does not need to exist at the time of the creation of the external stored procedure, but it must be created before the stored procedure is called for the first time. This is an optional clause. If it is not specified, the system assumes that the name of the program is the same as the name of the stored procedure, provided it’s a valid system name no longer than 10 characters. Two different stored procedures can point to the same external program name. An external program should be a *PGM object; it cannot be an ILE service program.

5 This is the NO/READS/MODIFIES/CONTAINS SQL DATA clause of the CREATE PROCEDURE statement. Here you specify the kind of SQL statements the procedure will execute. Refer to SQL Reference, SC41-5612, for a detailed description of the valid SQL statements for a given clause.

6 This is the PARAMETER STYLE clause of the CREATE PROCEDURE statement. For external stored procedures, it can be set to one of four values:

- SQL
- DB2SQL
- GENERAL WITH NULLS
- GENERAL

DB2 Universal Database for iSeries passes additional parameters apart from the arguments defined in the CREATE PROCEDURE statement based on the parameter style specified.

Now let us examine the external program CUSNUMDEL referred to in the CREATE PROCEDURE statement. We discuss the parameters that DB2 Universal Database for iSeries sends to the program and how the program makes use of these parameters. This program was written in ILE C with embedded SQL. The CUSNUMDEL external program accepts the customer number as the input argument. The SQL DELETE statement is
executed. Any errors or successful deletion is returned to the calling program, using the SQLSTATE output parameter. The SQLSTATE is set to "38IRC" when there is a delete rule violation. When SQLSTATE is set by the external program, the diagnostic message is also returned to the calling program.

Example 6-2 shows how the external stored procedure with the SQL parameter style is coded. The numbered areas are further explained in the list that follows.

Example 6-2  Coding of an external procedure with the SQL parameter style

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <recio.h>
EXEC SQL INCLUDE SQLCA ;
EXEC SQL BEGIN DECLARE SECTION;
char v_custno[5];
short int v_custno_ind;
EXEC SQL END DECLARE SECTION;
main(int argc,char *argv[]) {
   unsigned char statevar[5];
   unsigned char errmc[70];

EXEC SQL
   WHENEVER SQLERROR GO TO Error_Handler;
   struct procname{ short int length;
      unsigned char data[139];
   }procname_var;
   struct specname{ short int length;
      unsigned char data[128];
   }specname_var;
   struct outmsgtxt{ short int length;
      unsigned char data[70];
   }outmsgtxt_var={20,"referential const "};
   strncpy(v_custno,argv[1],5);  /* receives customer number to be deleted */
   v_custno_ind=*(short int*)argv[2]; /* customer number null indicator */
   procname_var=*(struct procname*)argv[4]; /* process name */
   specname_var=*(struct specname*)argv[5]; /* specific name of the SP. */
   if (v_custno_ind == -1)  /* if customer number is null, terminate */
      exit(0);
   /* customer number is deleted */
   EXEC SQL
      DELETE FROM orentlib.customer WHERE cusnbr=v_custno;
   /* SQL status is prepared to be returned as a parameter */
   strncpy(statevar,"00000",5);
   strncpy(argv[3],statevar,5);
   /* Stored procedure completes */
   exit(0);
   /* on error... */
   Error_Handler:
      /* retrieve SQLSTATE describing the error */
      strncpy(statevar,sqlca.sqlstate,5);
      if(sqlca.sqlcode=-532) { /* If the error is caused by a RI violation */
         puts(statevar);
         strncpy(statevar,"38IRC",5);
      }
```
strncpy(argv[3],statevar,5);
strncpy(errmc,
   "referential constraint exists. Customer cannot be deleted",45);
strncpy(outmsgtxt_var.data,errmc,45);
memcpy((void *)argv[6],(void *)&outmsgtxt_var,sizeof(outmsgtxt_var.data));
} else { /* any other error */
strncpy(statevar,"38999",5);
strncpy(argv[3],statevar,5);
strncpy(errmc,"This is an unhandled error. Check the program",47);
strncpy(outmsgtxt_var.data,errmc,47);
memcpy((void *)argv[6],(void *)&outmsgtxt_var,sizeof(outmsgtxt_var));
}
}

---

**Code sample notes**

The procedure name CUSNUMDEL is the name of the source file member and the name of the *PGM object, referred to in the CREATE PROCEDURE statement as shown here:

EXTERNAL NAME SPROCLIB.CUSNUMDEL

The external program coded in any host language should be complied with the Activation Group parameter as "CALLER.

1. The CUSNUMDEL procedure accepts an input parameter of type CHAR(5), which is the customer number to be deleted from the CUSTOMER table.

2. This parameter is a null indicator for the input parameter. Whenever a null value is passed into the program on input, the input null indicator contains -1. If the input parameter is a valid value, the null indicator is 0.

3. Parameter containing the fully qualified name of the procedure.

4. Parameter with the specific name of the procedure that has been called. The specific name can be used when the procedure is overloaded. Even if the procedure names are the same, the specific names have to be unique.

5. The next two parameters are SQLSTATE and the message text. They are used together. The parameter can be used to signal an error or warning condition to the calling program. The procedure can also set the message text output parameter to a customized error message. However, the message text parameter is returned back to the calling program only if the SQLSTATE is set to "38yxx". In our program, we execute the SQL DELETE statement for the non-null customer number that is passed as the input parameter. If the customer number we are trying to delete from the CUSTOMER table has a dependent row in the ORDEHDR table, the SQL DELETE will fail. This generates SQLCODE=-532 and SQLTATE=23001. This SQLSTATE cannot be directly returned to the calling program in the SQLSTATE output parameter, since it would be treated by the database run-time as an invalid state. The database would set the sqlca.sqlstate variable to "39001" and the sqlca.sqlcode to -463, which indicates an invalid SQL status. In our case, the SQLSTATE output parameter is set to "38IRC". When the SQLSTATE is set to a value matching the pattern "38yxx", the diagnostic message text is also returned to the calling program Error_Handler:

```c
strncpy(statevar,sqlca.sqlstate,5);
if(sqlca.sqlcode=-532) {
    puts(statevar);
    strncpy(statevar,"38IRC",5);
    strncpy(argv[3],statevar,5);
    strncpy(errmc,"referential constraint exists cannot delete",45);
```
Refer to Chapter 7, “Java stored procedures” on page 173, for a detailed discussion on error handling.

As mentioned earlier, the CUSNUMDEL program was created as a *PGM object. In this case, CUSNUMDEL is a program written in C with embedded SQL statements. It is compiled into the *MODULE object, and then the *MODULE object is bound into a *PGM object, which allows us to specify the activation group parameter as *CALLER.

The following CL commands are used to compile and bind the CUSNUMDEL program:

```
CRTSQLCI OBJ(SPROCLIB/CUSNUMDEL) SRCFILE(QCSRC/SPROCLIB)
SRCMBR(CUSNUMDEL) OUTPUT(*PRINT) DBGVIEW(*SOURCE)
CRTPGM PGM(SPROCLIB/CUSNUMDEL) ACTGRP(*CALLER)
```

**Calling the external stored procedure with SQL parameter style**

The SQL CALL statement invokes the external stored procedure as shown in Example 6-3. In the calling client program, you can use sqlca.sqlstate or SQLSTATE to check for the error condition that occurred within the stored procedure. If the sqlca.sqlstate matches the pattern "38yxx", the corresponding diagnostic message text can be retrieved from the sqlca.errmc field.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <decimal.h>
#include <recio.h>
#define SIZE 5
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char inpvar[5];
short int inindicator;
EXEC SQL END DECLARE SECTION;
void main(void)
{
  int res1,res2;
  unsigned char errmc[70];
  unsigned char mstatevar[5];
  EXEC SQL
    WHENEVER SQLERROR GOTO printmsg;
  EXEC SQL
    WHENEVER SQLWARNING GOTO printnomsg;
  strcpy(inpvar,"99999");
  puts(inpvar);
  EXEC SQL
    CALL sproclib.cusnumdel(:inpvar);
  strncpy(mstatevar,SQLSTATE,5);
  printf("The SQLSTATE returned from the Stored procedure:%s\n",mstatevar);
  exit(0);
}
```
printfmsg:
strncpy(mstatevar,SQLSTATE,5);
res1=strncmp(mstatevar,"37999",5);
res2=strncmp(mstatevar,"38999",5);
if ((res1 > 0) && (res2 <= 0))
{
    printf("The SQLSTATE returned from the Stored procedure:%s \n",mstatevar);
    strncpy(errmc,sqlca.sqlerrmc,69);
    printf("The message text :%s \n",errmc);
}
else
{
    printf("The invalid SQLSTATE set in the Stored procedure: \n");
    exit(1);
}
printnomsg:
strncpy(mstatevar,SQLSTATE,5);
printf("The Stored procedure returned a warning:\n");
printf("The SQLSTATE returned from the Stored procedure:%s\n",mstatevar);
exit(0);

**Note:** In the call statement shown in bold, we do not specify extra parameters for the SQL parameter style. They are implicitly passed to the stored procedure by SQL run time.

### 6.3.2 Coding the DB2SQL parameter style

This section looks at examples on how to code external stored procedure with the DB2SQL parameter style when DBINFO is specified. When DB2SQL parameter style is used in combination with NO DBINFO, it has the same effect as coding with SQL parameter style.

Following the same example shown in 6.3.1, “Coding for SQL parameter style” on page 132, suppose that we are required to modify the CusNumDel external stored procedure in order to record a trace row in table DELCTL notifying the user, time stamp, and deleted customer number.

Let us examine the CREATE PROCEDURE statement for the CusNumDel2 external stored procedure:

```
CREATE PROCEDURE PROCLIB.CUSNUMDEL2 (  
    IN CUSNBR CHAR(5))  
SPECIFIC CUSNUMDEL2  
LANGUAGE C  
EXTERNAL NAME SPROCLIB.CUSNUMDEL2  
MODIFIES SQL DATA  
PARAMETER STYLE DB2SQL  
DBINFO
```

**Note:** In regard to the lines in bold in the previous example, PARAMETER STYLE DB2SQL in combination with DBINFO instructs DB2 Universal Database for iSeries to pass the DBINFO data structure containing the fields described in Table 6-1.
Now let us examine the external program CUSNUMDEL2 referred to in the CREATE
PROCEDURE statement (see Example 6-4). This is similar to the CUSTNUMDEL exposed
earlier. However, it differs in that it uses the Authorization ID received as part of the DBINFO
data structure and adds a row into the DELCTL table.

Example 6-4 External program CUSNUMDEL2

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <recio.h>

/* sqludf.h contains UDF data structures that gives a very good */
/* compatibility level between DB2 UDB platforms, including */
/* sqludf_dbinfo, which is the data structure for DBINFO */
#include <sqludf.h>

EXEC SQL INCLUDE SQLCA ;
EXEC SQL BEGIN DECLARE SECTION;
char inputvar[5];
short int inpindicator;
/* the following data structure is to contain the user ID of the */
/* user calling the stored procedure, using traditional VARCHAR */
/* structure */
struct {
    short useridlen;
    char useridtxt[128];
} user_id;
EXEC SQL END DECLARE SECTION;

/* parameter 1: user defined parameter */
/* parameter 2: Null indicator for the user defined parameter */
/* parameter 3: Output parameter for SQLSTATE */
/* parameter 4: Fully qualified procedure name */
/* parameter 5: Specific name */
/* parameter 6: Output parameter for message text */
/* parameter 7: DBINFO data structure. */
main(int argc,char *argv[])
{
    unsigned char statevar[5];
    unsigned char errmc[70];

...
EXEC SQL
WHENEVER SQLERROR GO TO Error_Handler;

struct proname { short int length;
    unsigned char data[139];
    } proname_var;
struct specname { short int length;
    unsigned char data[128];
    } specname_var;
struct msgtxt  { short int length;
    unsigned char data[70];
    } msgtxt_var;
struct outmsgtxt{ short int length;
    unsigned char data[70];
    } outmsgtxt_var={20,"referential consist "};
/* The sqludf_dbinfo structure is predefined in the sqludf.h */
/* include file. Variable dbinfo is used to receive the */
/* DBINFO data structure */
struct sqludf_dbinfo dbinfo;

strncpy(inputvar,argv[1],5);
inpindicador=*(short int*)argv[2];
proname_var=*(struct proname*)argv[4];
specname_var=*(struct specname*)argv[5];

/* retrieving DBINFO */
dbinfo = *(struct sqludf_dbinfo*)argv[7];
/* retrieving the user id from DBINFO data structure */
user_id.useridlen = dbinfo.authidlen;
strncpy(user_id.useridtxt, dbinfo.authid, dbinfo.authidlen);
EXEC SQL
DELETE FROM ORDAPPLIB.CUSTOMER WHERE CUSNBR=:inputvar;
EXEC SQL
INSERT INTO ORDAPPLIB.DELCTL (CUSNBR, LAST_MOD_USR, LAST_MOD_TS)
VALUES (:inputvar, :user_id, CURRENT TIMESTAMP);
strncpy(statevar,"00000",5);
strncpy(argv[3],statevar,5);
exit(0);

Error_Handler:
strncpy(statevar,sqlca.sqlstate,5);
if(sqlca.sqlcode=-532)
{
    puts(statevar);
    strncpy(statevar,"38IRC",5);
    strncpy(argv[3],statevar,5);
    strncpy(errmc,"referential constraint exists cannot delete",45);
    strncpy(outmsgtxt_var.data,errmc,45);
    memcpy((void *)&outmsgtxt_var,
            (void *)&outmsgtxt_var.data,
            sizeof(outmsgtxt_var.data));
}
else
{
    strncpy(statevar,"38999",5);
    strncpy(argv[3],statevar,5);
}
strncpy(errmc,"this is an unhandled error check the program",46);
strncpy(outmsgtxt_var.data,errmc,46);
memcpy((void *)argv[6],
        (void *)&outmsgtxt_var,
        sizeof(outmsgtxt_var));

Code sample notes
The procedure named CUSNUMDEL2 (Example 6-4) is similar to CUSNUMDEL, but uses the DBINFO data structure to get the user ID that is invoking this stored procedure and keeps track of it on the DELCTL table. We highlight the differences here:

1. The CUSNUMDEL2 uses the DBINFO data structure common to other DB2 Universal Database platforms. This data structure is defined in the sqludf.h include file. For other programming languages, there are corresponding include files in the QSYSINC library. For example, you will find member SQLUDF in QSYSINC/QRPGLESRC to be copied into ILE RPG programs.
2. VARCHAR data structure to store user ID.
3. We defined here a variable called dbinfo based on the structure db2udf_dbinfo. This structure is predefined in sqludf.h.
4. DBINFO is passed to the stored procedure program after SQL parameters introduced in 6.3.1, “Coding for SQL parameter style” on page 132.
5. The caller user ID is located in DBINFO in varchar style. For details about the information available in DBINFO, refer to the sqludf.h include file.
6. This line inserts a new line into the DELCTL table.

Calling an external stored procedure with DB2SQL parameter style
There is no difference between a DB2SQL and an SQL parameter style external stored procedure from the caller program perspective. For details about calling an external stored procedure with DB2SQL parameter style and how to use SQLSTATE to check for possible error conditions occurred within the stored procedure, refer to “Calling the external stored procedure with SQL parameter style” on page 136.

6.3.3 Coding the GENERAL WITH NULLS parameter style
This section looks at an example of how to code an external stored procedure with the GENERAL WITH NULLS parameter style.

In our Order Entry database, there are two tables: ORDERHDR and ORDERDTL. There is a referential constraint defined between the ORDERHDR table and the ORDERDTL table. The ORDERHDR table is the parent table, with the parent key ORHNBR. The ORDERDTL table is the dependent table, with the foreign key ORHNBR.

Let us suppose that you want to insert the given order detail values into the ORDERDTL table. To accomplish this task, we implement this as a external stored procedure ORDDETINS using the general with null parameter style. This external stored procedure accepts four input parameters to be inserted into the ORDERDTL table:

- The order number orhnbr CHAR(5)
- The product number prdnbr CHAR(5)
- The order detail quantity ordQty DECIMAL(5,0)
- The order detail total orhtot DECIMAL(5,0)
The external stored procedure accepts the input parameters and executes an SQL INSERT statement to add a new row to the ORDERDTL table. If the order number ORHNBR to be inserted into the ORDERDTL table does not have an order with this order number in the ORDEHDR table, an error occurs. Otherwise, the insertion is successful. If an error occurs, it should be returned to the calling program, indicating the failure of the order detail insertion.

Let us examine the CREATE PROCEDURE statement for the ORDDETINS external stored procedure. The numbered sections are explained in the list that follows:

```sql
CREATE PROCEDURE SPROCLIB.ORDDETINS(
    IN ORHNBR CHAR(5),
    IN PRDNBR CHAR(5),
    IN ORDQTY DECIMAL(5,0),
    IN ORDTOT DECIMAL(9,2),
    OUT SQLST CHAR(5)  )
SPECIFIC ORDDETINS
LANGUAGE RPGLE
EXTERNAL NAME SPROCLIB.ORDDETINS
MODIFIES SQL DATA
PARAMETER STYLE GENERAL WITH NULLS
```

**CREATE PROCEDURE statement explanation**

The following notes refer to the previous example.

1. We qualify the procedure name with the library name SPROCLIB in this case. We use the system naming convention. If you do not qualify the procedure name in the CREATE PROCEDURE statement, the procedure is created in the current library. The procedure takes four input parameters: ORHNBR of type CHARACTER(5), PRDNBR of type CHARACTER(5), ORDQTY of type DECIMAL(11,2), and ORDTOT of type DECIMAL (11,2). An output parameter has been defined: SQLST of type CHAR(5).

2. This is the EXTERNAL NAME clause of the CREATE PROCEDURE statement. The external program is a *PGM object.

3. This is the PARAMETER STYLE clause. DB2 Universal Database for iSeries passes indicators as additional parameters, apart from the input output arguments defined in the CREATE PROCEDURE statement. When the stored procedure is registered using the iSeries Navigator, the parameter style Simple, allow null values must be selected.

Now let us examine the external program ORDDETINS referred to in the CREATE PROCEDURE statement above. This program is written in ILE RPG with embedded SQL.

The ORDDETINS external program accepts the order number, product number, order quantity, and the order quantity total for ordered product as the input arguments. The SQL INSERT statement is executed. If the order number passed to the stored procedure does not exist in the ORDERHDR table, the insert statement fails because the referential integrity constraint is enforced. The error or successful insertion status is returned to the calling program, using the sqlstate output parameter.
The code sample in Example 6-5 illustrates how the external stored procedure with the GENERAL WITH NULLS parameter style is coded. The numbered areas are further explained in the list that follows.

**Example 6-5  External stored procedure with the GENERAL WITH NULLS parameter style**

```
  dindds       ds
dindd1              1  2B  0
dindd2              3  4B  0
dindd3              5  6B  0
dindd4              7  8B  0
doutdd            s        1b  0

  c       *entry        plist
  c       parm        ordnbr     5
  c       parm        prdnbr     5
  c       parm        ordqty     5  0
  c       parm        ordtot     9  2
  c       parm        state      5
  c       parm        indds
  c       parm        outdd

C/EXEC SQL
C+ WHENEVER SQLERROR GOTO ERROR
C/END-EXEC
C/EXEC SQL
C+ INSERT INTO ORDENTLIB.ORDERDTL(ORHNBR, PRDNBR, ORDQTY, ORDTOT)
C+ VALUES(:ORDNBR, :PRDNBR, :ORDQTY, :ORDTOT)
C/END-EXEC
C
eval      state='00000'
goto      END
ERROR      TAG
if        SQLCOD=-530
  eval      state='38IRC'
else
  eval      state='38999'
endif
END      TAG
  eval      *inlr=*ON
```

**Code sample notes**

The following notes refer to Example 6-5:

1. The ORDDETINS program accepts the first four input parameters, which are to be inserted into the ORDERDTL table. DB2 Universal Database for iSeries passes the corresponding number of indicator variables as additional parameters to the procedure.

2. The last parameter is an output parameter SQLState. The parameter can be used to signal an error or warning condition on return to the calling program. In our program, we execute the INSERT statement for the order number that is passed as the input parameter. The order number we are trying to insert into the ORDERDTL table may not have matching value in the orhnbr primary key column of the ORDEHDR table. In this case, the SQL INSERT will fail, generating SQLCODE=-530 and SQLSTATE=23503.

3. These parameters are null indicators for input and output parameters. Whenever a null value is passed into the program on input, the input null indicator contains -1. Since there are four input parameters, there are four indicators associated with these parameters.

As mentioned earlier, the ORDDETINS program was created as a *PGM object in RPG with embedded SQL statements. It is compiled into a *MODULE object, and then the *MODULE object is bound into the *PGM object, which allows us to specify the activation group parameter as *CALLER.
The following CL commands are used to compile and bind the ORDDETINS program:

CRTSQLRPGI OBJ(SPROCLIB/ORDDETINS) SRCFILE(QCSRC/SPROCLIB) OPTION(*SQL)
SRCMBR(ORDDETINS) OUTPUT(*PRINT) DBGVIEW(*SOURCE) OPTION(*SQL)
CRTPGM PGM(SPROCLIB/ORDDETINS) ACTGRP(*CALLER)

Calling the external stored procedure

To invoke the external stored procedure, use the CALL statement, as shown in Example 6-6.

In the calling program or the client program, you can use the output parameter to check if there was an error within the stored procedure.

Example 6-6   Call to the external stored procedure

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <decimal.h>
#include <recio.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char OrdNbr[5];
char PrdNbr[5];
decimal(5,0) OrdQty;
decimal(9,2) OrdTot;
char state[5];
short int ind1;
short int ind2;
short int ind3;
short int ind4;
short int ind5;
EXEC SQL END DECLARE SECTION;
void main(void) {
    EXEC SQL WHENEVER SQLERROR GOTO printmsg;
    EXEC SQL WHENEVER SQLWARNING GOTO printnomsg;
    printf("Enter the ORDERDETAILS:
    ....
    EXEC SQL CALL SPROCLIB.ORDDETINS(:OrdNbr :ind1, :PrdNbr :ind2,
    :OrdQty :ind3, :OrdTot :ind4, :state :ind5);
    printf("The SQLSTATE returned from the Stored procedure:%s\n",state);
    exit(0);
    printmsg:
    if((strcmp(state,"37999",5) > 0) &&
        (strcmp(state,"38999",5) <= 0))
        printf("The SQLSTATE returned from the Stored procedure:%s \n",state);
    else
        printf("The invalid SQLSTATE set in the Stored procedure:%s \n",state);
    exit(1);
    printnomsg:
    printf("The Stored procedure returned a warning:\n");
    printf("The SQLSTATE returned from the Stored procedure:%s\n",state);
    exit(0);
}
```

**Note:** The code shown in bold illustrates how to use the user-defined SQL state set by the external stored procedure registered with the GENERAL WITH NULLS parameter style.
6.4 Returning result sets from external procedures

Until now, we discussed coding external stored procedures with different parameter styles and input/output parameters. These programs returned only stand-alone output values. However, you can also return a result set from an external stored procedure. There are two ways to return a result set:

- Cursor result set
- Array result set

The following section discusses these two methods. Examples are included to show how to code an external stored procedure that returns multiple result sets.

When the external stored procedure is created with the CREATE PROCEDURE statement or using the iSeries Navigator’s New Procedure window, you should specify the number of result sets that have to be returned to the calling program. An invoker must use JDBC, ODBC, or CLI when calling a procedure that returns result sets. The iSeries server has no embedded SQL support for handling result sets.

6.4.1 Coding external stored procedures returning cursor result sets

In this section, we return to our task of finding the best and the worst suppliers in the Order Entry database. The business logic was discussed in 5.7, “GetSuppliers example” on page 111.

Let us examine the CREATE PROCEDURE statement for the Get_Supplier_Rs external stored procedure, which returns multiple result sets. The numbered sections are explained further in the list that follows:

```sql
CREATE PROCEDURE SPROCLIB.Get_Supplier_Rs(
  IN year INTEGER,
  IN month INTEGER,
  INOUT rank INTEGER )
RESULT SETS 2
SPECIFIC Get_Supplier_Rs
LANGUAGE RPGLE
EXTERNAL NAME SPROCLIB.SELPGMRESR
MODIFIES SQL DATA
PARAMETER STYLE SQL
```

CREATE PROCEDURE statement explanation

The following notes refer to the previous example.

1. We qualify the procedure name with the library name, which in this case, is the library SPROCLIB. We use the system naming convention. If you do not qualify the procedure name in the CREATE PROCEDURE statement, the procedure is created in the current library. The procedure accepts two input parameters and one INOUT parameter rank.

2. This is the number of result sets returned from the procedure.

3. The parameter style is SQL.

Now let us examine the external program SELPGMRESR referred to in the CREATE PROCEDURE statement above. This program is written in ILE RPG with embedded SQL. The SELPGMRESR external program accepts the year, month, and rank as the input arguments. The stored procedure can be called with the SQL CALL statement. The source for the SELPGMRESR is shown in Example 6-7.
Example 6-7  SELPGMRESR source code

dCounter1 s 10i 0 inz(0)
dCounter2 s 10i 0 inz(0)
dtemp s 10i 0 inz(0)
dfsqIcod s 5i 0 inz(0)
dhsales s 11p 2
dyear s 10i 0
dmonth s 10i 0
drank s 10i 0
dindds ds
dindd1 1 2b 0
dindd2 3 4b 0
dindd3 5 6b 0
doutdd s 1b 0
dsqlstate s 5a
dprocname s 138a varying
dspecname s 128a varying
dmsgtxt s 70a varying

c *entry plist
  c parm year
  c parm month
  c parm rank
  c parm indds
  c parm outdd
  c parm sqlstate
  c parm procname
  c parm specname
  c parm msgtxt

c/exec sql
  c+ declare alt0 scroll cursor for
  c+ select totalsales
  c+ from suparna1/totalsale
  c+ where year=:year and month=:month
  c+ order by totalsales desc
  c/end-exec

c/exec sql
  c+ declare alt1 cursor for
  c+ select supplier_name,totalsales
  c+ from suparna1/totalsale
  c+ where year=:year and month=:month and totalsales>= :hsales
  c+ order by totalsales desc
  c/end-exec

c*/
c/exec sql
  c+ declare alt2 scroll cursor for
  c+ select totalsales
  c+ from suparna1/yearsale
  c+ where year=:year
  c+ order by totalsales desc
  c/end-exec

c/exec sql
  c+ declare alt3 cursor for
  c+ select supplier_name,totalsales
  c+ from suparna1/yearsale
  c+ where year=:year and totalsales>= :hsales
  c+ order by totalsales desc
  c/end-exec

c/exec sql
  c+ declare alt4 scroll cursor for
  c+ select totalsales
c+ from suparna1/totalsale
  c+ where year=:year and month=:month
  c+ order by totalsales
  c/end-exec
  c/exec sql
  c+ declare alt5 cursor for
    select  supplier_name, totalsales
    from suparna1/totalsale
    where year=:year and month=:month and totalsales<=:hsales
    c+ order by totalsales
  c/end-exec
  c*
  c/exec sql
  c+ declare alt6 scroll cursor for
    select  totalsales
    from suparna1/yearsale
    where year=:year
    c+ order by totalsales
  c/end-exec
  c/exec sql
  c+ declare alt7 cursor for
    select  supplier_name, totalsales
    from suparna1/yearsale
    where year=:year and totalsales<=:hsales
    c+ order by totalsales
  c/end-exec
  c                   if        (month=0)
  c/exec sql
  c+ open alt2
  c/end-exec
  c* in this loop -fetch all the rows in the resultant set into var:array
  c                   dow       ((sqlcod<>100) and (Counter1<rank))
  c/exec sql
  c+ fetch from alt2
  c/end-exec
  c                   eval      Counter1=Counter1+1
  c                   enddo
  c                   if        (sqlcod=100)
  c                   eval      Counter1=Counter1-1
  c                   eval      fsqlcod=1
  c                   eval      temp=rank
  c                   eval      rank=Counter1
  c                   endif
  c/exec sql
  c+ close alt2
  c/end-exec
  c/exec sql
  c+ open alt2
  c/end-exec
  c* in this loop -fetch all the rows in the resultant set into var:array
  c/exec sql
  c+ fetch relative :rank from alt2 into :hsales
  c/end-exec
  c                   if        (fsqlcod=1)
  c                   eval      rank=temp
  c                   endif
  c/exec sql
  c+ close alt2
  c/end-exec
  c/exec sql
c+ open alt3
/c/end-exec
/c/exec sql
/c+ open alt6
/c/end-exec

/* in this loop -fetch all the rows in the resultant set into var:array
   c         dow   ((sqlcod<>100) and (Counter2<rank))
/c/exec sql
/c+ fetch from alt6
/c/end-exec

c  eval   Counter2=Counter2+1
/c enddo
/c if (sqlcod=100)
c  eval   fsqlcod=1
/c eval   Counter2=Counter2-1
/c eval   temp=rank
/c eval   rank=Counter2
/c endif
/c/exec sql
/c+ close alt6
/c/end-exec
/c/exec sql
/c+ open alt7
/c/end-exec
/c/exec sql
/c+ set result sets cursor alt3,cursor alt7
/c/end-exec
/c if Counter1>Counter2
/c eval   rank=Counter1
/c else
/c eval   rank=Counter2
/c endif
/c else
/c/exec sql
/c+ open alt0
/c/end-exec
/* in this loop -fetch all the rows in the resultant set into var:array
   c         dow   ((sqlcod<>100) and (Counter1<rank))
/c/exec sql
/c+ fetch from alt0
/c/end-exec
/c eval   Counter1=Counter1+1
/c enddo
/c if (sqlcod=100)
c  eval   fsqlcod=1
/c eval   Counter1=Counter1-1
c                   eval      temp=rank
r                     eval      rank=Counter1
r                     endif
exec sql
r+ close alt0
r/end-exec
exec sql
r+ open alt0
r/end-exec
r* in this loop -fetch all the rows in the resultant set into var:array
exec sql
r+ fetch  relative :rank from  alt0 into :hsales
r/end-exec
r                   if        (fsqlcod=1)
r                   eval      rank=temp
r                   endif
exec sql
r+ close alt0
r/end-exec
exec sql
r+ open alt1
r/end-exec
exec sql
r+ open alt4
r/end-exec
r* in this loop -fetch all the rows in the resultant set into var:array
exec sql
r+ fetch  from  alt4
r/end-exec
r                   eval      Counter2=Counter2+1
r                   enddo
r                   if        (sqlcod<>100) and (Counter2<rank))
exec sql
r+ set result sets cursor alt1,cursor alt5
r/end-exec
Code sample notes
The external program name SELPGMRESR (Example 6-7 on page 145) is a *PGM object, referred to in the CREATE PROCEDURE statement in the following way:

```c
if Counter1>Counter2
  eval rank=Counter1
else
  eval rank=Counter2
endif
endif
return
```

1. To find the total number of suppliers in a given year and month, we declare the SELECT cursor. To find \( n \) highest total sales, we use the ORDER BY clause in the DECLARE CURSOR statement. We ORDER BY total sales in descending order. Therefore, we have the highest total sales value in the first row of the resultant table when the cursor is opened.

2. To find the \( n \) lowest total sales, we use the ORDER BY clause in the DECLARE CURSOR statement. We ORDER BY total sales in ascending order. Therefore, we have the lowest total sales value in the first row of the resultant table when the cursor is opened.

3. The cursor may return less rows than the requested value of RANK. The exact number of rows that can be returned is calculated in a counter that is increased until SQLCODE = 100.

4. The cursor may retrieve any number of rows, but the procedure should return only \( n \) rows, where \( n \) is the value of the third parameter RANK. The FETCH RELATIVE statement is used to retrieve the “cut off” sales value into the hsales variable. Since we use the FETCH RELATIVE statement, the cursor needs to be declared as a scrollable cursor.

5. The \( n \)th total sales value fetched from the ALT0 cursor is used in the ALT1 cursor to create a result set that contains records for \( n \) highest total sales.

6. This cursor ALT1 is opened and returned as a result set.

7. Similarly, the \( n \)th total sales value fetched from the ALT4 cursor is used in the ALT5 cursor to create a result set that contains records for \( n \) lowest total sales.

8. This cursor ALT5 is opened and returned as the second result set.

9. When the month is not entered by the user, the procedure returns the \( n \) highest total sales and the \( n \) lowest total sales for the whole year rather than for a given month within a year. To cover this case, we use a different set of cursors. The result sets are returned in ALT3 and ALT7 cursors.

As mentioned earlier, the SELPGMRESR program was created as a *PGM object. In this case, if SELPGMRESR is a program written in C with embedded SQL statements, it is compiled into a *MODULE object. Then the *MODULE object is bound into the *PGM object, which allows us to specify the activation group parameter as *CALLER.

Calling stored procedures returning multiple result sets
To invoke the external stored procedure, use the SQL CALL statement from the client program. The CLI client programs written in RPGLE and C are provided in the download package. The CLI C client program that can be used to call this external stored procedure is explained in 6.4, “Returning result sets from external procedures” on page 144.
6.4.2 Coding external stored procedures returning array result sets

This section shows how to code an array result set. The external stored procedure should be registered using either the CREATE PROCEDURE statement or the iSeries Navigator’s New Procedure window. The number of result sets returned should be set to one. An external stored procedure that returns an array result set can return only one result set.

Example 6-8 shows the appropriate CREATE PROCEDURE statement.

**Example 6-8  CREATE PROCEDURE statement**

<table>
<thead>
<tr>
<th>CREATE PROCEDURE</th>
<th>SPROCLIB.SELPGMARR(</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IN orhnbr CHAR(5) )</td>
</tr>
<tr>
<td>RESULT SETS</td>
<td>1</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>RPGLE</td>
</tr>
<tr>
<td>EXTERNAL NAME</td>
<td>SPROCLIB.SELPGMARR</td>
</tr>
<tr>
<td>READS</td>
<td>SQL DATA</td>
</tr>
<tr>
<td>PARAMETER STYLE</td>
<td>GENERAL</td>
</tr>
</tbody>
</table>

The ORDERDTL table contains the order detail entries for every order number in the ORDERHDR table. For every order number in the ORDERHDR table, there is a number of rows in the ORDERDTL table with the same order number, but with different product numbers. Suppose that we want to retrieve all rows with the same order number from the ORDERDTL table. You can process the ORDERDTL table as a file and return the rows as an array result set. The ORDERDTL file is processed in input mode for read only to retrieve records with the order number ORHNBR passed as an input parameter. The records are retrieved until EOF. The retrieved records are placed in an array as they are retrieved. On EOF, the array result set is returned to the client program. See Example 6-9.

**Example 6-9  ORDERDTL processed as a file and return rows as an array result set**

```plaintext
for ordtl in new  K disk rename(orddtl:norderdtl)
di s 3 0
dproduct ds occurs(5)
dnumber 5
  c *entry plist
  c parm ordnbr 5
  c eval i=0
  c *LOVAL SETLL norderdtl
  c read norderdtl
  c dow not(%eof)
  c if orhnbr=ordnbr
  c eval i=i+1
  c i occur product
  c move prdnbr product
  c endif
  c read norderdtl
  c enddo
  c* in this loop -fetch all the rows in the resultant set into var:array
  c/exec sql
  c+ set result sets array :product for :i rows
  c/end-exec
  c return
```
Sample code notes
The following notes refer to Example 6-9:
1. This is the data structure declaration to hold the rows of the array result set as they are read from the ORDERDTL file.
2. The SET RESULT SETS ARRAY: array FOR: count ROWS returns an array result set with a number of rows.

The following CL commands are used to compile and bind the SELPGMARR program:

```sql
CRTSQLRPGI OBJ(SPROCLIB/SELPGMARR) SRCFILE(QCSRC/SPROCTLIB)
   (SRCMBR(SELPGNRES) OUTPUT(*PRINT) DBGVIEW(*SOURCE)
   CRTPGM PGM(SPROCLIB/SELPGMARR) ACTGRP(*CALLER)
```

To invoke the external stored procedure from a client program, use the SQL CALL statement. The result set returned from the procedure is shown in Figure 6-6.

![Figure 6-6](image)

6.5 CLI client program that calls a procedure that returns multiple result sets

When calling a local or remote stored procedure, your client application should be connected to the target remote system at run time. You can use the SQL CONNECT or SET CONNECTION statements to connect to a database. You can also specify the database name explicitly at compile time using the compiler parameter called Relational Database (RDB). When you run an application program compiled with a Relational Database name, it is implicitly connected to the database specified in the parameter. This database may be either local or remote.

If you are going to call a remote stored procedure, remember to create an SQL package on the remote system. The SQL package is needed to call remote stored procedures.

In 6.4, “Returning result sets from external procedures” on page 144, we implemented an example that returns multiple result sets. In this section, we examine the CLI program written in ILE C, which calls the Get_Supplier_Rs stored procedure and displays the multiple result sets returned by the stored procedure. We present the most important parts of the CLI program. The complete listing of the CLI C source and the listing of the RPG version are available for download from the Web. The CLI C client program is called GETSUPPRS, and the CLI RPG client is called GETSUPPRSR.
We now examine the CLI C client program. See Example 6-10. The client program is created as a "PGM object.

**Example 6-10   CLI C client program**

```c
#include <stdio.h>
/****variable declaration**************/
..............
/****program *****/
printf( "Please enter the name of the server to connect :
" );
gets( Chr_ServerName );
printf( "Please enter th User Id :
" );
gets( Chr_UserId );
printf( "Please enter the Pass Word :
" );
gets( Chr_PassWord );
/**connect ************/
Nmi_ReturnCode = Fun_Connect( &Hnd_Henv, &Hnd_Hdbc, 
    Chr_ServerName, Chr_UserId, Chr_PassWord );
if ( Nmi_ReturnCode != SQL_SUCCESS )
{
    strcpy( Chr_UserMessage, "Error Connecting to the sever" );
    Nmi_ReturnCode = Fun_PrintError();
    exit( -1 );
}

strcpy( Chr_SqlStatement001, "call  ");
strcat( Chr_SqlStatement001, Chr_Libname );
strcat( Chr_SqlStatement001, ".");
strcat( Chr_SqlStatement001, Chr_Procedure );
strcat( Chr_SqlStatement001, "(" );
strcat( Chr_SqlStatement001, "?" );
strcat( Chr_SqlStatement001, ", ?" );
strcat( Chr_SqlStatement001, ", ?" );
strcat( Chr_SqlStatement001, ")" );

..............
Nmi_ReturnCode = SQLPrepare( Hnd_Hstmt, Chr_SqlStatement001, SQL_NTS );
if ( Nmi_ReturnCode != SQL_SUCCESS )
{
    Nmi_CleanUpCode = 4;
    strcpy( Chr_UserMessage, "Error in Preparing the statement" );
    Nmi_ReturnCode = Fun_PrintError();
    exit( -1 );
}

..............
Nmi_PcbValue=0;
Nmi_ReturnCode = SQLBindParameter( Hnd_Hstmt, 1, SQL_PARAM_INPUT, 
    SQL_INTEGER,SQL_INTEGER,
    sizeof( Nmpd_Year),0,
    ( SQLPOINTER ) &Nmpd_Year,
    sizeof( Nmpd_Year),
    ( SQLINTEGER * ) &Nmi_PcbValue );
..............

..............
Nmi_ReturnCode = SQLExecute( Hnd_Hstmt );
..............
Nmi_ReturnCode = SQLBindCol( Hnd_Hstmt, 1, SQL_CHAR,
    ( SQLPOINTER ) Chr_Supplier_Name,
    sizeof(Chr_Supplier_Name ),
    ( SQLINTEGER * ) &Nmi_PcbValue );
if ( Nmi_ReturnCode != SQL_SUCCESS )
```
{ 
    Nmi_CleanUpCode = 4;
    strcpy( Chr_UserMessage, "Error in Binding the Column Supplier Name");
    Nmi_ReturnCode = Fun_PrintError();
    exit( -1 );
}

Nmi_ReturnCode = SQLBindCol( Hnd_Hstmt, 2, SQL_DECIMAL,
    ( SQLPOINTER ) &Nmpd_Totalsales,
    ( SQLINTEGER ) ((256 * 11) + 2 ),
    ( SQLINTEGER * ) &Nmi_PcbValue );
if ( Nmi_ReturnCode != SQL_SUCCESS )
{
    Nmi_CleanUpCode = 4;
    strcpy( Chr_UserMessage, "Error in Binding the Column Totalsales" );
    Nmi_ReturnCode = Fun_PrintError();
    exit( -1 );
}

...........
printf( "The No of rows returned %d:\n", Nmpd_Rank );
printf( "The List of Supplier with %d Best Totalsales:\n" );

while ( Nmi_ReturnCode == SQL_SUCCESS )
{
    Nmi_ReturnCode = SQLFetch( Hnd_Hstmt );
    if ( Nmi_ReturnCode == SQL_SUCCESS )
    {
        printf( "The value of Supplier Name is %s\n", Chr_Supplier_Name );
        printf( "The value of Total Sales   is %D(11,2)\n", Nmpd_Totalsales );
        printf( "\n\n" );
    }
}

/*check for more result sets***********/
Nmi_ReturnCode = SQLMoreResults( Hnd_Hstmt );
if ( ( Nmi_ReturnCode == SQL_ERROR ) ||
    ( Nmi_ReturnCode == SQL_INVALID_HANDLE ) )
{
    Nmi_CleanUpCode = 4;
    strcpy( Chr_UserMessage, "Error in Getting result set data" );
    Nmi_ReturnCode = Fun_PrintError();
    exit( -1 );
}
if ( Nmi_ReturnCode == SQL_NO_DATA_FOUND )
{
    Nmi_ReturnCode = Fun_CleanUp004( &Hnd_Henv, &Hnd_Hdbc, &Hnd_Hstmt );
    printf( "There are no more result result sets\n" );
    exit( 0 );
}
/*Bind columns for the result set***********/
Nmi_ReturnCode = SQLBindCol( Hnd_Hstmt, 1, SQL_CHAR,
    ( SQLPOINTER ) Chr_Supplier_Name,
    sizeof( Chr_Supplier_Name ),
    ( SQLINTEGER * ) &Nmi_PcbValue );

...........
printf( "The No of rows returned %d\n", Nmpd_Rank );
printf( "The List of Supplier with %d Worst TotalSales:\n" );

while ( Nmi_ReturnCode == SQL_SUCCESS )
{ 

Nmi_ReturnCode = SQLFetch( Hnd_Hstmt );
if ( Nmi_ReturnCode == SQL_SUCCESS )
{
    printf( "The value of Supplier Name is %s\n", Chr_Supplier_Name );
    printf( "The value of Total Sales is %D(11,2)\n", Nmpd_Totalsales );
    printf( "\n\n" );
}
/*check for more result sets*********/

Nmi_ReturnCode = Fun_CleanUp004( &Hnd_Henv,
	&Hnd_Hdbc,
	&Hnd_Hstmt );

Code sample notes
The following notes refer to Example 6-10 on page 152:

1  The CALL statement is prepared with parameter markers. All the parameters, the name of
the procedure to be executed, and the library name are taken as input from the user.
Therefore, this CLI program can be used to execute any stored procedure written in C,
RPGLE, SQL, or Java that implements the business logic explained in 6.4, “Returning
result sets from external procedures” on page 144.

2  The SQLBindParameter ( ) function binds the parameter to the parameter markers before
the SQLExecute function is executed.

3  The SQLBindCol ( ) function binds the columns of the result set. For every column of the
result set, the appropriate SQLBindCol function has to be executed.

4  After the SQLBindCol, the SQLFetch function retrieves the column values from the
returned result set. SQLFetch executes until there are no more rows in the result set.

5  We check for more results sets with the SQLMoreResults function. If more than one result
set is found, the sequence of SQLBindCol and SQLFetch calls has to be repeated.

6  After the second result set is retrieved, we check again for more result sets. The result
depends on the number of result sets returned by the stored procedure. Finally, the
program releases all the handlers it acquired and disconnects from the server.

The results of executing the GETSUPPRS program are shown in Figure 6-7 on page 155.
Debugging a stored procedure

The external stored procedures called from the client program can be debugged as any other native ILE C program. To enable debugging, make sure that you compile the stored procedure with the DBGVIEW set to *SOURCE, as shown in the following example:

```
CRTSQLRPGI OBJ(SPROCLIB/SELPGMRESR) SRCFILE(QCSRC/SPROCTLIB) SRCMBR(SELPGMRESR) OUTPUT(*PRINT) DBGVIEW(*SOURCE)
CRTPGM PGM(SPROCLIB/SELPGMRESR) ACTGRP(*CALLER)
```

You can start the debug session with the Start Debug CL command as shown here:

```
STRDBG PGM(SPROCLIB/SELPGMRESR)
```

Figure 6-7  The CLI C client output

<table>
<thead>
<tr>
<th>Supplier Name</th>
<th>Total Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>8800.00</td>
</tr>
<tr>
<td>Red</td>
<td>7345.00</td>
</tr>
<tr>
<td>Blue</td>
<td>3150.00</td>
</tr>
<tr>
<td>Yellow</td>
<td>1200.00</td>
</tr>
</tbody>
</table>

The No of rows returned 4
The List of Supplier with 4 Best Total Sales:
- Supplier Name: Black, Total Sales: 8800.00
- Supplier Name: Red, Total Sales: 7345.00
- Supplier Name: Blue, Total Sales: 3150.00
- Supplier Name: Yellow, Total Sales: 1200.00

The No of rows returned 4
The List of Supplier with 4 Worst TotalSales:
- Supplier Name: Yellow, Total Sales: 1200.00
- Supplier Name: Blue, Total Sales: 3150.00
- Supplier Name: Red, Total Sales: 7345.00
- Supplier Name: Black, Total Sales: 8800.00

There are no more result result sets
Press ENTER to end terminal session.
The stored procedure source should now be loaded in a debug session. Set the required breakpoints, and press F12 to return to the CL prompt. Now you can run the CLI client with the following call:

```
CALL GETSUPPRS
```

The program execution stops after the control is passed to the stored procedure and the first breakpoint is reached.

The process of debugging a stored procedure from Visual Basic or Java clients running on the workstation is a bit more tricky. Refer to 3.10, “Testing and debugging” on page 38, for details.

### 6.6 Moving into production (save and restore)

While deploying a database application to a production system, you need to save and restore objects, such as external programs, that were registered as stored procedures. Depending on the type of stored procedure and external program that implements that stored procedure, there may be additional actions required to make it available on the target system.

When `CREATE PROCEDURE` is used to create an external stored procedure associated with an ILE external program, an attempt is made to save the stored procedure's attributes in the associated program object. If the *PGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes, subject to the following restrictions:

- The external program library must not be QSYS or QSYS2.
- The external program must be an ILE *PGM object.
- The external program must contain at least one SQL statement.

If the external program object is an OPM or an ILE with no embedded SQL statements, you need to register the program as a procedure with the `CREATE PROCEDURE` statement. This does not mean that the SQL Development Kit has to be installed on the target system. You can also use one of the following SQL interfaces:

- iSeries Navigator
- WRKQMQRY
- RUNSQLSTM CL command
- Program that uses ODBC, JDBC, or CLI interfaces

If the external object is an ILE program and contains at least one SQL statement, DB2 Universal Database for iSeries automatically tries to register the appropriate stored procedure. However, there are several topics that require your attention:

- The external procedure is registered into the target library specified by the restore command, regardless of the `CREATE PROCEDURE` source.
- If the matching external procedure is not found in catalogs, DB2 Universal Database for iSeries automatically registers the program as a stored procedure.
- If the matching external procedure is found in catalogs with the same signature (same number of parameters), the program is restored, but the catalog information is not updated. This works fine as long as the `CREATE PROCEDURE` statement did not change. However, if the statement changed, it is your responsibility to execute the `DROP PROCEDURE` statement.
- If the matching external procedure is found in catalogs and the signature is different (meaning different parameters), the program object is restored and the catalog information is updated.
In general, you should be careful when restoring a stored procedure with a modified number or type of parameters because the restore operation may overlay the program object for an existing procedure, which happens to share the same signature. If the procedure already exists in the target library or system and the number of parameters or data type of the parameters has changed, then a drop procedure statement should be performed prior to the restore operation.

### 6.7 The Order Entry application: Stored procedures examples

In our application scenario, we use a stored procedure to perform some composite operations on the inventory database file at the remote site. When the client program inserts the items for the order, the remote inventory is referenced for every item in the order, and the available quantity for the product is updated at the remote site. If the product we are requesting is not available, the inventory is searched for a replacement. If the inventory lookup and update completed successfully, a record is inserted in the local order detail file. Finally, the transaction is committed to free the inventory record from the locks. In this scenario, we exploit DB2 Universal Database for iSeries two-phase commit.

We decided to implement the inventory lookup and update through a stored procedure, since the search for an alternative item can involve quite a lot of database accesses on the remote inventory file. This processing can be carried out entirely at the server site where the general inventory resides. The stored procedure accepts these parameters:

- **PARM1**: INOUT, for product number (PRDNBR)
- **PARM2**: IN, for ordered quantity (ORDQTA)
- **PARM3**: OUT, for product description (PRDDES)
- **PARM4**: OUT, for product price (PRDPRC)

The stored procedure scans the inventory for replacements if the original item is not available. If no replacement can be found, the product description (PARM3) is returned as NULL. The stored procedure has been defined in the CREATE PROCEDURE statement as SIMPLE CALL WITH NULLS. The SIMPLE CALL WITH NULLS clause is synonymous with GENERAL WITH NULLS and is provided for the compatibility with other platforms.

Even though four parameters are specified in the CALL statement of our calling program (INSDET), the stored procedure (STORID) has to define five parameters for an additional indicator array parameter that is passed due to defining SIMPLE CALL WITH NULLS on the DECLARE PROCEDURE statement. In both INSDET and STORID programs, the additional indicator variable parameter is defined as a data structure (INDDS) with four two-digit binary subfields: IND1, IND2, IND3, and IND4. However, we are using only IND1 and IND3 in our program.

If the stored procedure finds the corresponding product record and its quantity is sufficient, it is updated, and its description and price are passed back through PARM3 and PARM4. The indicators IND1 and IND3 are set to zero.

If no record can be found, IND1 is set to -1, which means PARM1 has a NULL value. When returning to the calling program, this indicator should be tested. If it has a value of -1, an error message is displayed.

If the item is found but its quantity is not sufficient, an alternative with the same product category is searched. We assume that only the first match is passed back to the calling program even though there may be several alternatives that have the required quantity in stock. The alternative product description and price are passed back to the calling program.
If there is no alternative with the available quantity, the IND3 is set to -1. When returning to the calling program, the indicator is checked, and another error message is displayed in this case.

After the item is found and the order detail row is inserted in the local file, the local and remote changes can be committed or rolled back by the client application, exploiting two-phase commit.

See Chapter 2, “Stored procedures, triggers, and user-defined functions: Order entry application” on page 11, for details about our application scenario.

6.7.1 Calling a stored procedure

The listing in Example 6-11 is an RPG version of the Insert Detail (INSDET) client program with embedded SQL statements.

Example 6-11  RPG version of the Insert Detail client program

****************************************************************
*                                                              *
*   T4249RIDT   Program Overview:                              *
*   ---------   Insert Order Detail Items                      *
*               in Order Entry Application                     *
*                  using DRDA-2                                *
*                    2-Phase Commit                        *
*                    Stored Procedure                      *
*                    RI on Order Detail Table              *
*                                                              *
****************************************************************

*                                                              *
*   This program takes input from previous program T4249CINS:  *
*      customer number, order number. Order items are entered  *
*      from screen: product number, quantity.                  *
*      DRDA-2 connection is established to a remote system to  *
*      call a stored procedure, which updates the STOCK file   *
*      located at the head office (remote system) with the      *
*      ordered quantity. If quantity is not available, an      *
*      alternative item is searched. If there is no alternative *
*      available, this is indicated with a negative value in    *
*      the corresponding indicator variable of the parameter.  *
*      If quantity could be updated, connection is established *
*      to the local system for inserting an order detail record. *
*      Referential integrity rules check insert violation (see *
*      documentation before coding statements below).           *
*      A Distributed Unit of Work (DUW) in this program (using  *
*      DRDA-2) includes an update on the remote system by the    *
*      stored procedure and a record insert on the local system. *
*      This DUW can be rolled back, using PF keys. If not,      *
*      2-phase commitment control commits the logical transact.  *
*                                                              *
****************************************************************

*                                                              *
*   ZURICH  is local database system                           *
*   ROCHESTER  is remote database system                       *
*                                                              *
****************************************************************

*                                                              *
*   Indicator usage:                                           *
*                                                               *
*             03  F03 Exit/Finalize Order                      *
*             11  F11 Cancel Item                             *
*             12  F12 Cancel Order                            *
* 22 2nd EXFMT with info.  *
* 51 1st CONNECT local    *
* 63 Duplicate product ordered *
* 64 RI Constraint       *
* 65 Product not found   *
* 66 No alternative     *
* 67 This is an alternative *
* 77 To bypass first commit statement *
* *
*=================================================================================================* 
FINSDETD CFE WORKSTN
* 
*=================================================================================================* 
* These are the fields for the indicator variables, used by the stored procedure, to indicate the content of the passed parameters: 
*=================================================================================================* 
* 
IINDDS DS
I          B 1  20IND1
I          B 3  40IND2
I          B 5  60IND3
I          B 7  80IND4
* 
*=================================================================================================* 
* The following parameters are input from the previous program (ORDHDR), which created the order header data, and output to the next programs: finalize order, cancel order or main: 
*=================================================================================================* 
* 
C   *ENTRY   PLIST
C       PARM CUSNBR  5 Customer #
C       PARM ORDNBR 5 Order #
C       PARM DODCUM 112 Order cumulativ
C       PARM RTNCDE 1 Return code
* 
C   Z-ADD0  DODCUM
C       MOVE '0' RTNCDE
C       MOVE '0' *IN77
* 
C   BEGIN   TAG
* 
*=================================================================================================* 
* The following 2-Phase Commit for local and remote system is only executed, if the conditions are met, according to the indicators described above. 
* The first COMMIT after one DUW therefore is done only after the ordered quantity has been deducted from STOCK file on the remote system and the first order record has been inserted correctly in the ORDERDTL file on the local system. 
* Every item record for an order is committed, because of releasing the record lock on STOCK file. 
* Note: SQL COMMIT in the program starts commitment control for the activation group automatically: 
*=================================================================================================* 
* 
C   *IN11   IFEQ '0'
C   *IN12   ANDEQ'O'
C   *IN64   ANDEQ'O'
C           *IN65     ANDEQ'0'
C           *IN77     ANDEQ'1'
C/EXEC SQL
C+ COMMIT
C/END-EXEC
C                     END
C                     MOVE '1'       *IN77
C                     MOVE *BLANK    DPRDNR
C                     Z-ADD0         DQUANT
C                     EXFMTINSDT1                     1st display
C           *IN03     IFEQ '1'
C                     GOTO SETLR
C                     END
C                     MOVE '0'       RTNCDE
C                     MOVE '0'       RTNCDE
****************************************************************
*  -- Connection to the REMOTE database: --                    *
*  At start of the program DRDA connection mgmt. establishes   *
*  connection automatically to the remote sys. according to    *
*  the relational database specified in the compil. parameter  *
*  in command CRTSQLxxx RDB(....). Therefore this program      *
*  is connected to remote database ROCHESTER already.          *
*  For further remote re-connections SET CONNECTION is used:    *
****************************************************************
C*
C/EXEC SQL
C+ SET CONNECTION ROCHESTER                         -- After 1.conn
C/END-EXEC                                              remote
C/END-EXEC
****************************************************************
*  The CALL of the stored procedure (at the remote system)     *
*  is prepared. The indicator variables are set to zero:       *
****************************************************************
C                     MOVE DPRDNR    PARM1   5        Prod. #
C                     MOVE *ZERO     IND1             -1=no prod fnd
C                     MOVE DQUANT    PARM2   50       Ordered quant
C                     MOVE *BLANK    PARM3  20        Prod descriptio
C                     MOVE *ZERO     PARM4   72       Prod price
C                     MOVE *ZERO     PARM4   72       Prod price
****************************************************************
*  The stored procedure is declared, which is optional.        *
*  A performance advantage is gained by doing so:              *
****************************************************************
C/EXEC SQL
C+ DECLARE P1 PROCEDURE (:PARM1 INOUT CHAR (5), :PARM2 IN DEC (5,
C+ 0), :PARM3 OUT CHAR (20), :PARM4 OUT DEC (7, 2))(EXTERNAL NAME
C+ ORDENTLIB/STORID LANGUAGE RPG SIMPLE CALL WITH NULLS)
C/END-EXEC
****************************************************************
*  The stored procedure is called at the remote system,        *
*  because connection to remote (DB) system is established:    *
****************************************************************
C
C/EXEC SQL
C+  CALL P1(:PARM1:IND1, :PARM2, :PARM3:IND3, :PARM4)
C/END-EXEC
*
******************************************************************************
* Return from stored procedure. The indicator variables * *
* for PARM1 and PARM3 contain -1, if no data can be passed * *
* from the stored procedure to this calling program. * *
* This is checked in the following statements: * *
******************************************************************************
*
C           IND1      IFLT *ZERO                      Item not found
C                     MOVE '1'       *IN65
C                     MOVE *BLANK    PARM3
C                     MOVE *ZERO     PARM4
C                     ELSE
C           IND3      IFLT *ZERO                      No alternative
C                     MOVE '1'       *IN66
C                     MOVE *BLANK    PARM3
C                     END
C                     END
*
C           PARM1     IFNE DPRDNR                     Alternat. item
C                     MOVE '1'       *IN67
C                     END
*
******************************************************************************
*  Product details are moved from parameters of the stored     *
*  procedure to field variables of this calling program:       *
******************************************************************************
*
C                     MOVE PARM1     DPRDNR           Product #
C                     MOVE PARM3     DDESCR           Description
C                     MOVE PARM4     DPRICE           Price
*
******************************************************************************
*  -- Connection to the LOCAL database: --                     *
*  At this point, connection to the local database is estab-   *
*  lished. For the first time in the execution of the program *
*  the CONNECT statement has to be executed.                  *
*  The connection to the local database then goes to dormant   *
*  state, after connecting to the remote DB (above) again.     *
*  For further local re-connections SET CONNECTION is used:    *
******************************************************************************
*
C           *IN51     IFEQ '0'                        1st Connect
C/EXEC SQL                                        --     Local
C+ CONNECT TO ZURICH
C/END-EXEC
C                     MOVE '1'       *IN51            After 1.Conn
C                     ELSE                                Local
C/EXEC SQL
C+ SET CONNECTION ZURICH
C/END-EXEC
C                     END
C           *IN65     IFEQ '0'                        Item found
C                     *IN66     ANDEQ'0'                        Altern avail
C                     DPRICE    MULT DQUANT    DITTOT           Item total
C                     DITTOT    ADD  DODCUM    DODCUM           Cumulative
*
* An order detail record is inserted in the local database, if referential integrity rules are not violated, i.e. the primary key of ORDERDTL file must be unique, and/or a corresponding order number must exist in the ORDERHDR parent file. Otherwise an SQL error message is sent from database management:

```
C/EXEC SQL
C+ INSERT INTO ORDENTL/ORDERDTL (ORHNBR, PRDNBR, ORDQTY, ORDTOT)
C+ VALUES(:ORDNBR, :DPRDNR, :DQUANT, :DITTOT)
C/END-EXEC
```

* If primary key constraint is detected (duplicate key) update of order quantity in STOCK file on remote system is rolled back by 2-phase commitment control management:

```
C/EXEC SQL
C+ ROLLBACK
C/END-EXEC
```

* If ORDERHDR parent file does not have corresponding order number (RI rule violated), update of order quantity in STOCK file on remote system is rolled back by 2-phase commitment control management:

```
C/EXEC SQL
C+ ROLLBACK
C/END-EXEC
```
C        EXSR SUBTR
C        MOVE '1'       RTNCDE
*
****************************************************************
*  If customer does not agree with alternative item, the order *
*  item can be cancelled by pressing PF11.                    *
*  The update of order quantity in STOCK file on remote system *
*  and the insert of the record in ORDERDTL file on local sys. *
*  is rolled back by 2-phase commitment control management:    *
****************************************************************
*
C/EXEC SQL
C+ ROLLBACK
C/END-EXEC
C        END
*
C        *IN12     IFEQ '1'                        Cancel order
*
****************************************************************
*  If customer does not agree with alternative item, the whole *
*  order can be cancelled by pressing PF12.                    *
*  A cancel order program is called by the main program,       *
*  according to the return code passed on RETRN. The called *
*  program deletes one record in ORDERHDR file, one or more    *
*  records in ORDERDTL file, and updates the STOCK file        *
*  accordingly.                                                *
****************************************************************
*
C        MOVE '1'       RTNCDE
*
****************************************************************
*  The update of order quantity in STOCK file on remote system *
*  and the insert of the record in ORDERDTL file on local sys. *
*  are rolled back by 2-phase commitment control management    *
*  for the last item entered:                                 *
****************************************************************
*
C/EXEC SQL
C+ ROLLBACK
C/END-EXEC
C        RETRN
C        END
*
C        GOTO BEGIN
*
C        SETLR     TAG
*
****************************************************************
*  If PF3 is pressed, order entry has finished. All            *
*  connections are released in order to save on resources:     *
****************************************************************
*
C/EXEC SQL
C+ RELEASE ALL
C/END-EXEC
*
****************************************************************
*  The following COMMIT statement activates previous RELEASE:  *
****************************************************************
*
6.7.2 Sample stored procedure: SQL RPG version

The code listing in Example 6-12 shows the SQL RPG version of the stored procedure.

Example 6-12  SQL RPG version of the stored procedure

The code listing in Example 6-12 shows the SQL RPG version of the stored procedure.
C                     MOVE *BLANK WCAT  4
C*
C* If the product number is not found, the program sets
C* IND1 to "-1", which means "NULL", and it is checked
C* by the calling program when it returns.
C* And the other parameters have to be cleared in order to
C* prevent the previous values from being passed back to the
C* calling program.
C*
C/EXEC SQL
C+ DECLARE ALT0 CURSOR FOR
C+ SELECT PRDDES,PRDPRC,PRDQTA,PRDCAT
C+ FROM ORDENTR/STOCK
C+ WHERE PRDNBR = :PARM1
C+ FOR UPDATE OF PRDQTA
C/END-EXEC
C*
C/EXEC SQL
C+ OPEN ALT0
C/END-EXEC
C*
C/EXEC SQL
C+ FETCH ALT0 INTO :PARM3, :PARM4, :WQTA, :WCAT
C/END-EXEC
C*
C           SQLCOD    IFEQ 100
C                     MOVE -1        IND1
C                     MOVE *ZERO     IND3
C                     MOVE *BLANK    PARM3
C                     MOVE *ZERO     PARM4
C                     GOTO DONE0
C                     END
C*
C* If the available quantity is enough to meet the ordered
C* quantity, the product record is updated and the product
C* information is passed back to the calling program.
C* All indicator variables are set to *ZERO.
C*
C           WQTA      IFGE PARM2
C           WQTA      SUB  PARM2     LEFTQ   50
C/EXEC SQL
C+ UPDATE ORDENTR/STOCK SET PRODUCT_AVAIL_QTY = :LEFTQ
C+ WHERE CURRENT OF ALT0
C/END-EXEC
C                 MOVE *ZERO     IND1
C                 MOVE *ZERO     IND3
C                 GOTO DONE0
C                 END
C*
C* If there is not enough quantity for the original product
C* an alternative within same product category number is
C* searched.
C* We assume that we'll pass back only the first one to the
C* calling program even though there may be several alternatives
C* in the STOCK file.
C*
C/EXEC SQL
C+ DECLARE ALT1 CURSOR FOR
C+ SELECT PRDNBR,PRDDES,PRDPRC,PRDQTA
C+ FROM ORDENTR/STOCK
6.8 External stored procedure using service program

New in V5R3, you can declare an external stored procedure that uses an OS/400 service 
program object. This object provides more deployment flexibility for external stored 
procedures. We give you an example in this section on how to exploit this new support.

In this section, we use example program codes (see Example 6-13) from 3.5.4, “Using an ILE 
Service Program,” of the IBM Redbook Who Knew You Could Do That with RPG IV? A 
Example 6-13  Source codes of the NameOfDay and DayOfWeek subprocedures

* NameOfDay and DayOfWeek subprocedures
*------------------------------------------------------------------
H Nomain

* Prototype for subprocedure DayOfWeek
D DayOfWeek     PR             1  0
D InputDate                      D   Datfmt(*ISO)

* Prototype for subprocedure DayName
D NameOfDay       PR
D InputDate                      D   Datfmt(*ISO)
D DayName                       9A

* Days of the week name table - note no field names are required
D NameData        DS
D                                9    Inz('Monday')
D                                9    Inz('Tuesday')
D                                9    Inz('Wednesday')
D                                9    Inz('Thursday')
D                                9    Inz('Friday')
D                                9    Inz('Saturday')
D                                9    Inz('Sunday')

* Define the array as an overlay of the DS name
D Name                          9    Dim(7) Overlay(NameData)

* SubProcedure: NameOfDay (Name of the Day)
* The subprocedure accept a valid date (format *ISO) and return
* a string representing the name of the day
P NameOfDay       B                   Export
D NameOfDay       PI
D  WorkDate                       D   Datfmt(*ISO)
D  DayName                       9A
C                   EVAL      DayName = Name(DayOfWeek(Workdate))
C     Return    Name(DayOfWeek(Workdate))
C     Return    DayName

* SubProcedure: DayOfWeek (Day of the Week)
* The subprocedure accept a valid date (format *ISO) and return
* a number (1 digit) representing the day of the week
* (Monday = 1, ... , Sunday = 7)
P DayOfWeek       B                   Export
D DayOfWeek       PI             1  0
D  WorkDate                       D   Datfmt(*ISO)
C     WorkDate      Subdur    AnySunday     WorkNum:*D
C     WorkNum       Div       7             WorkNum
C     Mvr                     WorkDay
These example program codes are of two subprocedures named NameOfDay (1) and DayOfWeek (3A). The subprocedure NameOfDay is the entry point of the external stored procedure that you will create. This means that, from the external procedure, you make a program call to the NameOfDay subprocedure (1) together with a value for its input parameter WorkDate (2), which is a text string of an ISO date format, for example, 'yyyy-mm-dd'.

Then the subprocedure NameOfDay internally calls another subprocedure named DayOfWeek (3 and 3A). It also receives the WorkDate input parameter and calculates a return value as a pointer for the Name variable (4A) to help select the name of day from the data structure NameData (4). The resulting name of day text string is then passed to the output parameter DayName (3 and 5).

With these source codes and assuming that all objects are created in the library named SQLLIB, you use the following CL commands to create the service program object:

```cl
CRTRPGMOD MODULE(SQLLIB/NAMEOFDAY) SRCFILE(SQLLIB/QRPGLESRC) SRCMBR(NAMEOFDAY)
CRTSRVPGM SRVPGM(SQLLIB/NAMEOFDAY) EXPORT(*ALL)
```

You now can register the service module as an external stored procedure:

```sql
CREATE PROCEDURE SQLLIB.NAMEOFDAY (
    IN WorkDate DATE, OUT DayName CHARACTER(9))
    LANGUAGE RPGLE
    SPECIFIC SQLLIB.NAMEOFDAY
    NOT DETERMINISTIC
    NO SQL
    CALLED ON NULL INPUT
    EXTERNAL NAME 'SQLLIB/NAMEOFDAY(NAMEOFDAY)'
    PARAMETER STYLE GENERAL WITH NULLS;
```

Notice at line A that SQLLIB/NAMEOFDAY represents the service program object while (NAMEOFDAY) is the entry point name that represents the subprocedure NameOfDay that is to be invoked.

After the external stored procedure is declared, you can use Run SQL Scripts utility to try the procedure call with the output parameter displayed in the Messages tab of the window. Notice that the output parameter DayName (B) must be specified with a question mark (?) value when the procedure is called as shown in the following example and in Figure 6-8:

```sql
CALL SQLLIB.NAMEOFDAY('2005-11-25', ?) ;
```
6.9 RPG IV example for external stored procedure

In this section, we give you examples of RPG IV programs that write to and read from a data queue, which you can registered as external stored procedures. These RPG programs use the OS/400 APIs SNNDDTAQ and RCVDTAQ. For a detailed description of these APIs, see 5.2.2, “List of data queue APIs,” of the redbook Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More, SG24-5402. The example codes in the following sections are from 5.2.3, “Programming with data queue APIs,” of the same redbook.

Let us assume that all program objects and data queue are created in a library named SQLLIB and the data queue name is DTAQFIFO for this example.

First, you create a data queue (with a 40-byte maximum length) with the following CL command:

```
CRTDTAQ DTAQ(SQLLIB/DTAQFIFO) MAXLEN(40)
```

Now, you create the programs and register them as external stored procedures as explained in the following section.

6.9.1 External stored procedure that writes to a data queue

You create a bound RPG program object from the program code shown in Example 6-14.

Example 6-14  Program example WRTDTAQ for writing a text string to a data queue

******************************************************************************
* Prototype for API QSNDDTAQ - Send To a Data Queue                     *
D SndDtaQ         PR                  EXTPGM('QSNDDTAQ')              *
D  DataQueueNam   10A   Const      *
D  DataQueueLib   10A   Const      *
D  DataLength     5P 0 Const      *
D  DataBuffer     32767A Const Options(*Varsize) *
******************************************************************************
* Prototype definitions                                                 *
D WRTDTAQ         PR                   *
D  DQname          10A  
D  DQlib           10A  
D  DataSnd         40A  

* Program variable definitions                                          *
D WRTDTAQ         PI                   *
D  DQname          10A  
D  DQlib           10A  
D  DataSnd         40A  

* Write data to data queue names DTAQFIFO in library S QLLIB              *
C                   CallP     SndDtaQ(DQname : DQlib : %Len(%Trim(DataSnd)) : DataSnd)  *
C                   Eval      *InLR = *On

This program takes three input parameters:

- The data queue name (DQname at 1)
- The data queue library (DQlib at 2)
- The text string to be written to the data queue (DataSnd at 3 - 40 bytes maximum)

With these source codes, you create a program object of this example, named WRTDTAQ, using the following command:

CRTBNDRPG PGM(SQLLIB/WRTDTAQ) SRCFILE(SQLLIB/XXXX) DFTACTGRP(*NO) ACTGRP(QILE)

After the program object is created, you register it as an external stored procedure by using the following SQL statement:

CREATE PROCEDURE SQLLIB.WRTDTAQ (   
  IN DQNAME CHAR(10) , IN DQLIB CHAR(10) , IN TEXTSTRING CHAR(40) )   
LANGUAGE RPGLE   
SPECIFIC SQLLIB.WRTDTAQ   
NOT DETERMINISTIC   
NO SQL   
CALLED ON NULL INPUT   
EXTERNAL NAME 'SQLLIB/WRTDTAQ'   
PARAMETER STYLE GENERAL WITH NULLS ;

We now move to the second program that read from the data queue.
### 6.9.2 External stored procedure that reads from a data queue

You create a bound RPG program object from the program code shown in Example 6-15.

#### Example 6-15 Program example RDDTAQ for reading a text string from a data queue

Listing:

```
*********************************************************************
* Prototype for API QRCVDTAQ - Received From a Data Queue
DRcvDtaQ     PR                  EXTPGM('QRCVDTAQ')
D  DataQueueNam                 10A   Const
D  DataQueueLib                 10A   Const
D  DataLength                    5P 0
D  DataBuffer                32767A         Options(*Varsize)
D  WaitTime                      5P 0 Const
*--------------------------------------------------------------------
* Prototype definitions
DRDDTAQ           PR
D  DQname                       10A
D  DQlib                        10A
D  Output                       40A
*
* Program variable definitions
DRDDTAQ           PI
D  DQname                       10A
D  DQlib                        10A
D  Output                       40A
D  DataRcv        S             40A
D  Length         S              5P 0
D  WaitTime       C                   5
*--------------------------------------------------------------
* Read from datat queue DTAFIFO in library SQLLIB
*
C                   CallP     RcvDtaQ( DQname : DQlib : Length
C                                      : DataRcv : WaitTime )
C                   eval      Output = DataRcv
C                   Eval      *InLR = *On
```

This program takes two *input* parameters, Data queue name (DQname at A) and Data queue library (DQlib at B). The text string read from the data queue (DataRcv at D1 and D2) is passed to the *output* parameter (Output at C1 and C2).

You create a program object of this example, named RDDTAQ, using the following command:

```
CRTBNDRPG PGM(SQLLIB/RDDTAQ) SRCFILE(SQLLIB/XXXX) DFTACTGRP(*NO) ACTGRP(QILE)
```

After the program object is created, register it as an external stored procedure by using the following SQL statement:

```
CREATE PROCEDURE SQLLIB.RDDTAQ (  
   IN DQNAME CHAR(10), IN DQLIB CHAR(10), OUT OUTPUTTEXT CHAR(40) )  
   LANGUAGE RPGLE  
   SPECIFIC SQLLIB.RDDTAQ  
   NOT DETERMINISTIC  
   NO SQL  
   CALLED ON NULL INPUT  
   EXTERNAL NAME 'SQLLIB/RDDTAQ'  
   PARAMETER STYLE GENERAL WITH NULLS ;
```

You can now use the registered external stored procedures WRTDTAQ and RDDTAQ.
6.9.3 Calling external stored procedures from Run SQL Scripts utility

You can call these sample external stored procedures to test them by issuing the CALL statement from Run SQL Scripts utility of iSeries Navigator:

```
CALL SQLLIB.WRTDTAQ('DTAQFIFO', 'SQLLIB', 'Hello data queue 1');
CALL SQLLIB.RDQDTAQ('DTAQFIFO', 'SQLLIB', '?');
```

This utility can display the value of the output parameter of the stored procedure for you in its Messages tab as shown in Figure 6-9.

```
CALL SQLLIB.WRTDTAQ('DTAQFIFO', 'SQLLIB', 'Hello data queue 1');
CALL SQLLIB.RDQDTAQ('DTAQFIFO', 'SQLLIB', '?');
>
Return Code = 0
Statement ran successfully (361 ms)
>
Return Code = 0
Output Parameter #3: Hello data queue 1
Statement ran successfully (370 ms)
```

*Figure 6-9 Calling external stored procedures WRTDTAQ and RDQDTAQ and their results*

Note the following points:

- The data queue name and its library name must be entered in uppercase.
- The output parameter of RDQDTAQ procedure must be specified with the NULL value when called. You can type the question mark (?) to represent a NULL value.

**Note:** This is a useful technique to use an external procedure to access a non-SQL OS/400 object from an SQL stored procedure.
Java stored procedures

This chapter describes the Java stored procedures implementation on the iSeries server. It also compares the support provided by the iSeries server with the implementations on other DB2 platforms. The combination of Java and a powerful database server, such as the DB2 Universal Database for iSeries, can result in a scalable and robust software solution. From the implementation point of view, Java stored procedures are simply another type of external stored procedure.

We expect that you are already familiar with Java, especially since this book provides a detailed description on Java stored procedure implementation. For a more detailed description on how to use Java on the iSeries server, refer to Building Java Applications for the iSeries Server with VisualAge for Java, SG24-6245.

The current implementation of Java stored procedures on the iSeries server aims at providing the same level of support as the other DB2s. Refer to 7.10, “GetSuppliers example: Implementation with result sets” on page 211, for more information about Java stored procedures on the DB2 Universal Database platform.

This chapter focuses on the four fundamental tasks involved in the creation and using a stored procedure:

- How to code it
- How to deploy it in the database
- How to define it as a stored procedure
- How to call it

It illustrates each step with simple examples and discusses some problem determination tools. This chapter also outlines the major considerations for current implementation on the iSeries server. Finally, it provides a more elaborate example of a stored procedure called GetSupplier.
7.1 Prerequisites

There are two requirements for Java stored procedure support on the iSeries server:

- 5769-SS1 V4R5M0 Operating System/400 or above
- 5769-JV1 V4R5M0 Java Developer Kit or above

We strongly recommend that you install the latest database fixpack (SF99105) that is available for your system.

7.2 Coding DB2 Universal Database for iSeries Java stored procedures

A Java stored procedure corresponds to a method in a Java class that meets some requirements explained in this section.

Before we start with a detailed explanation on how to code this method, we need to discuss briefly two different parameter styles supported on the DB2 Universal Database for iSeries. The parameter style chosen for a Java stored procedure determines how the Java method exchanges parameters with the SQL runtime and how result sets are returned to the SQL runtime. It also has important implications for the portability of your code.

7.2.1 Parameter styles

DB2 Universal Database for iSeries supports two parameter styles for Java stored procedures:

- JAVA: Conforms with the SQL3 standard
- DB2GENERAL: Is provided for compatibility with other DB2 Universal Database platforms

Java parameter style

In this parameter style, the Java method must be defined as a public static void method. The following Java code template illustrates a Java class with two methods that we define in a later step as two Java stored procedures:

```java
public class SomeStoredProcs {
    public static void myStoredProc(...) {
        // SP implementation
    }
    public static void anotherStoredProc(...) {
        // SP implementation
    }
}
```

This first example shows that a single Java class can contain the implementation of many stored procedures. If one of its methods is not defined as public static void, this method cannot be used as a Java stored procedure.

Stored procedures can use input, output, and inout parameters. Unfortunately, Java does not have the notion of output or inout parameters. Therefore, the convention adopted is that output (and inout) parameters are represented in the Java method by an array of size one.
This convention and the way result sets are returned are the main characteristic of Java parameter style stored procedures. The following example illustrates how to define the methods corresponding to stored procedures with input, output, and inout parameters. For illustration purposes, we placed the various types of parameters in distinct methods. In a real life scenario, a stored procedure and its corresponding method are likely to use a mix of several types of parameters (IN, INOUT, and OUT):

```java
public class SomeStoredProcs {
    public static void myInputSP(String myInputP1, int myInputP2) {
        // some code
    }

    public static void myOutputSP(String[] myOutputP1, int[] myOutputP2) {
        // some code
        myOutputP1[0] = "This is a string that will be returned as output parameter";
        myOutputP2[0] = 1; // this returns an integer value
    }

    public static void myInoutSP(String myInoutP1, int myInoutP2) {
        String receivedString = myInoutP1;
        int receivedInt = myInoutP2;
        // some more code
        myInoutP1[0] = "This is a string that will be returned as output parameter";
        myInoutP2[0] = 1; // or whatever value
    }
}
```

The type of parameters you choose for your procedure is very important because you have to choose a Java data type that can be mapped to an SQL data type. Otherwise, your Java method is simply ignored by the database. Refer to Table 7-1 on page 177 for more details regarding the compatibility of data types.

The SQL NULL value can be passed to a stored procedure only when the corresponding Java data type can have a null reference value. This is the case for String, byte[], BigDecimal, Date, Time, TimeStamp, Double, Float, Integer, and Long, but not for the scalar data types: boolean, byte, short, int, long, float, and double. You can pass an SQL NULL value to a stored procedure if the corresponding Java data type is, for example, a String, but not if it is an int. In the implementation of Java stored procedures on the DB2 platform, it is currently impossible to test if, for example, an SQL INTEGER parameter passed to the Java stored procedure using the Java parameter style is NULL. The SQLJ standard describes an optional feature to circumvent this problem, but it is not currently implemented on the DB2 platform. With the DB2GENERAL parameter style, it is possible to test variables of any type to see if their value is NULL.

For portability reasons, you should consider using the Java parameter style because it is supported by most database platforms.

**Note:** The binary large objects (CLOB, BLOB, DBCLOB) cannot be passed as parameters of the Java stored procedure in this implementation of the Java parameter style on the iSeries server.

**DB2GENERAL parameter style**

This parameter style is specific to the DB2 platform and is not a part of the JDBC or SQLJ standard. It was first introduced in Version 5 of DB2 Universal Database.

**Note:** The stored procedure interface provided by iSeries Navigator to define stored procedures does not support the DB2GENERAL parameter style.
The class that contains one or more methods that will be defined as stored procedures must extend the `com.ibm.db2.app.StoredProc` class. This class is provided in the `db2routines_classes.jar` file located in the IFS in the `/QIBM/ProdData/OS400/Java400/ext` directory. The method corresponding to the Java stored procedure must be defined, as a public void method as illustrated here:

```java
public class SomeStoredProcs extends StoredProc {
    public void myStoredProc(...) {
        // SP implementation
    }
    public void anotherStoredProc(...) {
        // SP implementation
    }
}
```

The output parameters are declared like the input parameters (no array convention used), but they must be set by the `set()` method of the `StoredProc` class before the method returns as shown in the following code example:

```java
public class SomeStoredProcs extends StoredProc {
    public static void myInputSP(String myInputP1, int myInputP2) {
        // SP implementation
    }
    public static void myOutputSP(String myOutputP1, int myOutputP2) {
        // some code
        set(1, "This is a string that will be returned as output parameter");
        set(2, 1); // this returns an integer value
    }
    public static void myInOutSP(String myInOutP1, int myInOutP2) {
        String receivedString = myInOutP1[0];
        int receivedInt = myInOutP2[0];
        // some code
        set(1, "This is a string that will be returned as output parameter");
        set(2, 1); // this returns an integer value
    }
}
```

The Java data types used for the parameters must be compatible with SQL data types. Refer to Table 7-1 for more details regarding data type compatibility.

The `DB2GENERAL` parameter style allows you to check whether the value of any parameter received by the stored procedure is SQL NULL with the `isNull()` method provided in the `StoredProc` class.

The LOB, BLOB, and CLOB data can be accessed directly through the `com.ibm.db2.app.Lob`, `com.ibm.db2.app.Blob`, and `com.ibm.db2.app.Clob` classes contained in the same `db2routines_classes.jar` file as the `StoredProc` class.
7.2.2 Data type compatibility

Table 7-1 summarizes the compatibilities between the SQL and Java data types depending on the parameter style convention used.

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Java type (Java parameter style)</th>
<th>Java type (DB2GENERAL parameter style)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short (Not nullable)</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int (Not nullable)</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long (Not nullable)</td>
<td>long</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>BigDecimal (Nullable)</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>NUMERIC(p, s)</td>
<td>BigDecimal (Nullable)</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>REAL or FLOAT(p)</td>
<td>float (Not nullable)</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE PRECISION or FLOAT or FLOAT(p)</td>
<td>double (Not nullable)</td>
<td>double</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>String (Nullable)</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>String (Nullable)</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR(n) FOR BIT DATA</td>
<td>-</td>
<td>com.ibm.db2.app.Blob</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>String (Nullable)</td>
<td>String</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>String (Nullable)</td>
<td>String</td>
</tr>
<tr>
<td>DATE</td>
<td>Date (Nullable)</td>
<td>String</td>
</tr>
<tr>
<td>TIME</td>
<td>Time (Nullable)</td>
<td>String</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Timestamp (Nullable)</td>
<td>String</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>com.ibm.db2.app.Clob</td>
</tr>
<tr>
<td>BLOB</td>
<td>-</td>
<td>com.ibm.db2.app.Blob</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>com.ibm.db2.app.Clob</td>
</tr>
</tbody>
</table>

7.2.3 Database connection in a Java stored procedure

The Java code can interact with the database like any other Java program, but the database connection can only be established with the local database using the current user ID. Both the JDBC and SQLJ interfaces can be used to access the database.

**JDBC**

The method of connecting to the database depends on the parameter style convention used:

- **JAVA**: The JDBC connection object is instantiated by the default DriverManager. The regular Java applications must load the DriverManager before they can instantiate a connection to the database. However, in a Java stored procedure, a default DriverManager is provided by the database runtime. The syntax is shown here:
  ```java
  Connection con = DriverManager.getConnection("jdbc:default:connection");
  ```
DB2GENERAL: JDBC creates the connection object to the DB2 Universal Database for iSeries database with the getConnection() method provided by StoredProc. The following code example illustrates the syntax:

```java
Connection con = getConnection();
```

**SQLJ**

In the case of SQLJ, the notion of a connection is replaced by that of a context. Here, it is also necessary to make the distinction between the two parameter styles:

- **Java:** Use the ExecutionContext class as shown in the following example:

  ```java
  ExecutionContext ec = DefaultContext.getDefaultContext().getExecutionContext();
  ```

- **DB2GENERAL:** You must explicitly create the DefaultContext object with the getDefaultContext method, as shown in the following code example:

  ```java
  DefaultContext ctx = DefaultContext.getDefaultContext();
  if (ctx == null) {
    Connection con = getConnection();
    ctx = new DefaultContext(con);
    DefaultContext.setDefaultContext(ctx);
  }
  ```

### 7.2.4 Returning result sets in Java stored procedures

Starting in V5R1, Java stored procedures in DB2 Universal Database for iSeries now support returning result sets. The way in which result sets are declared and coded in the stored procedure differs significantly depending on the parameter style used.

#### Returning result sets in Java parameter style stored procedures

In stored procedures with Java parameter style, result sets are declared as output parameters at the end of the parameter list. The number of result sets must be explicitly declared in the CREATE PROCEDURE (or DECLARE PROCEDURE) statement. A stored procedure that receives an integer parameter, returns a string parameter, and returns two result sets will be declared to DB2 Universal Database for iSeries as follows:

```
CREATE PROCEDURE SPWITHTWOSENTRIES (IN INTEGER, OUT VARCHAR(50))
EXTERNAL NAME MyClass!myJavaStoredProcedure
PARAMETER STYLE JAVA
RESULT SETS 2
LANGUAGE JAVA
```

The Java method implementing this stored procedure looks like this example:

```java
import java.sql.*;
public class SomeStoredProcs {
  public static void myJavaStoredProcedure(
      int myInputInteger,
      String[] myOutputString,
      ResultSet[] myFirstResultSet,
      ResultSet[] mySecondResultSet) {
    // SP implementation
    ...
    myFirstResultSet[0] = stmt1.executeQuery(qry1);
    ...
    mySecondResultSet[0] = stmt2.executeQuery(qry2);
  }
}
```
Returning result sets in DB2GENERAL style stored procedures

To return a result set in procedures that use the DB2GENERAL parameter style, the result set, and responding statement, must be left open at the end of the procedure. The result set that is returned must be closed by the client application. If multiple results sets are returned, they are returned in the order in which they were opened. The RESULT SETS in CREATE PROCEDURE establish the maximum number of result sets. For example, the following stored procedure returns two results sets.

```
CREATE PROCEDURE RETURNTWORESULTSETS ()
EXTERNAL NAME Sample2!returnTwoResultSets
PARAMETER STYLE DB2GENERAL
RESULT SETS 2
LANGUAGE JAVA
```

Example 7-1 shows the Java method implementing this stored procedure.

```
 Example 7-1   Java method implementing the CREATE PROCEDURE

import java.sql.*;
import com.ibm.db2.app.*; // StoredProc and associated classes

public class sample2 extends StoredProc {

    /** 
     * Java Stored procedure with DB2GENERAL style Parameters 
     * that execute TWO predefined statements 
     */
    public void returnTwoResultSets () throws Exception {
        // get caller's connection to the database; inherited from StoredProc
        Connection con = getConnection(); // StoredProc and associated classes
        Statement stmt1 = con.createStatement();
        String sql1 = "SELECT EMPNO FROM SAMPLEDB02.EMPLOYEE WHERE HIREDATE < '1980-01-01';";
        stmt1.execute(sql1);
        Statement stmt2 = con.createStatement();
        String sql2 = "SELECT EMPNO FROM SAMPLEDB02.EMPLOYEE WHERE SALARY > 50000";
        stmt2.execute(sql2);
    }
}
```

Notes: The following notes refer to the previous example.

1. In the CREATE PROCEDURE, the result sets are not declared as parameters. The parameters are declared in the usual way.
2. Result sets in stored procedures with the Java parameter style differs significantly from result sets in stored procedures with DB2GENERAL parameter style.
3. Number of result set must be defined explicitly in Java parameter style stored procedures.
4. In the method, result sets are declared as output parameters.
5. In the implementation, the result sets are returned just like any other output parameter, assigning a result set object to the proper parameter vector.
7.3 Coding examples

The simple example described in this section refers to the CUSTOMER table, which is part of our test database.

Here we present several versions of two Java stored procedures, depending on how the stored procedure interacts with the database, either using JDBC or SQLJ:

- Java stored procedure with the Java parameter style: It inserts a row into the CUSTOMER table. The input parameters are the columns of the table, and the output parameter is the number of rows inserted. We provide an explanation of two versions of this example:
  - JavaInsertCus using JDBC to connect and work with the database
  - JavaSQLJInsertCus using SQLJ to interact with the database

- Java stored procedure with DB2GENERAL parameter style: It counts the number of customers in a given city. The input parameter is the name of the city, and the output parameter is the number of customers within the city. Three versions of this example are described:
  - DB2CusInCity using JDBC
  - DB2SQLJCusInCity using SQLJ and an SQLJ iterator for internal processing
  - DB2SQLJCusInCity2 using SQLJ and where the SQLJ iterator is replaced by the SELECT INTO statement

Since the stored procedures used in these examples have output parameters, we cannot call them directly from an interactive interface like the 5250 Interactive SQL or the Run SQL Scripts utility available in iSeries Navigator. We created a Java application named Client that calls the stored procedures, receives the results, and displays them.
JavaInsertCus and JavaSQLJInsertCus examples

The JDBC implementation of the JavaInsertCus stored procedure is discussed in this section. Example 7-2 shows the JavaInsertCus stored procedure.

Example 7-2  JavaInsertCus stored procedure

```java
import java.sql.*;
public class JavaInsertCus {
    public static void JavaInsertCus (  
        String s1, String s2, String s3, String s4, String s5,  
        String s6, String s7, java.math.BigDecimal bd1,  
        java.math.BigDecimal bd2, int[] insertCount )  
    throws SQLException, Exception {
        Connection con = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement stmt = null;
        String sql;
        sql = "INSERT INTO CUSTOMER VALUES(?,?,?,?,?,?,?,?,?)";
        stmt = con.prepareStatement( sql );
        stmt.setString( 1, s1 );
        stmt.setString( 2, s2 );
        stmt.setString( 3, s3 );
        stmt.setString( 4, s4 );
        stmt.setString( 5, s5 );
        stmt.setString( 6, s6 );
        stmt.setString( 7, s7 );
        stmt.setBigDecimal( 8, bd1 );
        stmt.setBigDecimal( 9, bd2 );
        insertCount[0] = stmt.executeUpdate();
        try {
            if (stmt != null) stmt.close();
            if (con != null) con.close();
        } catch (SQLException e) { /* ignore */ };
    }
}
```

Notes: The following notes refer to Example 7-2:

1. Import of the standard JDBC API.
2. The class must be defined as public.
3. The method to be defined as the stored procedure must be declared as `public static void`. There can be more than one method in the class. The method can have any name. It does not have to match the name of the class.
4. Input parameters are defined like any parameter in a Java method. The output parameter, `insertCount`, is defined as an array of integers.
5. We obtain the default connection from the Driver Manager. The URL "jdbc:default:connection" must always be used in a Java stored procedure with the Java parameter style.
6. After receiving all parameters, we use them to prepare an Insert statement, and execute it. Then we place the return code from the `executeUpdate` method into the first element of the array declared in the parameters. It is returned as an output parameter. This array cannot be used to return multiple values. Only the first element is to be assigned.
7. Error handling is discussed in Chapter 7, “Java stored procedures” on page 173.
We can now compare the JDBC version with its much more compact SQLJ counterpart, JavaSQLJInsertCus.

**Example 7-3  JavaSQLJInsertCu**

```java
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

public class JavaSQLJInsertCus
{
    public static void JavaSQLJInsertCus (String s1, String s2, String s3,
    String s4, String s5, String s6, String s7, java.math.BigDecimal bd1,
    java.math.BigDecimal bd2, int[] insertCount )
    throws SQLException, Exception
    {
        ExecutionContext ec =
            DefaultContext.getDefaultContext().getExecutionContext();
        #sql {
            INSERT INTO CUSTOMER VALUES(:s1,:s2,:s3,:s4,:s5,:s6,:s7,:bd1,:bd2);
        }
        insertCount[0] = ec.getUpdateCount();
    }
}
```

**Notes:** In the following notes, we only discuss the SQLJ specific features. However, comments 1 through 4 made for the JDBC version apply to this example as well.

1. These classes are always needed when you want to use SQLJ.
2. The execution context from the default context gives us access to special variables, like environment variables set after the execution of the SQLJ statement. Note also that, with SQLJ and the Java parameter style, the connection is there implicitly while the JDBC version requires the use of the getConnection method.
3. The SQLJ statement is simply an SQL statement embedded in Java. It will be translated in JDBC API calls by the SQLJ translator.
4. We use the getUpdateCount method on the ExecutionContext object to set the count variable. It contains the number of rows that have been inserted by the SQLJ statement. It corresponds to the return value passed by the JDBC executeUpdate method seen in the previous version of this example.

**DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples**

Example 7-4 illustrates the DB2GENERAL parameter style. The first version of this stored procedure uses a JDBC connection to the database.

**Example 7-4  DB2GENERAL parameter style**

```java
import java.sql.*;
import com.ibm.db2.app.*;

class DB2CusInCity extends StoredProc
{
    public void DB2CusInCity (String s, int i) throws Exception
    {
        Connection con = getConnection();
        PreparedStatement ps = null;
        ResultSet rs = null;
        String sql;
        sql = "SELECT Count(*) FROM CUSTOMER WHERE (CUSTOMER_CITY = ?)";
```
ps = con.prepareStatement( sql );
ps.setString( 1, s );
rs = ps.executeQuery();
rs.next();
set(2, rs.getInt(1));
if (rs != null) rs.close();
if (ps != null) ps.close();
if (con != null) con.close();
}
}

**Notes:** The following notes refer to Example 7-4:

1. We import the StoredProc class. This statement also imports the CLOB, BLOB, and other associated classes.
2. In the DB2GENERAL parameter style, we must subclass the StoredProc class when defining a Java stored procedure.
3. In the definition of the method, no distinction is made between the input and output parameters. They are all potentially inout parameters.
4. We instantiate the connection to the database with the getConnection method provided by the StoredProc class.
5. Any output or inout parameter must be set by the set method of StoredProc to return its value to the caller. The first argument of this method refers to the position of the considered parameter in the method signature, and the second argument indicates its value.

The following versions of this stored procedure use SQLJ to access the database. The comments made for the previous version also apply to these versions. We only emphasize the differences.

**DB2SQLJCusInCity** is an SQLJ version of DB2CusInCity. It uses a result set to obtain the number of customers in a given city.

**Example 7-5  DB2SQLJCusInCity**

```java
import java.sql.*;
import com.ibm.db2.app.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator DB2SQLJCusInCity_C (int);

class DB2SQLJCusInCity extends StoredProc
{
    public void DB2SQLJCusInCity ( String s, int i )
        throws SQLException, Exception
    {
        DB2SQLJCusInCity_C c;
        DefaultContext ctx = DefaultContext.getDefaultContext();
        if (ctx == null)
        {
            Connection con = getConnection ();
            ctx = new DefaultContext(con);
            DefaultContext.setDefaultContext(ctx);
        }
        #sql c = { SELECT Count(*) FROM CUSTOMER WHERE (CUSTOMER_CITY = :s) };
        #sql { Fetch :c into :i};
```
Notes: The following notes refer to Example 7-5:

1. SQLJ defines a cursor as an iterator. It must be declared before the class and is transformed at compile time in a Java class. We declare that the cursor will contain one column of type int.

2. We declare “c” as an instance of type iterator. We use it like a cursor in the embedded SQL.

3. With the DB2GENERAL parameter style, we must explicitly specify that we are running in the default context.

4. The cursor is processed as an embedded SQL program.

The last SQLJ version uses a simpler way of obtaining the number of customers. The result set is replaced by a SELECT INTO statement. The corresponding code of DB2SQLJCusInCity2 is shown in Example 7-6.

Example 7-6   Corresponding code of DB2SQLJCusInCity2

```java
import java.sql.*;
import com.ibm.db2.app.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class DB2SQLJCusInCity2 extends StoredProc {
    public void DB2SQLJCusInCity2 ( String s, int i ) throws SQLException, Exception {
        DefaultContext ctx = DefaultContext.getDefaultContext();
        if (ctx == null) {
            Connection con = getConnection ();
            ctx = new DefaultContext(con);
            DefaultContext.setDefaultContext(ctx);
        }
        #sql { SELECT Count(*) INTO :i FROM CUSTOMER WHERE (CUSTOMER_CITY = :s) };
        set(2, i);
    }
}
```

Note: With SQLJ, we can use the SELECT INTO statement. This is not possible with JDBC. Note that SQLJ uses JDBC underneath, so the SELECT INTO statement will be transformed in a result set and fetched with the next method as in our JDBC version.
7.3.1 Compilation of Java code

CLASSPATH is the key environment variable to compile a Java program. Depending on the imports made in the Java code, you need to modify CLASSPATH so that it points at the .jar or .zip files required by these imports. Table 7-2 summarizes the files that are needed for different programming styles.

Table 7-2 Files required in the classpath

<table>
<thead>
<tr>
<th>Parameter style</th>
<th>Database connection</th>
<th>File required in the classpath</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA</td>
<td>JDBC</td>
<td>db2_classes.jar</td>
</tr>
<tr>
<td>JAVA</td>
<td>SQLJ</td>
<td>db2_classes.jar, translator.zip, runtime.zip</td>
</tr>
<tr>
<td>DB2GENERAL</td>
<td>JDBC</td>
<td>db2_classes.jar, db2routines_classes.jar</td>
</tr>
<tr>
<td>DB2GENERAL</td>
<td>SQLJ</td>
<td>db2_classes.jar, db2routines_classes.jar, translator.zip, runtime.zip</td>
</tr>
</tbody>
</table>

All these files are located in the IFS in the /QIBM/ProdData/OS400/Java400/ext directory. If you decide to compile on the iSeries server, the db2_classes.jar file containing the DB2 native driver and the JDBC API are automatically added to the classpath. If you prefer to compile on your PC, you must specify all these files in your classpath.

The compilation is made with the `javac` command when the stored procedure only uses JDBC to connect to the database. It results in one .class file. When SQLJ is used, the compilation command is `sqlj`. It creates one .ser file, two .class files, and one additional .class file for each iterator.

JavaInsertCus and JavaSQLJInsertCus examples

Before compiling your Java code for these examples, you need to set up your classpath so that it points to the JDBC and SQLJ classes. For the ease of our tests, we decided to compile on the PC and copied the iSeries server db2_classes.jar, db2routines_classes.jar, translator.zip, and runtime.zip files on the local disk of our PC in the D:\as400_classes directory. Our classpath is set with the command shown here:

```
set CLASSPATH= %CLASSPATH%;D:\as400_classes\db2_classes.jar; D:\as400_classes\translator.zip;D:\as400_classes\runtime.zip
```

Notes: The following notes refer to the previous example.

1. The db2_classes.jar file contains the JDBC driver and the JDBC classes. Your original classpath (%CLASSPATH%) should already point to the JDK™ classes.
2. The translator.zip file provides the SQLJ translator (or precompiler), while the runtime.zip file contains the Java runtime support for SQLJ.

You can now compile your .java and .sqlj files containing the stored procedures with the commands listed in Table 7-3.

Table 7-3 Compilation commands and their output files

<table>
<thead>
<tr>
<th>Compilation commands</th>
<th>Result files</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac JavaInsertCus.java</td>
<td>JavaInsertCus.class</td>
</tr>
<tr>
<td>sqlj JavaSQLJInsertCus.sqlj</td>
<td>JavaSQLJInsertCus.java, JavaSQLJInsertCus.class, JavaSQLJInsertCus_SJProfile0.ser, JavaSQLJInsertCus_SJProfileKeys.class</td>
</tr>
</tbody>
</table>
The classpath and compilation considerations are similar to those made for the Java parameter style example. The difference is the additional .jar file containing the StoredProc class db2routines_classes.jar in the classpath. Our classpath is set with the following command:

```
set CLASSPATH= %CLASSPATH%;D:\as400_classes\db2_classes.jar;
D:\as400_classes\db2routines_classes.jar;
D:\as400_classes\translator.zip;
D:\as400_classes\runtime.zip
```

The compilation of the .java and .sqlj files containing the stored procedures produces the files shown in Table 7-4.

<table>
<thead>
<tr>
<th>Compilation commands</th>
<th>Result files</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac DB2CusInCity.java</td>
<td>DB2CusInCity.class</td>
</tr>
<tr>
<td>sqlj DB2SQLJCusInCity.sqlj</td>
<td>DB2SQLJCusInCity.java</td>
</tr>
<tr>
<td></td>
<td>DB2SQLJCusInCity.class</td>
</tr>
<tr>
<td></td>
<td>DB2SQLJCusInCity_C.class</td>
</tr>
<tr>
<td></td>
<td>DB2SQLJCusInCity_SJProfile0.ser</td>
</tr>
<tr>
<td></td>
<td>DB2SQLJCusInCity_SJProfileKeys.class</td>
</tr>
<tr>
<td>sqlj DB2SQLJCusInCity2.sqlj</td>
<td>DB2SQLJCusInCity2.java*</td>
</tr>
<tr>
<td></td>
<td>DB2SQLJCusInCity2.class</td>
</tr>
<tr>
<td></td>
<td>DB2SQLJCusInCity2_SJProfile0.ser</td>
</tr>
<tr>
<td></td>
<td>DB2SQLJCusInCity2_SJProfileKeys.class</td>
</tr>
</tbody>
</table>

* This .java file produced by the sqlj precompilation confirms that a result set and the next() method are used for the SELECT INTO implementation, as discussed in “DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples” on page 182.

### 7.3.2 Where to place Java classes

The compiled code must be loaded into the function directory, `/QIBM/UserData/OS400/SQLLib/Function`, of the iSeries server.

The process of deploying the compiled code into the function directory depends on where the code was compiled. If it was compiled on a Windows workstation, you can take advantage of iSeries server NetServer™ and map the function directory as a network drive on your machine.

### JavaInsertCus and JavaSQLJInsertCus examples

Copy the JavaInsertCus.class file for the JDBC type stored procedure. Copy the JavaSQLJInsertCus.class, JavaSQLJInsertCus_SJProfile0.ser, and JavaSQLJInsertCus_SJProfileKeys.class files for the SQLJ type stored procedure into the IFS function directory of the iSeries server (`/QIBM/UserData/OS400/SQLLib/Function`), as shown in Figure 7-1.
In this example, apart from the .java file produced by the SQLJ precompilation, we copy all other files resulting from the compilation into the function directory of the iSeries server.

### 7.3.3 Creating Java programs

After you compile the Java stored procedure and deploy it on the iSeries server, you may improve the Java code performance by using the Create Java Program (CRTJVAPGM) command. The CRTJVAPGM command uses bytecodes to create a Java program that contains optimized native instructions for the iSeries server and associates the Java program object with the class file. To create the optimized Java program for the DB2CusInCity class, use the following CL command:

```
CRTJVAPGM CLSF(0b2CusInCity.class) OPTIMIZE(40)
```
7.4 Registering Java stored procedures

The last step before completing the Java stored procedure is to identify the code you loaded in the function directory as a Java stored procedure and to register it into the system catalog (SYSROUTINES and SYSPARMS system tables). This is done by using the CREATE PROCEDURE SQL statement. The syntax of this statement was extended to allow the new type of stored procedures. Figure 7-2 details the new syntax of the CREATE PROCEDURE statement.

---

```
CREATE PROCEDURE procedure-name
(
  parameter-name [AS LOCATOR]
) [OUT | IN | INOUT]

LANGUAGE JAVA

SPECIFIC specific-name

NOT DETERMINISTIC

MODIFIES SQL DATA

DETERMINISTIC

NO SQL

CONTAINS SQL

READS SQL DATA

FENCED

CALLED ON NULL INPUT

EXTERNAL NAME external-program-name

PARAMETER STYLE SQL

OBINFO

EXTERNAL
```

---

Figure 7-2  CREATE PROCEDURE syntax
The most important options are:

- **RESULT SET**: This clause is supported for Java stored procedures starting with the V5R1 implementation of Java stored procedures.
- **LANGUAGE**: JAVA; the external program is written in Java.
- **PARAMETER STYLE**: DB2GENERAL or JAVA; these values can only be used when the language Java is specified.
- **EXTERNAL NAME**: A Java class and a method can now be specified, separated by an exclamation mark (!) or by a period (.). The use of "!" was introduced by the DB2GENERAL parameter style, and the use of "." is part of the SQLJ standard. For example, you may specify TheClass!theMethod or TheClass.theMethod. Only the name of the method can be specified, not the method and its parameters.

**Note**: You do not specify the class path name, since all Java stored procedure classes must reside in the /QIBM/UserData/OS400/SQLLib/Function directory.

### 7.4.1 Registering Java stored procedures with iSeries Navigator

For the Java parameter style, the Java stored procedure can be defined with the iSeries Navigator New External Procedure window. The required steps are listed here:

1. In iSeries Navigator, expand **Database -> Libraries**. Right-click the library in which you want to create the Java stored procedure and select **New -> Procedure -> External**. See Figure 7-3.

![Figure 7-3 Creating a Java stored procedure](image)
2. The New External Procedure window (Figure 7-4) opens.
   a. On the General tab, enter the name for the new stored procedure. You can also enter a description and a specific name.

   ![New External Procedure window](image)

   Figure 7-4 Creating a Java stored procedure: The General tab

   b. Click the Parameters tab. Choose the Java parameter style, and click the Insert button to add the parameters, as shown in Figure 7-5.

   **Note:** The DB2GENERAL parameter style Java stored procedures cannot be registered using iSeries Navigator.
3. Click the **External Program** tab of the window. Choose a Java method, and enter the name of the class and the name of the method separated by an exclamation mark (!) or a period (.) as indicated in Figure 7-6.

Figure 7-6  Creating a Java stored procedure: The External Program tab

   c. Click the **OK** button. The Java procedure registration is now complete.
7.4.2 Using the Run SQL Scripts utility

As mentioned earlier, loading the classes into the function directory does not mean that DB2 Universal Database for iSeries is ready to use the new stored procedures. First we have to register our methods in the Java classes as stored procedures with the CREATE PROCEDURE SQL statement.

In the example in Figure 7-7, we execute these statements in the Run SQL Scripts utility. To access the utility from the main iSeries Navigator window, right-click the Database object, and select Run SQL Scripts.

![Figure 7-7 Creating the stored procedures in Run SQL Scripts](image)

The execution of these statements updates the system catalog adding the two new stored procedures and their parameters in the SYSROUTINES and SYSPARMS tables.

**Note:** We do not qualify the stored procedure names with a library (schema) name. If the current naming convention for the Run SQL Scripts session is *SQL, the stored procedures are registered in the library with the same name as the current user profile for this session. If the naming convention is *SYS, the stored procedures are registered in the current library.

7.4.3 Using the native interface

This time, we use the 5250 session to create (register) our stored procedures. Rather than using an Interactive SQL session, you can write the statements in a source physical file. The content of the example source file is shown here:

```sql
CREATE PROCEDURE DB2CUSINCITY(IN S CHAR(20), OUT I INTEGER)
LANGUAGE JAVA PARAMETER STYLE DB2GENERAL NOT FENCED
EXTERNAL NAME 'DB2CUSINCITY!DB2CUSINCITY';

CREATE PROCEDURE DB2SQLJCUSINCITY(IN S CHAR(20), OUT I INTEGER)
LANGUAGE JAVA PARAMETER STYLE DB2GENERAL NOT FENCED
EXTERNAL NAME 'DB2SQLJCUSINCITY!DB2SQLJCUSINCITY';

CREATE PROCEDURE DB2SQLJCUSINCITY2(IN S CHAR(20), OUT I INTEGER)
LANGUAGE JAVA PARAMETER STYLE DB2GENERAL NOT FENCED
EXTERNAL NAME 'DB2SQLJCUSINCITY2!DB2SQLJCUSINCITY2';
```
7.5 Calling Java stored procedures

A Java stored procedure is like any other stored procedure and can be called from any programming interface that supports the SQL CALL function. The convention that the output parameters in the Java parameter style must be defined as arrays impacts only the Java method code. From the calling process point of view, there is no difference between an output parameter returned by a Java stored procedure and an output parameter returned by a stored procedure written in other programming languages.

JavaInsertCus and JavaSQLJInsertCus examples

To test our Java stored procedures, we use the client application in Example 7-7 that calls our JDBC and SQLJ Java stored procedures.

**Example 7-7  Client application to call JDBC and SQLJ Java stored procedures**

```java
import java.math.*;
import java.sql.*;
import com.ibm.as400.access.*;

class Client
{static
    {try
        System.out.println();
        System.out.println("  Java Stored Procedure Sample");
        Class.forName("com.ibm.as400.access.AS400JDBCDriver").newInstance();
    }
    catch (Exception e)
    {System.out.println("  Error loading AS400JDBCDriver...
    e.printStackTrace();
    }
}

public static void main (String argv[])
{ Connection con = null;
    CallableStatement ps;
    String sql;
    try
    { String url = "jdbc:as400://AS400WS"
        if (argv.length == 0) {
            con = DriverManager.getConnection(url);
        }
        else if (argv.length == 2) {
            String userid = argv[0];
            String passwd = argv[1];
            con = DriverManager.getConnection(url, userid, passwd);
        }
        else {System.out.println("Usage: java spbjavaclient [username password]");
            System.exit(0);
        }
        sql = "Call JAVAINSERTCUS (?,?,?,?,?,?,?,?,?,?)";
        ps = con.prepareCall(sql);
        ps.setString(1, "00002");
        ps.setString(2, "Josephina Kissinger");
        ps.setString(3, "xx-00-1-7777788");
        ps.setString(4, "xx-00-1-7777778");
        ps.setString(5, "2nd Avenue NW");
```
The output of the client is shown here:

Java Stored Procedure Sample
There was 1 record inserted.
There was 1 record inserted.

It indicates that the stored procedures were successfully executed.
### DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples

Our same client application calls the new stored procedures illustrating the DB2GENERAL parameter style as shown in Example 7-8.

**Example 7-8  New stored procedures illustrating the DB2GENERAL parameter style**

```java
import java.math.*;
import java.sql.*;
import com.ibm.as400.access.*;

class Client
{
    static
    {
        try
        {
            System.out.println();
            System.out.println("  Java Stored Procedure Sample");
            Class.forName("com.ibm.as400.access.AS400JDBCDriver").newInstance();
        }
        catch (Exception e)
        {
            System.out.println("\n  Error loading AS400JDBCDriver...
");
            e.printStackTrace();
        }
    }

    public static void main (String argv[])
    {
        Connection con = null;
        CallableStatement ps;
        String sql;
        try
        {
            String url = "jdbc:as400://AS400WS";
            if (argv.length == 0) {
                con = DriverManager.getConnection(url);
            }
            else if (argv.length == 2) {
                String userid = argv[0];
                String passwd = argv[1];
                con = DriverManager.getConnection(url, userid, passwd);
            }
            else {
                System.out.println("\nUsage: java spbjavaclient [username password]\n");
                System.exit(0);
            }

            sql = "Call DB2CUSINCITY (?,?)"
            ps = con.prepareCall(sql);
            String city = "Calentina";
            int nbrCusInCity = 0;
            ps.setString(1, city);
            ps.registerOutParameter(2, Types.INTEGER);
            ps.execute();
            System.out.println(ps.getInt(2) + " person(s) found in "+city+".");

            sql = "Call DB2SQLJCUSINCITY (?,?)"
            ps = con.prepareCall(sql);
            city = "Calentina";
            nbrCusInCity = 0;
            ps.setString(1, city);
            ps.registerOutParameter(2, Types.INTEGER);
            ps.execute();
            System.out.println(ps.getInt(2) + " person(s) found in "+city+".");

            sql = "Call DB2SQLJCUSINCITY2 (?,?)"
            ps = con.prepareCall(sql);
            city = "Calentina";
            nbrCusInCity = 0;
            ps.setString(1, city);
```
ps.registerOutParameter(2, Types.INTEGER);
ps.execute();
System.out.println(ps.getInt(2) + " person(s) found in " + city + ".");

if (ps != null) ps.close();
if (con != null) con.close();
}
catch (Exception e)
{
   e.printStackTrace();
}
}

The output of this client is as expected:

Java Stored Procedure Sample
2 person(s) found in Calentina.
2 person(s) found in Calentina.
2 person(s) found in Calentina.

7.6 Using SQL NULL

The theory concerning SQL NULL values was presented in 7.2, “Coding DB2 Universal Database for iSeries Java stored procedures” on page 174. This section compares two stored procedures using the JAVA and DB2GENERAL parameter styles: JAVASPNULL and DB2SPNULL. These two procedures receive two parameters, a character string, and an integer. They insert them in the table, called T, created with the following SQL statement:

CREATE TABLE T(S CHAR (5), I INTEGER)

Both columns of the table are NULL capable.

The code of JAVASPNULL is shown in Example 7-9.

Example 7-9   JAVASPNULL code

import java.sql.*;

public class javaspnull
{
   public static void javaspnull (String s, int i) throws SQLException, Exception
   {
      Connection con = DriverManager.getConnection("jdbc:default:connection");
      PreparedStatement ps = null;
      String sql;
      sql = "insert into t(s,i) values(?,?)";
      ps = con.prepareCall(sql);
      if (s == null)
      { ps.setNull(1, Types.CHAR); }
      else { ps.setString(1, s); }
      ps.setInt(2, i);
      ps.executeUpdate();
      try {if (ps != null) ps.close();
      if (con != null) con.close();
      } catch (SQLException e) { /* ignore */ };
   }
}
Example 7-10 shows the code of DB2SPNULL.

**Example 7-10   DB2SPNULL code**

```java
import java.sql.*;
import com.ibm.db2.app.);

public class db2spnull extends StoredProc
{
    public void db2spnull (String s, int i) throws SQLException, Exception
    {
        Connection con = getConnection();
        PreparedStatement ps = null;
        String sql;
        sql = "insert into t(s,i) values(?,?)";
        ps = con.prepareCall(sql);
        if (isNull(1))
        {ps.setNull(1, Types.CHAR); }  \[1\]
        else {ps.setString(1, s);}
        if (isNull(2))
        {ps.setNull(2, Types.INTEGER); }  \[2\]
        else {ps.setInt(2, i);}
        ps.executeUpdate();
        try {if (ps != null) ps.close();
        if (con != null) con.close();
        } catch (SQLException e) { /* ignore */ ;}
    }
}
```

**Notes:** The following notes refer to Example 7-10:

1. With the DB2GENERAL parameter style, we test if a parameter has an SQL NULL value with the isNull method implemented by the StoredProc class. Its argument refers to the position of the parameter in the method definition. It returns TRUE if the value is NULL.
2. We use the same setNull method to prepare our insert SQL statement as in the Java parameter style.
3. The isNull method allows us to check if the integer parameter has an SQL NULL value.
4. We can insert a NULL value for the integer parameter. Note that this is impossible with the Java parameter style.
After creating the two stored procedures corresponding to the two Java methods, we call them with the SQL statements in the Interactive SQL session, as shown in Figure 7-8. Passing NULL as an integer results in an error. The javaspnull user-defined function or procedure has an input argument with a null value.

![Enter SQL Statements](image)

We receive the output shown in Figure 7-9 that confirms the previous comments.

![Display Data](image)

**7.7 SQLJ procedures to manipulate JAR files**

Java stored procedures (as well as Java user-defined functions) can use Java classes that are stored in Java JAR files. To use a JAR file, a *jar-id* must be associated with the JAR file.

Starting on V5R1, the system provides stored procedures in the SQL schema that allow *jar-ids* and JAR files to be manipulated. These procedures allow JAR files to be associated with stored procedures.

Here we explain briefly the following stored procedures:

- SQLJ.INSTALL_JAR
- SQLJ.REPLACE_JAR
- SQLJ.REMOVE_JAR
- SQLJ.UPDATE_JARINFO
- SQLJ.RECOVER_JAR

To use these stored procedures, we need INSERT, UPDATE, DELETE, and SELECT privileges for SYSJAROBJECTS and SYSJARCONTENTS catalog tables and *EXECUTE authority on library QSYS2. We also need proper authorities to the affected JAR file and /QIBM/UserData/OS400/SQLLib/Function/jar/schema directory, where schema is the schema of the *jar-id*. More details can be found in SQL Programming Guide, SC41-5611.

Adopted authority cannot be used for these authorities.
7.7.1 SQLJ.INSTALL_JAR

The SQLJ.INSTALL_JAR stored procedure installs a JAR file into DB2 Universal Database for iSeries. This JAR file can be used in subsequent CREATE PROCEDURE statements.

The SQLJ.INSTALL_JAR stored procedure requires three parameters:

- **jar-url**: The URL containing the JAR file to be installed or replaced. The only URL schema supported is "file:"
- **jar-id**: The JAR identifier in the database to be associated with the file specified by the jar-url. The jar-id will use SQL naming, and the JAR file will be installed in the schema or library specified by the implicit or explicit qualifier.
- **deploy**: Value used to describe the install_action of the deployment descriptor file. If this integer is a non-zero value, then the install_action of a deployment descriptor file should be executed at the end of the install_jar procedure. The current version of DB2 Universal Database for iSeries only supports a value of zero.

When a JAR file is installed, DB2 Universal Database for iSeries registers the JAR file in the SYSJAROBJECTS system catalog. It also extracts the names of the Java class files from the JAR file and registers each class in the SYSJARCONTENTS system catalog. DB2 Universal Database for iSeries copies the JAR file to a jar/schema subdirectory of the /QIBM/UserData/OS400/SQLLib/Function directory. DB2 Universal Database for iSeries gives the new copy of the JAR file the name given in the jar-id clause.

When a JAR file is installed, DB2 Universal Database for iSeries registers the JAR file in the SYSJAROBJECTS system catalog. It also extracts the names of the Java class files from the JAR file and registers each class in the SYSJARCONTENTS system catalog. DB2 Universal Database for iSeries copies the JAR file to a jar/schema subdirectory of the /QIBM/UserData/OS400/SQLLib/Function directory. DB2 Universal Database for iSeries gives the new copy of the JAR file the name given in the jar-id clause.

**Note**: A JAR file that has been installed by DB2 Universal Database for iSeries into a subdirectory of /QIBM/UserData/OS400/SQLLib/Function should not be modified. Instead, remove or replace an installed JAR file.

**Example**

We create a sample JAR file using the following command in the Qshell interface provided by OS/400:

```
jar -cvf sample.jar sample*.class
```

The following command is issued from an SQL interactive session:

```
CALL SQLJ.INSTALL_JAR('file:/home/dlema/JavaTest/sample.jar', 'mysample_jar', 0)
```

The sample.jar file located in the /home/dlema/JavaTest directory is installed in DB2 Universal Database for iSeries with the name of MYSAMPLE_JAR. Subsequent SQL commands that use the samples.jar file refer to it with the name of MYSAMPLE_JAR.

If we review the contents of the /QIBM/UserProd/OS400/SQLLib/Function/jar/DLEMA, we will see MYSAMPLE_JAR.file. Notice that the .../jar/DLEMA directory was automatically created by the SQLJ.INSTALL_JAR stored procedure.
Now, if we examine the contents of QSYS2.SYSJAROBJECTS, we can see our recently installed JAR file, as shown in Figure 7-10.

![Figure 7-10](image)

**Figure 7-10**  QSYS2.SYSJAROBJECTS contents after installing the MYSAMPLE JAR file
The QSYS2.SYSJARCONTENTS table will have a row for each class inside the installed JAR files, as shown in Figure 7-11.

```
SELECT * FROM QSYS2.SYSJAROBJECTS;
SELECT * FROM QSYS2.SYSJARCONTENTS;
```

**Figure 7-11** QSYS2.SYSJARCONTENTS after installing the MYSAMPLE JAR file

### 7.7.2 SQLJ.REMOVE_JAR

The SQLJ.REMOVE_JAR stored procedure removes a JAR file into DB2 Universal Database for iSeries. The SQLJ.REMOVE_JAR stored procedure requires two parameters:

- **jar-id**: The JAR identifier associated with the JAR file to be removed from the database.
- **undeploy**: Value used to describe the install_action of the deployment descriptor file. If this integer is a non-zero value, then the remove_actions of a deployment descriptor file should be executed at the end of the install_jar procedure. The current version of DB2 Universal Database for iSeries only supports a value of zero.

**Example**

The following command is issued from an SQL interactive session:

```
CALL SQLJ.REMOVE_JAR('mysample_jar', 0)
```

The entries in QSYS2.SYSJAROBJECTS and QSYS2.SYSJARCONTENTS will be deleted and the /QIBM/UserData/OS400/SQLLib/Function/jar/schema/MYSAMPLE.jar will be deleted, but the ... jar/schema directory will remain.
7.7.3 SQLJ.REPLACE_JAR

The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into DB2 Universal Database for iSeries. The SQLJ.REPLACE_JAR stored procedure requires two parameters:

- **jar-url**: The URL containing the JAR file to be replaced. The only URL schema supported in "file:"
- **jar-id**: The JAR identifier associated with the JAR file to be removed from the database.

**Example**

The following command is issued from an SQL interactive session:

```
CALL SQLJ.REPLACE_JAR(‘file:/home/dlema/JavaTest/mysample.jar‘, ‘mysample_jar’)
```

The current JAR file referred to by the jar-id mysample_jar is replaced with the new mysample.jar file located in the specified directory.

7.7.4 SQLJ.UPDATEJARINFO

The SQLJ.UPDATEJARINFO stored procedure updates the CLASS_SOURCE column of the SYSJARCONTENTS catalog table. This procedure is not part of the SQLJ standard, but it is used by the DB2 Universal Database for iSeries stored procedure builder.

The SQLJ.UPDATEJARINFO stored procedure requires three parameters:

- **jar-id**: The JAR identifier that is to be updated.
- **class-id**: The package qualified class name of the class to be updated.
- **jar-url**: The URL containing the JAR file with which to update the JAR file. The only URL schema supported in "file:"

**Example**

The following command is issued from an SQL interactive session:

```
CALL SQLJ.UPDATEJARINFO('mysample_jar', 'mypackage.myclass', 'file:/home/dlema/JavaTest/mypackage/myclass.class')
```

The JAR file associated with the jar-id, mysample_jar is updated with a new version of the mypackage.myclass class. The new version of the class is obtained from the file /home/dlema/JavaTest/mypackage/myclass.class.

7.7.5 SQLJ.RECOVERJAR

The SQLJ.RECOVERJAR stored procedure takes the JAR file that is stored in the SYSJAROBJECTS catalog and restores it to the /QIBM/UserData/OS400/SQLLib/Function/jar/jar_schema/jar_id.jar file.

The SQLJ.RECOVERJAR stored procedure requires the jar-id parameter. This is the JAR identifier that is to be recovered.

**Example**

The following command is issued from an SQL interactive session:

```
CALL SQLJ.RECOVERJAR('mysample_jar')
```

The JAR file associated with the jar-id, mysample_jar, is updated with the contents from the SYSJARCONTENT table. The file is copied to the /QIBM/UserData/OS400/SQLLib/Function/jar/jar_schema/mysample_jar.jar file.
7.8 Additional considerations

This section focuses on important topics that are specific to the current implementation of Java stored procedures on the iSeries server.

General considerations are listed here:

- **Built-in functions**: The built-in functions described in the SQLJ Routines standard were introduced in V5R1. SQLJ.ALTER_JAR_PATH is not supported yet.

- **Result set**: Result sets are supported starting on V5R1.

- **Adopted authority**: The authority of the current user is used. There is no support for the concept of adopted authority.

- **Threads**: Java stored procedures should not create additional threads.

- **Java Development Kit (JDK)**: Java stored procedures automatically use the latest version of the Java Development Kit installed on the iSeries server.

- **Connection**: The only possible connection is to the current database so that no connect, disconnect, or set connect SQL statements can be used by the stored procedures.

- **Commitment control**: All actions take place in the current transaction, in the same commitment definition as the caller. No COMMIT or ROLLBACK can be executed.

- **Privileges**: The REVOKE and GRANT SQL statements cannot be used to allow or deny a user the privilege of executing a Java stored procedure. The user must be granted or denied authority to the underlying Java class file to impose execution privileges for the Java stored procedure.

**Unicode and character conversion**

Java always processes character data in Unicode. This means that when the character is not stored in Unicode, a character conversion must be done on any input and output of character data between Java and the database. This has a performance impact that can be avoided by storing the character data directly in Unicode CCSID 13488.

Two important consequences must be taken into account when considering this approach. First, if the character data is stored in Unicode, a character conversion is required for the applications that do not manipulate the data in Unicode. Secondly, the Unicode has double-byte encoding so that two bytes are required per character in place of one for a single-byte character set. For large tables, this can mean a significant increase in DASD requirements. Consequently, the performance of a Java application may improve when the character data is stored in Unicode. In this case, you should only consider changing the character data to Unicode when the data is exclusively or almost exclusively used by JDBC applications and when the additional DASD requirements are not a concern.

7.8.1 Moving into production (save and restore)

While deploying a database application to a production system, you need to save and restore objects such as external programs, that were registered as a stored procedures. Depending on the type of stored procedure and external program that implements that stored procedure, there may be some additional actions required to make it available on the target system.

For Java stored procedures, you can save the .class or .jar files implementing Java stored procedures to a save file or any other media and then restore it to a the production system in the /QIBM/UserData/OS400/SQLLib/Function library. The Java code loaded in a function directory in the IFS must be saved by the SAV command. When you restore the classes of
Java stored procedures into the function directory, you must manually recreate the procedure with the CREATE PROCEDURE statement because the system catalog is not updated automatically. This means that a CREATE PROCEDURE must be performed for each restored Java stored procedure.

For performance reasons, it is desirable to also perform a CRTJVAPGM with OPTIMIZE(40).

7.9 GetSuppliers example: Implementation with no result sets

As mentioned earlier, V4R5 implementation of Java stored procedures does not support result sets. In the Java version of this stored procedure, we circumvent this limitation by storing the content of the two result sets in a String object and returning it as an output parameter.

Since the result sets in Java stored procedures are supported in V5R1, in 7.10, “GetSuppliers example: Implementation with result sets” on page 211, we illustrate how the Java stored procedure can return the result sets to the calling process.

7.9.1 Stored procedure: GetSupplier

The business logic of this example is described in 5.7, “GetSuppliers example” on page 111. The stored procedure returns two result sets containing the list of n best and n worst suppliers and their total sales amount for a given month in a year or for the whole year.

Code overview
First we introduce the code of the GetSupplier Java class as shown in Example 7-11.

Example 7-11 GetSupplier Java class

```java
import java.sql.*;

public class GetSupplier
{
    public static void GetSupplier (int year, int month, int[] rank, String[] suppliers) throws SQLException, Exception
    {
        Connection con = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement ps = null;
        ResultSet rs = null;
        String sql;
        int rowCount;
        suppliers[0] = "";

        // best suppliers
        if (month < 1)
        {
            sql = "SELECT supplier_name, totalsales FROM yearsale WHERE (year = ?) " + "ORDER BY totalsales DESC";
        }
        else
        {
            sql = "SELECT supplier_name, totalsales FROM totalsale " + "WHERE ((year = ?) "AND (month = ?)) ORDER BY totalsales DESC";
        }
        ps = con.prepareStatement( sql );
        ps.setInt( 1, year );
        if (month > 0) {ps.setInt( 2, month );}
        rs = ps.executeQuery();
        rowCount = 0;
```

204 Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries
while ((rs.next()) && (rowCount<rank[0]))
{
    suppliers[0] = suppliers[0] + rs.getString(1).trim() + "/" + 
    rs.getBigDecimal(2, 0) + "/";
    rowCount++;
}
if (rowCount != rank[0]) {rank[0] = rowCount;}
try {if (rs != null) {rs.close();}}
catch (SQLException e) { /* ignore */ };

// worse suppliers
if (month < 1)
{
    sql = "SELECT supplier_name, totalsales FROM yearsale " + 
    "WHERE (year = ?) ORDER BY totalsales ASC";
}
else
{
    sql = "SELECT supplier_name, totalsales FROM totalsale " + 
    "WHERE ((year = ?) AND (month = ?)) ORDER BY totalsales ASC";
}
ps = con.prepareStatement( sql );
ps.setInt( 1, year );
if (month > 0) ps.setInt( 2, month );
rs = ps.executeQuery();
rowCount = 0;
while ((rs.next()) && (rowCount<rank[0]))
{
    suppliers[0] = suppliers[0] + rs.getString(1).trim() + "/" + 
    rs.getBigDecimal(2, 0) + "/";
    rowCount++;
}
if (rowCount > rank[0]) {rank[0] = rowCount;}
try {if (rs != null) {rs.close();}}
    if (ps != null) ps.close();
    if (con != null) con.close();
catch (SQLException e) { /* ignore */ };
}

Notes: The following notes refer to Example 7-11:

1. suppliers is the new output parameter used for returning the concatenation of all suppliers and their corresponding total sales amount. rank is an inout parameter that passes the requested number of the suppliers. It returns the actual number of suppliers returned by the stored procedure.

2. The suppliers variable contains the list of the best suppliers first, followed by the list of the worst suppliers.

3. The input parameter month contains 0 if we want the data for the whole year, and a value 1 through 12 if we want the data for a given month.

4. Each token in the suppliers is separated by a forward slash (/).

5. We count the number of the best suppliers that are available so that we can return the actual number of suppliers in the inout parameter rank.

6. We follow the same logic to include the worst suppliers in the suppliers String.
Now we can compile and then copy the GetSupplier class to the function directory on the iSeries server.

**Stored procedure creation**

We register our stored procedure with the following CREATE PROCEDURE SQL statement:

```sql
CREATE PROCEDURE GET_SUPPLIER (in year integer, in month integer,
inout rank integer, out suppliers varchar(1000))
LANGUAGE JAVA PARAMETER STYLE JAVA NOT FENCED
EXTERNAL NAME 'GetSupplier!GetSupplier';
```

In the following section and in 7.9.3, "Java GUI client: ClientGetSupplierGUI" on page 210, we discuss the Java clients used to call this Java stored procedure. ClientGetSupplier is a text version, while ClientGetSupplierSwing is a GUI version of the Java client.

### 7.9.2 Java client: ClientGetSupplier

This client application is intended to call the GetSupplier stored procedures written in any language on both DB2 Universal Database for iSeries and DB2 Universal Database on other platforms. It also displays the result sets or the equivalent string output parameter.

This client code is text-based, so its size is limited. You can review its details in Example 7-12.

*Example 7-12  ClientGetSupplier client code*

```java
import java.math.*;
import java.util.*;
import java.io.*;
import java.sql.*;
import COM.ibm.db2.jdbc.app.*;
import com.ibm.as400.access.*;

class ClientGetSupplier
{
public static void main (String argv[])
{
Properties props = new Properties();
Connection con = null;
CallableStatement cs;
in rankCount;
boolean returnsRS;
String sql;
String suppliers = "";
try
{
props.load(new BufferedInputStream(new FileInputStream("logon.properties")));
String dbDriver = props.getProperty("dbDriver");
String dbUrl = props.getProperty("dbUrl");
String dbUser = props.getProperty("dbUser").trim();
String dbPassword = props.getProperty("dbPassword").trim();
String yearS = props.getProperty("year");
int year = Integer.parseInt(yearS);
System.out.println("Year : "+year);
String monthS = props.getProperty("month");
int month = 0;
try {month = Integer.parseInt(monthS);}
catch(NumberFormatException nfe) {};
System.out.println("Month : "+month);
String rankS = props.getProperty("rank");
int rank = Integer.parseInt(rankS);
System.out.println("Rank : "+rank);
String storedProc = props.getProperty("storedProc");
System.out.println("Stored procedure : "+storedProc);
```
String returnsRSS = props.getProperty("returnsResultSet");
if (returnsRSS.toUpperCase().startsWith("Y")) {returnsRS = true;}
else {returnsRS = false;}
Class.forName(dbDriver);
con = DriverManager.getConnection(dbUrl, dbUser, dbPassword);
System.out.println("got connection");

if (returnsRS) // we handle a stored procedure returning 2 result set
{
  String sql = "Call " + storedProc + " (?,?,?)";
  cs = con.prepareCall(sql);
  cs.setInt(1, year);
  cs.setInt(2, month);
  cs.setInt(3, rank);
  cs.registerOutParameter(3, Types.INTEGER);
  cs.execute();
  rank = cs.getInt(3);
  System.out.println();
  System.out.println("Available rank is : " + rank);
  System.out.println();
  boolean cursor;
  ResultSet brs = cs.getResultSet(); // best suppliers result set
  if (brs != null)
  {
    System.out.println("The best suppliers are : ");
    System.out.println("------------------------");
    cursor = brs.next();
    while (cursor)
    {
      System.out.println(brs.getString(1).trim() + " with a total sale of " + brs.getBigDecimal(2, 0));
      cursor = brs.next();
    }
    cs.getMoreResults();
  }
  ResultSet wrs = cs.getResultSet(); // the worst suppliers result set
  if (wrs != null)
  {
    System.out.println("The worse suppliers are : ");
    System.out.println("------------------------");
    cursor = wrs.next();
    while (cursor)
    {
      System.out.println(wrs.getString(1).trim() + " with a total sale of " + wrs.getBigDecimal(2, 0));
      cursor = wrs.next();
    }
    System.out.println("There is no second result.");
  }
  else {System.out.println("There is no result set.");}
}
else  // we handle a stored procedure returning a string
{
  String sql = "Call " + storedProc + " (?,?,?,?)";
  cs = con.prepareCall(sql);
  cs.setInt(1, year);
  cs.setInt(2, month);
  cs.setInt(3, rank);
  cs.registerOutParameter(3, Types.INTEGER);
  cs.registerOutParameter(4, Types.VARCHAR);
  cs.execute();
  rank = cs.getInt(3);
  suppliers = cs.getString(4);
System.out.println();
System.out.println("Available rank is "+rank);
    String[] suppliersAndSales = getTokens(suppliers);  
16
System.out.println();
System.out.println("The best suppliers are :");
System.out.println("------------------------");
for (int i=0; i<rank; i++)
    { rankCount = i+1;
        System.out.println(suppliersAndSales[2*i] + " with a total sale of " +
         suppliersAndSales[(2*i) + 1]);
    }
System.out.println();
System.out.println("The worst suppliers are :");
System.out.println("-------------------------");
for (int i=0; i<rank; i++)
    { rankCount = i+1;
        System.out.println(suppliersAndSales[(2*rank) + (2*i)] + " with a total sale of " +
         suppliersAndSales[(2*rank) + (2*i) + 1]);
    }
}

if (cs != null) cs.close();
if (con != null) con.close();
}
}

private static String[] getTokens(String enteredString)
{String[] enteredValues = null;
try
{StringTokenizer enteredStringTokenizer = new StringTokenizer(enteredString, "/", false);
    enteredValues = new String[enteredStringTokenizer.countTokens()];
    int j = 0;
    while (enteredStringTokenizer.hasMoreTokens())
    {enteredValues[j++] = enteredStringTokenizer.nextToken();
    }
}

catch (Exception e)
{ e.printStackTrace();
return enteredValues;
}
}

Notes
The following notes refer to Example 7-12.

1 and 2 The client can access both DB2 Universal Database for iSeries and DB2 Universal Database on other platforms. The classpath must contain the Toolbox driver for the iSeries server (jt400.jar) and the db2java.zip classes for DB2 Universal Database, as presented in 7.10, “GetSuppliers example: Implementation with result sets” on page 211. If you only want to access one of the two platforms, you can omit the corresponding import statement.

3 The application reads the information required to connect to the DB2 platform and to call the stored procedure from the properties file called logon.properties. The properties dbDriver, dbUrl, dbUser, and dbPassword are used to connect to the database using the JDBC driver. The storedProc property indicates the name of the
stored procedure to call. The SQL naming convention is used to find the stored procedure. The returnsResultSet property is set to Yes, if the stored procedure returns two result sets; and it is set to No for the Java stored procedure on the iSeries server that returns the suppliers output parameter containing the results. The last three properties are the input parameters passed to the stored procedure. The content of the properties file is shown in the following example:

```
#logon properties
# dbDriver=COM.ibm.db2.jdbc.app.DB2Driver
dbDriver=com.ibm.as400.access.AS400JDBCDriver
# dbUrl=jdbc:db2:db2local
dbUrl=jdbc:as400://AS400WS
dbUser=db2admin
dbPassword=db2admin
storedProc=GET_SUPPLIER
returnsResultSet=No
year=1999
month=3
rank=2
```

4 and 5 We make the distinction between the stored procedure returning two result sets and the one returning the suppliers parameter instead.

6 When calling a stored procedure with an inout parameter in a Java client, we first set its value as an input parameter and then declare that it is also an output parameter. Note that even if a Java stored procedure considers a result set as an output parameter, the corresponding client that calls it does not declare the result set as a parameter.

7 We get the result set returned by the stored procedure with the getResultSet() method of the CallableStatement class.

8 We can now handle this result set as any other result set returned by the executeQuery() method of the Statement class.

9 We read each supplier and its corresponding total sales amount from the result set and display the information. Note that an SQL decimal data type is mapped to the BigDecimal Java type. Also notice that we don’t want to receive any scale digits as indicated by the second parameter of the getBigDecimal() method.

10 The getMoreResults() method from the Statement class closes the current result set received with the previous getResultSet() method. It also tries to open the next result. It returns true if another result set is available.

11 Now that the statement is positioned at the next result set, we can access it by using the getResultSet() method.

12 In case the getMoreResult() method fails, it returns false and the result of the getResultSet() method is null.

13 We fetch the records and read the columns of the second result set as we did for the first one.

14 This is the second part of the application where we handle the iSeries server Java stored procedures returning a parameter containing the concatenated string representing the two result sets. Our next task is to decompose the received String and to display it.

15 We register and receive the fourth parameter.
Each element of the String is separated from the next by a special character that allows us to use the StringTokenizer class from the java.util package. We put each element in a String array.

We display the elements from this array.

With our classpath pointing to the driver implementations referred to in \texttt{1} and \texttt{2} and with the logon.properties file in our current directory, we compile and execute the ClientGetSupplier class. The results are shown here:

Year : 1999  
Month : 3  
Rank : 2  
Stored procedure : GET_SUPPLIER  
got connection

Available rank is 2

The best suppliers are :
------------------------
Black with a total sale of 8800  
Red with a total sale of 7345

The worst suppliers are :
------------------------
Yellow with a total sale of 1200  
Blue with a total sale of 3150

We now compare the output of this client with the GUI client.

7.9.3 Java GUI client: ClientGetSupplierGUI

The Java GUI client presented in this section interacts with the user through a visual interface. Here, we can enter the information placed in the logon.properties file in the previous section and display the results returned by the stored procedure. The GUI interface is similar to that of the Visual Basic client and is not presented in detail. It was developed with VisualAge® for Java (JDK 1.1.8). Figure 7-12 shows how it works.
This same GUI client is used to call the external SQL and Java stored procedures created on the iSeries server and the stored procedures created on DB2 Universal Database V6.1. We first enter the user, password, and JDBC URL to connect to the desired database. When connected, we enter the stored procedure that we want to execute and its parameters. Then, we call it by clicking the Query button.

7.10 GetSuppliers example: Implementation with result sets

The business logic for this example is discussed in 5.7, “GetSuppliers example” on page 111. The example was modified in 7.9, “GetSuppliers example: Implementation with no result sets” on page 204, to be run on the iSeries server V4R5. A Java stored procedure on iSeries server V5R1 or DB2 Universal Database Version 6 and later can return result sets. In this section, we present a Get Supplier Java stored procedure that returns two result sets, the best suppliers and the worst suppliers, respectively.

7.10.1 GetSuppliers stored procedure with the Java parameter style

Example 7-13 shows the GetSuppliers stored procedure with the Java parameter style. The numbered lines are explained in the notes that follow.

```java
import java.math.*;
import java.sql.*;

public class GetSupplierRS
{
    public static void GetSupplierRS (int year, int month, int[] rank,
    ResultSet[] bestSuppliers, ResultSet[] worstSuppliers)
        throws SQLException, Exception
    {
        Connection con = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement ps = null;
        PreparedStatement ps2 = null;
        ResultSet rs = null;
```
String sql;
int rowCount;
BigDecimal bestRankTotalSales = new BigDecimal("0");
BigDecimal worstRankTotalSales = new BigDecimal("0");

// we get the total sales amount of the best suppliers
if (month < 1)
    \{sql = "SELECT totalsales FROM yearsale WHERE (year = ?) ORDER BY totalsales DESC";\}
else
    \{sql = "SELECT totalsales FROM totalsale WHERE ((year = ?) AND (month = ?)) ORDER BY totalsales DESC";\}
ps = con.prepareStatement( sql );
ps.setInt( 1, year );
if (month > 0) \{ps.setInt( 2, month );\}
rs = ps.executeQuery();
rowCount = 0;
while ((rs.next()) && (rowCount<rank[0]))
    \{ bestRankTotalSales = rs.getBigDecimal(1, 2); \}
    rowCount++;
if (rowCount != rank[0]) \{rank[0] = rowCount;\}
try \{if (rs != null) \{rs.close();\}\}
catch (SQLException e) \{ /* ignore */ \};

// we get the total sales amount of the worst suppliers
if (month < 1)
    \{sql = "SELECT totalsales FROM yearsale WHERE (year = ?) ORDER BY totalsales ASC";\}
else
    \{sql = "SELECT totalsales FROM totalsale WHERE ((year = ?) AND (month = ?)) ORDER BY totalsales ASC";\}
ps = con.prepareStatement( sql );
ps.setInt( 1, year );
if (month > 0) \{ps.setInt( 2, month );\}
rs = ps.executeQuery();
rowCount = 0;
while ((rs.next()) && (rowCount<rank[0]))
    \{ worstRankTotalSales = rs.getBigDecimal(1, 2); \}
    rowCount++;
if (rowCount > rank[0]) \{rank[0] = rowCount;\}
try \{if (rs != null) \{rs.close();\}\}
catch (SQLException e) \{ /* ignore */ \};

if (month < 1)
    \{sql = "SELECT supplier_name, totalsales FROM yearsale " +
        "WHERE ((year = ?) AND (totalsales >= ?)) " +
        "ORDER BY totalsales DESC";\}
else
    \{sql = "SELECT supplier_name, totalsales FROM totalsale " +
        "WHERE ((year = ?) AND (month = ?) AND (totalsales >= ?)) " +
        "ORDER BY totalsales DESC";\}
ps = con.prepareStatement( sql );
ps.setInt( 1, year );
if (month > 0)
    \{ps.setInt( 2, month );\}
ps.setBigDecimal(3, bestRankTotalSales);
else {ps.setBigDecimal(2, bestRankTotalSales);}
bestSuppliers[0] = ps.executeQuery();
if (month < 1) {
  sql = "SELECT supplier_name, totalsales FROM yearsale " +
    "WHERE ((year = ?) AND (totalsales <= ?)) " +
    "ORDER BY totalsales ASC";
} else {
  sql = "SELECT supplier_name, totalsales FROM totalsale " +
    "WHERE ((year = ?) AND (month = ?) AND (totalsales <= ?)) " +
    "ORDER BY totalsales ASC";
}
ps2 = con.prepareStatement( sql );
ps2.setInt( 1, year );
if (month > 0) {
  ps2.setInt( 2, month );
  ps2.setBigDecimal(3, worstRankTotalSales);
} else {
  ps2.setBigDecimal(2, worstRankTotalSales);
}
worstSuppliers[0] = ps2.executeQuery();

Notes: The following notes refer to Example 7-13:

1. In the Java language, a result set is simply an instance of the java.sql.ResultSet class. As such, it is considered like another variable. When returned by a Java stored procedure using the Java parameter style, it is declared as any other output parameter, an array of size one. In our example, we declare that the method returns two result sets: bestSuppliers and worstSuppliers.

2. Because we want to return the first $n$ and the last $n$ suppliers, we first calculate the total sales amount achieved by the $n$th best and the $n$th worst supplier. Afterwards, we retrieve two result sets. The first contains the suppliers that have totalsales higher than or equal to the value for the $n$th best supplier. The rank is passed to the procedure as an input parameter.

3. The second result set contains the suppliers that have totalsales less than or equal to the value for the $n$th worst supplier.

4. Now that we have the value of the totalsales amount for the $n$th best supplier (bestRankTotalSales), we build the SQL SELECT statement that returns the first result set.

5. We get the result set with the executeQuery() method and assign it to the output parameter. The result set (or cursor) is automatically opened.

6. We follow the same logic for getting the second result set returning the worst suppliers.

7. We assign the second result set to the second output parameter.
**Stored procedure creation**

We register our stored procedure with the following CREATE PROCEDURE SQL statement:

```
CREATE PROCEDURE GET_SUPPLIER_RS (in year integer, in month integer, 
inout rank integer)
DYNAMIC RESULT SETS 2 LANGUAGE JAVA PARAMETER STYLE JAVA FENCED
EXTERNAL NAME 'GetSupplierRS!GetSupplierRS';
```

**Note:** In DB2 Universal Database, the Java stored procedure only returns result sets if FENCED is used in the CREATE PROCEDURE SQL statement. Specifying NOT FENCED prevents the stored procedure from returning any result set, but no error message is issued as a warning. In DB2 Universal Database for iSeries, this parameter is provided only for compatibility with other platforms and does not have any effect.

**7.10.2 GetSuppliers stored procedure with the DB2GENERAL parameter style**

Example 7-14 shows the GetSuppliers stored procedure with the DB2GENERAL parameter style. The numbered lines are explained in the notes that follow.

**Example 7-14 GetSuppliers stored procedure with the DB2GENERAL parameter style**

```java
import java.math.*;
import java.sql.*;
import com.ibm.db2.app.*;

public class GetSupplierResultSetDB2GENERAL extends StoredProc
{
    public void GetSupplierRS (int year, int month, int rank)
throws SQLException, Exception
    {
        Connection con = getConnection();
        PreparedStatement ps = null;
        PreparedStatement ps2 = null;
        ResultSet rs = null;
        String sql;
        int rowCount;
        BigDecimal bestRankTotalSales = new BigDecimal("0");
        BigDecimal worstRankTotalSales = new BigDecimal("0");

        // we get the total sales amount of the best suppliers 2
        if (month < 1)
        {
            sql = "SELECT totalsales FROM ordapplib.yearsale WHERE (year = ?) " +
            "ORDER BY totalsales DESC";
        }
        else
        {
            sql = "SELECT totalsales FROM ordapplib.totalsales WHERE ((year = ?) " +
            "AND (month = ?)) "ORDER BY totalsales DESC";
        }
        ps = con.prepareStatement(sql);
        ps.setInt(1, year);
        if (month > 0)
        {
            ps.setInt( 2, month );
        }
        rs = ps.executeQuery();
        rowCount = 0;
        while ((rs.next()) && (rowCount<rank))
        {
```
bestRankTotalSales = rs.getBigDecimal(1);
rowCount++;
}
if (rowCount != rank)
{
    rank = rowCount;
}
try {if (rs != null) {rs.close();}}
catch (SQLException e) { /* ignore */};

// we get the total sales amount of the worst suppliers
if (month < 1)
{
    sql = "SELECT totalsales FROM ordapplib.yearsale WHERE (year = ?) " +
          "ORDER BY totalsales ASC";
}
else
{
    sql = "SELECT totalsales FROM ordapplib.totalsale WHERE ((year = ?) " +
          "AND (month = ?)) ORDER BY totalsales ASC";
}
ps = con.prepareStatement(sql);
ps.setInt(1, year);
if (month > 0) {ps.setInt( 2, month );}
rs = ps.executeQuery();
rowCount = 0;
while ((rs.next()) && (rowCount<rank))
{ worstRankTotalSales = rs.getBigDecimal(1);
    rowCount++;
}
if (rowCount > rank) {rank = rowCount;}
try {if (rs != null) {rs.close();}}
catch (SQLException e) { /* ignore */};

if (month < 1)
{ sql = "SELECT supplier_name, totalsales FROM ordapplib.yearsale WHERE ((year = ?) " +
          "AND (totalsales >= ?)) ORDER BY totalsales DESC";}
else
{ sql = "SELECT supplier_name, totalsales FROM ordapplib.totalsale WHERE ((year = ?) " +
          "AND (month = ?) AND (totalsales >= ?)) ORDER BY totalsales DESC";}
ps = con.prepareStatement(sql);
ps.setInt(1, year);
if (month > 0)
{ ps.setInt(2, month);
    ps.setBigDecimal(3, bestRankTotalSales);
}
else {ps.setBigDecimal(2, bestRankTotalSales);}  
ps.execute();

if (month < 1)
{ sql = "SELECT supplier_name, totalsales FROM ordapplib.yearsale WHERE " +
          "((year = ?) AND (totalsales <= ?)) ORDER BY totalsales ASC";
}
else
{ sql = "SELECT supplier_name, totalsales FROM ordapplib.totalsale WHERE " +
          "((year = ?) AND (month = ?) AND (totalsales <= ?)) ORDER BY totalsales ASC";
}
ps2 = con.prepareStatement(sql);
ps2.setInt(1, year);
if (month > 0)
{ ps2.setInt(2, month);
  ps2.setBigDecimal(3, worstRankTotalSales);
}
else
{ ps2.setBigDecimal(2, worstRankTotalSales);
}
ps2.execute();
set(3, rank);  // returns number of retrieved rows

Notes: The main differences from the Java parameter style version are:
1. Import file that contains the StoredProc superclass.
2. A stored procedure class containing DB2GENERAL stored procedures must extend the com.ibm.db2.app.StoredProc class.
3. Methods that conform DB2GENERAL stored procedures are not static. IN, OUT, and INOUT parameters are defined like an input parameter (no array convention used).
4. Connection is established by the getConnection() method inherited from the StoredProc class.
5. OUT and INOUT parameters are returned by the set method inherited from the StoredProc class.

Stored procedure creation
We register our stored procedure with the following CREATE PROCEDURE SQL statement:

CREATE PROCEDURE GET_SUPPLIER_RS_DB2GENERAL
(in year integer, in month integer, inout rank integer)
DYNAMIC RESULT SETS 2 LANGUAGE JAVA PARAMETER STYLE DB2GENERAL FENCED
EXTERNAL NAME 'GetSupplierResultSetDB2GENERAL!GetSupplierRS';

Note: In DB2 Universal Database, the Java stored procedure only returns result sets if FENCED is used in the CREATE PROCEDURE SQL statement. Specifying NOT FENCED prevents the stored procedure from returning any result set, but no error message is issued as a warning. In DB2 Universal Database for iSeries, this parameter is provided only for compatibility with other platforms and does not have any effect.

7.10.3 Java clients: ClientGetSupplier and ClientGetSupplierGUI

We use the Java clients presented in 7.9.2, “Java client: ClientGetSupplier” on page 206, and 7.9.3, “Java GUI client: ClientGetSupplierGUI” on page 210, to test whether they work with the Java stored procedure created both on DB2 Universal Database V6.1 and on DB2 Universal Database for iSeries. The result is displayed in Figure 7-13.

As shown in Figure 7-13, the same client can be used to call stored procedures written in different languages and on different platforms. The same satisfactory results were also obtained with the text version ClientGetSupplier.
7.11 Problem determination

This section focuses on the debugging and the tracing possibilities of Java stored procedures.

7.11.1 Debugging

At first glance, debugging may seem slightly complicated because it requires the Java Virtual Machine (JVM™) to be started when trying to access the source. We found that the easiest way to debug was in a client/server environment. The following steps outline the process used in this environment:

1. Compile the Java stored procedure that you want to debug so that the class contains the debugging information. The compilation parameter is -g, for example:
   
   javac -g MyClass.java

2. Copy the .java source file with its corresponding .class file in the iSeries server function directory /Qibm/UserData/OS400/SQLLib/Function.

3. Find the full name of the job where the Java stored procedure will be executed. If you use the Java toolbox to connect to the iSeries server, the job is QZDASOINIT. The easiest way to find the right job is to use the command:
   
   WRKOBJLCK OBJ(*USERID) OBJTYPE(*USRPRF)

   Here, USERID is the user ID used to connect to the iSeries server. Note the job number and user profile.

4. Start servicing the job you found in step 3, for example, run:
   
   STRSRVJOB JOB(076853/QUSER/QZDASOINIT)

5. Make sure the JVM has been started in this job. For example, the client can call a dummy Java stored procedure doing nothing and then wait until you set up the debugging.

6. Start debugging for the desired class, for example:
   
   STRDBG CLASS(MyClass)
7. If everything is all right, you now see the Java source file on your 5250 emulation. You can step through it and add breakpoints the same as you would do with any other traditional language. Figure 7-14 shows a breakpoint that was just added.

8. After the breakpoint is set, leave the debug window by pressing F12, and let your client execute the Java stored procedure. For example, our client is waiting on a key to be pressed before calling the stored procedure.

9. On the 5250 emulation, the debug screen displays, and you can now debug your Java code.

10. After debugging, stop your debug and service sessions using these two CL commands:

```
    ENDDBG
    ENDSRVJOB
```

### 7.11.2 Tracing

The trace level component can be used to trace the actions that are performed by the OS/400 module responsible for the Java stored procedures support. However, its interpretation can be difficult and should be directed to IBM Support Services.

The trace is enabled by adding the environment variable `QIBM_COMPONENT_TRACE_LEVEL` with a value of ‘SQJAVA,3’ to the job where the Java stored procedure is executed. The trace is disabled by removing this environment variable.

After the trace is taken, it can be dumped with the CL command `DMPUSRTRC` either to STDOUT or to a file in QTEMP.
Illustration
To make it easier to start the trace, stop it, and obtain a spooled file when working from an SQL interface, we created two CL programs that we defined as two external stored procedures: SPTRACEON and SPTRACEOFF. The first program is used to start the trace. The second program is used to stop it and output it to a spooled file before the QTEMP library is cleaned. We can now easily call the first stored procedure to enable the tracing, then call the Java stored procedure we want to trace, and finally call the last stored procedure to stop the tracing and obtain the spooled file.

The code that makes up the SPTRACEON procedure is shown here:

```sql
PGM
ADDENVVAR ENVVAR(QIBM_COMPONENT_TRACE_LEVEL) + VALUE('SQJAVA,3')
MONMSG MSGID(CPFA980) EXEC(CHGENVVAR + ENVVAR(QIBM_COMPONENT_TRACE_LEVEL) + VALUE('SQJAVA,3'))
ENDPGM
```

The SPTRACEOFF code is shown here:

```sql
PGM
DMPUSRTRC
CPYF FROMFILE(QTEMP/QAP0ZDMP) TOFILE(*PRINT)
RMVENVVAR ENVVAR(QIBM_COMPONENT_TRACE_LEVEL)
MONMSG MSGID(CPFA981)
ENDPGM
```

The trace can be useful, for example, when the signature of the CREATE PROCEDURE SQL statement does not correspond to the signature of the Java method in the class. For example, the JavaInsertCus method in the previous example might incorrectly be defined with the following SQL statement:

```
CREATE PROCEDURE JAVAINSERTCUS(in s1 char(5), in s2 char(20), in s3 char(15), in s4 char(15), in s5 char(20), in s6 char(20), in s7 char(5), in bd1 decimal(11,2), in bd2 decimal(11,2), out insertCount char(5))
LANGUAGE JAVA PARAMETER STYLE JAVA NOT FENCED EXTERNAL NAME 'JavaInsertCus!JavaInsertCus';
```

At execution time, the SQL interface first checks whether the parameters passed by the SQL CALL match those declared by the CREATE PROCEDURE statement. If this is the case, the system tries to locate the JavaInsertCus method in the JavaInsertCus class with the corresponding parameters. This last step will fail with SQLException SQL0443: Trigger program or external routine detected an error. No more information is available concerning the cause of the error. It can prove difficult to determine what went wrong. By taking a component level trace, you can see that GetMethodID failed when looking for the ConvertUnicode method with a given signature in the ConvertUnicode class.

The trace is partially shown in Figure 7-15 and is further explained in the following notes.
Notes: The following notes refer to Figure 7-15:

1 and 2 Indicates that there is a problem finding a method in the JavaInsertCus class.
3 and 4 The JavaInsertCus method cannot be found in the JavaInsertCus class.
5 The JavaInsertCus method has the signature described in 6 through 7 and cannot be found.
Chapter 8. Stored procedure error handling

We can see error handling from two different, but complimentary, points of view. From the server point of view, we are interested in how to report errors to the caller and how to manage errors that occur inside a procedure. From the client perspective, we are interested in how to retrieve error and warning conditions.

We will review how to manage and report database error conditions. We also review how to manage database error and warning conditions on the client side.

Even if the tips and techniques shown in this chapter are related specifically with stored procedures on DB2 Universal Database for iSeries, many of the concepts are closely related to user-defined functions (UDFs).

This chapter covers:

- Database error reporting strategy
- Consistent error handling in stored procedures
- Error handling in:
  - SQL stored procedures
  - External stored procedures
  - Java stored procedures
- Retrieving error conditions in client applications
- Transaction management, COMMIT, SAVEPOINT and ROLLBACK usage
8.1 Database error reporting strategy

In the DB2 Universal Database family of database managers, two variables are used by the database management system (DBMS) to return feedback that we must be familiar with: SQLCODE and SQLSTATE. SQLCODE is the original way in which DB2 reports error and warning conditions. Each DBMS provider developed its own error code structure, making it difficult to build portable code that manages error conditions. But in SQL92, the error conditions were standardized for all of us. That standardized error condition code is called SQLSTATE; now we have a platform-independent error code structure.

When DB2 Universal Database for iSeries encounters an error, the SQLCODE returned is negative, and the first two digits of the SQLSTATE are different from '00', '01', and '02'. If SQL encounters a warning, but it is a valid condition while processing the SQL statement, the SQLCODE is a positive number, and the first two digits of the SQLSTATE are '01' (warning condition) or '02' (no data condition). When the SQL statement is processed successfully, the SQLCODE returned is 0, and SQLSTATE is '00000'.

8.1.1 User-defined errors and warnings

User-defined errors are certain conditions in a application that are defined as errors by the business logic rather than by the runtime environment. For example, there may be a business rule in your company that total compensation for an employee cannot exceed the compensation of the employee's manager. Therefore, a database routine (stored procedure or user-defined function) used to modify the compensation needs to check whether the new value complies with this company regulation. If the new value exceeds the limit, the routine needs to signal an error to the calling process.

The SQLSTATE error messages are five character codes in which the first two characters represents the nature of the error or warning, also called error class, and the last three characters represent the detailed error condition, also called error subclass. When the first two characters are “38”, the error condition is caused by an external function, which means a UDF, a stored procedure, or a trigger. Its commonly accepted to code user-defined SQLSTATES in the form 38yxx, where y can be any letter or number, and xx is any two digits or uppercase letters, taking care not to use one of the predefined SQLSTATES like 38502. (The external function is not allowed to execute SQL statements.) For more information about SQLCODEs and SQLSTATES, refer to DB2 Universal Database for iSeries SQL Messages and Codes, which is available on the iSeries Information Center.

You can also define user-defined warnings in a consistent way by using SQLSTATEs 01Hxx.

8.1.2 Consistent error handling

Currently, a growing number of development establishments deal with heterogeneous environments where existing applications need to be enhanced so that they can interact with newer solutions. Therefore, it is critical that you adopt a consistent approach for user-defined error handling that can be used across various stored procedure types. The major benefit of the proposed methodology is that the client application can be completely isolated from the implementation details of a stored procedure. At a certain point, an existing RPG stored procedure can be rewritten in SQL or Java with no implications for the client code.
8.2 Error handling in SQL stored procedures

This section describes different types of error handling statements for the SQL stored procedure, including:

- Condition and handler declarations
- SIGNAL and RESIGNAL statement
- SQLCODE and SQLSTATE variables
- GET DIAGNOSTICS EXCEPTION

8.2.1 Condition and handler declaration

Some typical situations that require explicit error handling include:

- In a sequential FETCH, record not found means that the cursor reached the end of the file.
- In an INSERT, there are duplicate key values or check constraint inconsistencies.
- In a DELETE, there is a referential integrity violation.
- In an UPDATE, there is a check constraint violation.
- In a SELECT, a row is not found.

You can handle these situations by declaring condition variables and a handler for each condition. The condition declaration allows you to declare a meaningful condition name for a corresponding SQLSTATE value. The following example illustrates how to code condition declarations:

```sql
DECLARE record_not_found
    CONDITION FOR '02000';
DECLARE check_constraint_error
    CONDITION FOR '23513';
```

The first condition declaration is called `record_not_found`, and it corresponds to SQLCODE +100, which has an SQLSTATE of ‘02000’. The second condition declaration is called `check_constraint_error`, and it corresponds to SQLCODE -545 and to SQLSTATE ‘23513’. This occurs when an UPDATE or INSERT violates a check constraint defined for one of the fields.
To use a condition name, you need to declare a handler. A condition handler is an SQL statement that is executed when an exception or completion condition occurs within the body of a compound statement. The actions specified in a handler can be any SQL statement, including a compound statement. The scope of a handler is limited to the compound statement in which it is defined. A handler declaration associates a handler with an exception or completion condition in a compound statement.

There are three types of condition handler:

- **CONTINUE**: When this condition is specified, after the SQL statement in the handler is successfully executed, the control is returned to the SQL statement following the one that raised the exception.
- **EXIT**: If EXIT is specified, once the SQL statements in the handler is successfully executed, the control is returned to the end of the compound statement that defines the handler.
- **UNDO**: When UNDO is specified, a rollback operation is performed within the compound statement and then the handler is invoked. When the handler is invoked successfully, control is returned to the end of the compound statement that defines the handler.

**Important**: The UNDO handler can be defined only in an ATOMIC compound statement.

Regarding the conditions that can cause a handler to be invoked, the DB2 SQL procedural language defines three general conditions that are associated with different SQLSTATEs. An SQLSTATE is a five-character string contained in the DB2 Communications Area (DB2 CA). DB2 runtime sets this value each time an SQL statement is executed. SQLSTATEs are consistent across all DB2 platforms. The general conditions are:

- **SQLEXCEPTION** specifies that the handler is invoked when an SQL exception occurs. This corresponds to an SQLSTATE with a class value other than ‘00’, ‘01’, and ‘02’. The SQLSTATE class is defined by its first two characters.
- **SQLWARNING** specifies that the handler is invoked when an SQL warning occurs. This corresponds to SQLSTATE class ‘01’.
- **NOT FOUND** specifies that the handler is invoked when a NOT FOUND condition occurs. This corresponds to SQLSTATE class ‘02’.

In addition, it is possible to provide handlers for a specific condition. For example, you may declare a handler for SQLSTATE ‘02505’, which corresponds to the duplicate key exception. Here’s a code snippet that illustrates the handler:

```sql
DECLARE EXIT HANDLER FOR '02505'
BEGIN
  GET DIAGNOSTICS EXCEPTION 1 SQLERRM = MESSAGE_TEXT;
  INSERT INTO jm_debug (SQLTEXT, T1) VALUES ('Level2 - Exit Handler for DUPLICATE_KEY: Error message: ' || SQLERRM, CURRENT_TIMESTAMP) WITH NC;
END;
```

The code fragment shows an exit handler for a specific SQL exception. It is invoked when a duplicate key violation occurs in the compound statement that contains the handler.

A better programming technique is to declare a condition name for a specific SQLSTATE to avoid hardcoding a particular SQLSTATE on a handler declaration. That way, the source code becomes easier to read and maintain. Consider the following code fragment:

```sql
DECLARE DUPLICATE_KEY CONDITION FOR SQLSTATE '02505';
DECLARE EXIT HANDLER FOR DUPLICATE_KEY ...
```
The condition declaration associates a meaningful, descriptive name with the SQLSTATE that it represents.

The following code snippet shows a typical block of statements at the beginning of a stored procedure that handles special conditions:

```sql
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET v_sqlcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR SQLWARNING
SET v_sqlcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR NOT FOUND
SET v_sqlcode = SQLCODE;
```

The handlers are invoked starting from the most specific to the most generic. Suppose that you have the following two declarations:

```sql
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET v_sqlcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505',
                  SQLSTATE '23510',
                  SQLSTATE '23511',
                  SQLSTATE '23512',
                  SQLSTATE '23513'
SET chk_constr_violation = TRUE;
```

If a check constraint is violated, firing any of the SQLSTATEs defined in the list, the handler in 2 is invoked. The generic handler for an SQL exception in 1 is called for all other exception states.

### Notes:
- If an unhandled SQL exception occurs within an SQL procedure, the execution of the procedure is terminated, and the SQLCODE is returned to the caller.
- The support of the definition of handlers for multiple conditions were added in V5R2.

The following example defines all key elements of the compound control statement, condition and handler:

```sql
DECLARE not_found CONDITION FOR '02000';
DECLARE c1 CURSOR FOR
    SELECT cusnbr, cuscrd
    FROM  ordapplib.customer;
DECLARE CONTINUE HANDLER FOR not_found
    SET at_end = 1;
```

In the example, the handler declaration sets the variable at_end to 1 if the condition not_found is true. The condition not_found occurs when SQLSTATE is equal to '02000'. After the variable at_end is set to 1, the control is returned to the SQL statement following the one that raised the condition.

Consider the following example:

```sql
DECLARE c1 CURSOR FOR
    SELECT cusnbr, cuscrd
    FROM  ordapplib.customer;
DECLARE UNDO HANDLER FOR SQLEXCEPTION
    SET errmsg = 'ERROR, ROLLBACK WAS ISSUED';
```
In this example, the handler is not associated with a condition declaration. Instead, if the error is an exception, the procedure rolls back (UNDO) all the transactions done in the compound statement, and errmsg is set to 'ERROR, ROLLBACK WAS ISSUED'. The control is returned to the end of the compound statement.

Now we are ready to complete our procedure.

Example 8-1  Error handling example

CREATE PROCEDURE CREDITP
(IN    i_perinc DECIMAL(3,2),
INOUT o_numrec DECIMAL(5,0))
LANGUAGE SQL
BEGIN atomic
DECLARE proc_cusnbr CHAR(5);
DECLARE proc_cuscrd DECIMAL(11,2);
DECLARE numrec DECIMAL(5,0);
DECLARE at_end INT DEFAULT 0;
DECLARE not_found CONDITION FOR '02000';
DECLARE c1 CURSOR FOR
SELECT cusnbr, cuscrd
FROM  ordapplib.customer;
DECLARE CONTINUE HANDLER FOR not_found
SET at_end = 1;
SET numrec = 0;
OPEN c1;
FETCH c1 INTO proc_cusnbr, proc_cuscrd;
WHILE at_end = 0 DO
SET proc_cuscrd = proc_cuscrd +(proc_cuscrd * i_perinc);
UPDATE ordapplib.customer
SET cuscrd = proc_cuscrd
WHERE CURRENT OF c1;
SET numrec = numrec + 1;
FETCH c1 INTO proc_cusnbr, proc_cuscrd;
END WHILE;
SET o_numrec = numrec;
CLOSE c1;
END

There are cases when you may need to execute more than one statement on the DECLARE of the handler, as shown in the following code snippet.

Example 8-2  Compound statement in error handlers

BEGIN
DECLARE SQLSTATE char(5);
DECLARE PrvSQLState char(5) DEFAULT '00000';
DECLARE ExceptState int;

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN
SET PrvSQLState = SQLSTATE;
SET ExceptState = TRUE;
END;
...
END

DB2 Universal Database for iSeries before V5R2 does not support nested compound statements. However, you can circumvent this limitation by coding a "dummy" loop that does the same as the previous example, using the following approach.
Example 8-3  Multiple statements in error handlers before V5R2

BEGIN
    DECLARE SQLSTATE char(5);
    DECLARE PrvSQLState char(5) DEFAULT '00000';
    DECLARE ExceptState int;

    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        ExceptHandler: LOOP
            SET PrvSQLState = SQLSTATE;
            SET ExceptState = TRUE;
            LEAVE ExceptHandler;
        END LOOP;

    ...

END;

8.2.2 SIGNAL and RESIGNAL

The SQL Persistent Stored Module (PSM) database language supports two programming constructs that can be used to handle the user defined errors: SIGNAL and RESIGNAL.

The SIGNAL statement signals an error or warning condition explicitly. It causes an error or warning to be returned with the specified SQLSTATE, along with the message text. If a handler is defined to handle the exception, it is called immediately by the SIGNAL statement, as shown in Example 8-4.

Example 8-4  Error raising using SIGNAL statement

CREATE PROCEDURE G10()
    LANGUAGE SQL
BEGIN
    DECLARE c1 CONDITION FOR SQLSTATE '38001';
    DECLARE CONTINUE HANDLER FOR C1
        INSERT INTO RESULT(proc,res) VALUES ('exec of G10','EXIT handler fired');
        INSERT INTO result(proc,res) VALUES ('exec of G10','START of Proc');
        SIGNAL SQLSTATE '38001';/*the handler will be fired by this statement*/
        INSERT INTO result(proc,res) VALUES ('exec of G10','END of Proc');
END;

After calling the G10 procedure, you see the entries log in the result table (Figure 8-1).

![Figure 8-1](image-url)  Results from the G10 procedure
If no handler is defined to catch the SQLSTATE in the SIGNAL statement, the exception is propagated to the caller, as shown in Example 8-5.

**Example 8-5  Error raising using SIGNAL statement - variation**

```
CREATE PROCEDURE G11()
    LANGUAGE SQL
BEGIN
    DECLARE c1 CONDITION FOR SQLSTATE '38001';
    INSERT INTO result(proc,res) VALUES ('exec of G11','START of Proc');
    SIGNAL SQLSTATE '38001'; /*the handler will be fired by this statement*/
    INSERT INTO result(proc,res) VALUES ('exec of G11','END of Proc');
END;
```

After calling the G11 procedure, you see the entries log in the result table (Figure 8-2).

![Figure 8-2  Results from the G11 procedure](image)

In the second case, the SQLSTATE is returned back to the caller application by placing the value in the SQLCA of the invoker. For example, the following embedded SQL RPGLE program can retrieve and display the returned SQLSTATE from the G11 SQL procedure:

```
C/EXEC SQL
C+ CALL G11()
C/END-EXEC
C     SQLSTT        DSPLY
C                   MOVE      *ON           *INLR
```

Note that any valid SQLSTATE value can be used in the SIGNAL statement. You are not limited to the sqlstates of '38yxx' class. However, for consistency reasons, we recommend that you use the '38yxx' sqlstate pattern also for SQL stored procedures. The additional advantage of this methodology is that it prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

The RESIGNAL statement can only be coded as part of the SQL PSM condition handler and is used to resignal an error or warning condition. It returns SQLSTATE and SQL Message text to the invoker.

Using the RESIGNAL statement without an operand causes the identical condition to be passed outwards. A RESIGNAL statement with an operand causes the original condition to be replaced with the new condition you specified. Consider Example 8-6.

**Example 8-6  Error raising using SIGNAL and RESIGNAL statements**

```
CREATE PROCEDURE G8()
    LANGUAGE SQL
BEGIN
    DECLARE not_found_text CHAR(70);
    DECLARE CONTINUE HANDLER FOR SQLSTATE '38TNF'
    BEGIN
        INSERT INTO result(proc,res) values('exec of G8','ErrHandler Fired');
        RESIGNAL SQLSTATE '38TNF' SET MESSAGE_TEXT=not_found_text;
        INSERT INTO result(proc,res) values('exec of G8','Stmt after resignal');
    END;
END;
```
Chapter 8. Stored procedure error handling

After calling the G8 procedure, you see the entries log in the result table (Figure 8-3).

<table>
<thead>
<tr>
<th>PROC</th>
<th>RES</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec of G8</td>
<td>Start</td>
</tr>
<tr>
<td>exec of G8</td>
<td>EndHandler Fired</td>
</tr>
</tbody>
</table>

**Figure 8-3 Result from the G8 procedure**

The following embedded SQL RPGLE program illustrates how to retrieve the user-defined SQLSTATE and the error message text returned from the G8 SQL procedure:

```sql
DErrMsg           S             52A
C/EXEC SQL
C+ CALL G8 ()
C/END-EXEC
C     SQLSTT        DSPLY
C                   EVAL      ErrMsg=%subst(SQLERM:1:52)
C     ErrMsg        DSPLY
C                   MOVE      *ON           *INLR
```

If the SQL store procedure returns the user-defined SQLSTATE and error message, the SQLCODE is set to -438 to indicate an error condition or +438 to indicate a warning. The native JDBC, the toolbox JDBC, and the ODBC drivers monitor for those SQLCODEs, retrieve the user-defined SQLSTATE from the SQLCA, return its value to the caller, and return the user-defined error message.

Let us look at some practical examples. We start with an SQL stored procedure implemented in SQL PSM. The routine is called MODSAL and it is used to modify an employee's salary. The personal data for employees, such as serial number, compensation details, and department number, is stored in the EMPLOYEE table. The DEPARTMENT table, in turn, contains the department information, including the department's manager serial number. The rows in EMPLOYEE and DEPARTMENT are related by the department number. The MODSAL SQL stored procedure implements a business rule that the total compensation of an employee must not exceed the compensation of his/her manager. The routine's logic checks if the rule is not compromised, and if so it signals an error condition to the calling
process. The SIGNAL/RESIGNAL statements are used to pass the user-defined errors to the calling process. The routine accepts two parameters: Employee number of type CHAR(5) and salary change of type DECIMAL(9,2). See Example 8-7. The numbered sections are explained further in the list that follows.

**Example 8-7  Stored procedure using SIGNAL and RESIGNAL**

```sql
create procedure db2user.modsal ( in i_empno char(6), in i_salary dec(9,2) )
language SQL
begin atomic

declare  v_job  char(8);
declare  v_salary dec(9,2);
declare  v_bonus dec(9,2);
declare  v_comm dec(9,2);
declare  v_mgrno char(6);
declare  v_mgrcomp dec(9,2);
-- Retrieve compensation details for an employee from employee table,
-- join by department number to department table to retrieve the
-- manager's employee number, use scalar subselect to retrieve manager's
-- compensation.
declare c1 cursor for
    select  job, salary, bonus, comm, d.mgrno,
          (salary+bonus+comm) from employee where empno = d.mgrno
    as mgrcomp
    from employee e, department d
    where empno = i_empno and e.workdept = d.deptno;
-- Declare handlers for user-defined error sql states
declare exit handler for sqlstate '38S01'
    signal sqlstate '38S01' set message_text='MODSAL: Compensation exceeds the limit.';

declare exit handler for sqlstate '02000'
    signal sqlstate '38S02' set message_text='MODSAL: Invalid employee number.';
[end callout B]
open c1;
fetch c1 into v_job, v_salary, v_bonus, v_comm, v_mgrno, v_mgrcomp;
close c1;
-- check, if the new compensation within the limit
if (i_empno <> v_mgrno) and ((v_salary + i_salary + v_bonus + v_comm) >= v_mgrcomp)
  then signal sqlstate '38S01';
end if;
update employee set salary = v_salary + i_salary where empno = i_empno;
end
```

**Code sample notes**
The following notes refer to Example 8-7.

1. If the business rule is compromised, sqlstate '38S01' is signaled. The control is transferred to the error handler defined for this state. Note that the SIGNAL could have included the message text and been signaled directly to the invoker.

2. This error handler defined for the '38S01' sqlstate signals the user-defined error condition. The RESIGNAL statement is used to reset the return sqlstate to '38S01'. It also sets the
diagnostic message. After the RESIGNAL is fired, the stored procedure immediately returns the specified error to the caller. Upon return the sqlcode is set to -438. Unlike the external stored procedure, the entire sqlerrmc element of the SQLCA area is available for the customized message. No truncation of the user-defined error message text occurs with SQL stored procedures.

The sqlstate '02000' is returned to the SQL SP if there is no data for the employee number passed as the first parameter. This condition may be thrown either by the FETCH or searched UPDATE statement. The error handler handles this condition by signaling sqlstate '38S02' to the caller.

For a more detailed explanation of error handling, refer to the white paper Take Advantage of User-Defined SQL Errors, by Jarek Miszczyk. You can find this paper on the Web at:


8.2.3  SQLCODE and SQLSTATE variable in the SQL procedure

You might find it useful to examine and manipulate the SQLCODE and SQLSTATE values in your SQL procedure. To access the SQLCODE and SQLSTATE values, you must declare the following SQL variables in the SQL procedure body:

DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

After the variables are declared, DB2 Universal Database for iSeries sets these local variables after the execution of each SQL statement. Since the local SQLCODE and SQLSTATE variables are reset after each statement, their values should be copied to other local variables. Here is an example of using condition handlers to assign the value of the SQLSTATE and SQLCODE variables to a local variables:

DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET retcode=SQLCODE;
DECLARE CONTINUE HANDLER FOR SQLWARNING SET retcode=SQLCODE;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET retcode=SQLCODE;

8.2.4  Returning values using the RETURN statement

A return value can be used to return a program code to a caller application. For example, you may use the return value to inform the caller application as to whether the SQL procedure completed successfully. In Example 8-8, if no records were updated, the procedure returns -1. Otherwise, it returns 0 to represent success.

Example 8-8  Usage of the RETURN statement in SQL stored procedures

CREATE PROCEDURE more_credit(city char(20))
LANGUAGE SQL
BEGIN
  DECLARE num_records INTEGER;
  UPDATE CUSTOMER SET cuscrd=cuscrd * 1.05 WHERE CUSCTY= city;
  GET DIAGNOSTICS num_records = ROW_COUNT;
  IF (num_records > 0) then
    RETURN 0;
  ELSE
    RETURN -1;
  END IF;
END
The RETURN value can be examined by the caller with the GET DIAGNOSTIC statement. See “RETURN_STATUS” on page 233 for a coding example. You can also retrieve the return value directly from the SQLCA area by reading the value of sqlerrd[0].

**Note:** The RETURN value is supported in the latest Open Source JDBC driver, which is available for download on the Web at: [http://www.iSeries.ibm.com/toolbox/](http://www.iSeries.ibm.com/toolbox/)

### 8.2.5 GET DIAGNOSTICS

The GET DIAGNOSTIC statement can be used in several different ways. The following sections explain the possible forms of this statement.

#### EXCEPTION

The GET DIAGNOSTICS EXCEPTION statement is used to access information that is associated with an error or warning from the SQLCA of the procedure. In most cases, it is used as the first statement in a handler to determine what happened. For example, the error handling procedure in Example 8-9 writes SQLSTATE and the error message text to the errorlog table.

*Example 8-9  Usage of GET DIAGNOSTICS in SQL PSL*

```sql
CREATE PROCEDURE GetDiag()
LANGUAGE SQL
BEGIN
  DECLARE msgtxt CHAR(70);
  DECLARE msgtxtlen INTEGER;
  DECLARE PrevSQLState CHAR(5);
  DECLARE SQLSTATE CHAR(5);
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  BEGIN
    GET DIAGNOSTICS EXCEPTION 1
    msgtxt=MESSAGE_TEXT, msgtxtlen=MESSAGE_LENGTH;
    SET PrevSQLState=SQLSTATE;
    INSERT INTO errorlog VALUES(PrevSQLState,msgtxt);
  END;
  INSERT INTO result(proc,res) values('exec of GetDiag','Start');
  INSERT INTO unknown values('TEST');
END;
```

**Notes:** The following notes refer to Example 8-9:

1. Every SQL statement implicitly sets the SQLSTATE variable, if it is declared.
2. When the error handler is invoked, it writes an entry to the error log. In this example, the error log entry reads:
   
   `'42704', "UNKNOWN in ORDENTLIB type *FILE not found."`.
3. The INSERT statement tries to insert a row into a non-existing table. This action invokes an SQL error, which is handled by the error handler.

**Important:** GET DIAGNOSTICS EXCEPTION is one statement that does not reset the SQL state (which is a field in the SQLCA structure). Consequently, GET DIAGNOSTICS EXCEPTION should generally be the first statement in a condition handler, followed immediately by an assignment statement that saves the SQLState value to a local variable.
**ROW_COUNT**

ROW_COUNT is a new feature of the GET DIAGNOSTICS statement available since V4R4. It allows you to retrieve the number of rows affected by an INSERT, UPDATE, or DELETE statement. The numrec procedure shown in Example 8-10 is used to increase the credit of a customer in the city of Rochester by 5 percent. GET DIAGNOSTIC is used to assign the number of records updated to a variable num_records, which may, in turn, be returned to the caller application.

*Example 8-10  Usage of ROW_COUNT in SQL PSL*

```sql
CREATE PROCEDURE numrec(OUT num_records INTEGER) LANGUAGE SQL BEGIN 
    UPDATE CUSTOMER SET cuscrd=cuscrd * 1.05 WHERE CUSCTY='ROCHESTER'; 
    GET DIAGNOSTICS num_records=ROW_COUNT; 
    ...
END
```

This implementation is slightly different from other database implementations since it is not affected by SELECT.

**RETURN_STATUS**

The RETURN_STATUS is used to examine the return value of the previous CALL statement to an SQL procedure. The update_total procedure shown in Example 8-11 attempts to increase the credit limit for all customers in Rochester.

*Example 8-11  Usage of RETURN_STATUS in SQL PSL*

```sql
CREATE PROCEDURE update_total(IN cusnbr char(5)) LANGUAGE SQL BEGIN 
    DECLARE retval INTEGER DEFAULT 0; 
    ...
    SET retval = 0; 
    IF (cus_total + new_purchase < cus_credit) THEN 
        CALL more_credit('ROCHESTER'); 
        GET DIAGNOSTIC retval = RETURN_STATUS; 
        IF retval <> 0 THEN 
            GOTO BadNews; 
        END IF; 
    END IF; 
    ...
BadNews: 
    RETURN -1; 
END
```

**Notes:** The following notes refer to Example 8-11:

1. The more_credit SQL procedure is called with the city parameter set to 'ROCHESTER'.
2. The GET DIAGNOSTIC statement is used to retrieve the return value directly after the call statement executes.
3. If the return value indicates an error, the control is transferred to the error handling block.
8.2.6 Error handling in nested compound statements

When nested compound statements are used, each compound statement has its own scope for variable definitions as well as for its condition definitions and error handlers. Example 8-12 illustrates the scope of different error handlers.

Example 8-12 Error handlers in nested compound statement

CREATE PROCEDURE ERROR_HANDLERS(IN PARAM INTEGER)
LANGUAGE SQL
SET OPTION DBGVIEW=*SOURCE, OUTPUT=*PRINT
BEGIN
  DECLARE I INTEGER;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE EXIT HANDLER FOR
    SQLSTATE VALUE '38H02',
    SQLSTATE VALUE '38H04',
    SQLSTATE VALUE '38HI4',
    SQLSTATE VALUE '38H06'
  BEGIN
    DECLARE TEXT VARCHAR(70);
    SET TEXT = SQLSTATE || ' RECEIVED AND MANAGED BY OUTER ERROR HANDLER' ;
    RESIGNAL SQLSTATE VALUE '38HE0'
    SET MESSAGE_TEXT = TEXT;
  END;
  BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H03'
    RESIGNAL SQLSTATE VALUE '38HI3'
    SET MESSAGE_TEXT = '38H03 MANAGED BY INNER ERROR HANDLER';
    DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H04'
    RESIGNAL SQLSTATE VALUE '38HI4'
    SET MESSAGE_TEXT = '38H04 MANAGED BY INNER ERROR HANDLER';
    DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H05'
    RESIGNAL SQLSTATE VALUE '38HI5'
    SET MESSAGE_TEXT = '38H05 MANAGED BY INNER ERROR HANDLER';
    CASE PARAM
    WHEN 1 THEN
      SIGNAL SQLSTATE VALUE '38H01'
      SET MESSAGE_TEXT = 'EXAMPLE 1: ERROR SIGNALED FROM INNER COMPOUND STMT';
    WHEN 2 THEN
      SIGNAL SQLSTATE VALUE '38H02'
      SET MESSAGE_TEXT = 'EXAMPLE 2: ERROR SIGNALED FROM INNER COMPOUND STMT';
    WHEN 3 THEN
      SIGNAL SQLSTATE VALUE '38H03'
      SET MESSAGE_TEXT = 'EXAMPLE 3: ERROR SIGNALED FROM INNER COMPOUND STMT';
    WHEN 4 THEN
      SIGNAL SQLSTATE VALUE '38H04'
      SET MESSAGE_TEXT = 'EXAMPLE 4: ERROR SIGNALED FROM INNER COMPOUND STMT';
    ELSE
      SET I = 1; /*Don't do anything */
    END CASE;
  END;
  CASE PARAM
  WHEN 5 THEN
    SIGNAL SQLSTATE VALUE '38H05'
    SET MESSAGE_TEXT = 'EXAMPLE 5: ERROR SIGNALED FROM OUTER COMPOUND STMT';
  WHEN 6 THEN
    SIGNAL SQLSTATE VALUE '38H06'
    SET MESSAGE_TEXT = 'EXAMPLE 6: ERROR SIGNALED FROM OUTER COMPOUND STMT';
END;
ELSE
    SET I = 1;  /* Don't do anything */
END CASE;
END;

The expected behavior is described in Table 8-1.

<table>
<thead>
<tr>
<th>PARAM value</th>
<th>Expected behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Error 38H01 is fired from the internal compound statement. That error is not handled by any error handler and will be passed back to the caller program.</td>
</tr>
<tr>
<td>2</td>
<td>Error 38H02 is fired from the internal compound statement. That error is not managed by any error handler in the internal compound statement, but is handled by an error handler in the external error handler, which will fire error 38HE0 that will pass back to the caller.</td>
</tr>
<tr>
<td>3</td>
<td>Error 38H03 is fired from the internal compound statement. That error is managed by an error handler in the internal compound statement, firing error 38HI3. This new error will not be handled by any error handler and will be received by the caller.</td>
</tr>
<tr>
<td>4</td>
<td>Error 38H04 is fired from the internal compound statement. That error will be managed by an error handler in the internal compound statement, firing error 38HI4. Error 38HI4 will be managed by the error handler in the external error handler, firing error 38HE0 to the caller.</td>
</tr>
<tr>
<td>5</td>
<td>Error 38H05 is fired in the external compound statement. This error will not be managed by any error handler and the error will be passed back to the caller.</td>
</tr>
<tr>
<td>6</td>
<td>Error 38H06 is fired in the external compound statement. This error will be managed by the external error handler, which will fire error 38HE0 to the caller.</td>
</tr>
<tr>
<td>7</td>
<td>The stored procedure will terminate without errors.</td>
</tr>
</tbody>
</table>

In the following snippet, when statement 2 causes a NOT FOUND condition, the defined handler captures the exception and, after finishing its operation, it exits the nested compound exception, not the whole program, continuing with stmt 4.

...  
BEGIN
    DECLARE EXIT HANDLER FOR NOT FOUND
    SET I = 1;  /* DON'T DO NOTHING */
    stmt 1
    stmt 2  /* FIRING NOT FOUND CONDITION */
    stmt 3
END;
stmt 4
...

To illustrate further the behavior of condition handlers in nested compound statements, we examine one more stored procedure named p_nested_test(). This procedure manipulates data in a sample table called COFFEES. First the routine calculates the average price of coffee brands contained in the table, and then it inserts a new row.
Example 8-13 shows the source code listing of the procedure.

**Example 8-13  p_nested_test() procedure**

```sql
CREATE PROCEDURE SQLTUTOR.P_NESTED_TEST ( IN P_TABLE_NAME VARCHAR(128),
OUT P_ERROR_IND_OUT CHARACTER(1) )

LANGUAGE SQL
SPECIFIC P_NESTED_TEST
Level_1 :
BEGIN -- Main Procedure Body; Level-1 Compound Statement
DECLARE V_REF_CURSOR_TEXT VARCHAR ( 1024 ) ;
DECLARE V_SQL_STMT_EXEC1 VARCHAR ( 1024 ) ;
DECLARE SQLERRM VARCHAR ( 4000 ) DEFAULT '' ;
DECLARE V_AVG_PRICE DOUBLE PRECISION ;
DECLARE V_ROWS_INSERTED INTEGER DEFAULT 0 ;
DECLARE OBJECT_NOT_FOUND CONDITION FOR SQLSTATE '42704' ;
DECLARE COFFEES_QUERY_FAILED CONDITION FOR SQLSTATE '70010' ;
DECLARE COFFEES_UNKNOWN_AVG_PRICE CONDITION FOR SQLSTATE '70019' ;
DECLARE COFFEES_INSERT_FAILED CONDITION FOR SQLSTATE '70020' ;
DECLARE C_GET_COFFEES CURSOR FOR V_CUR_STMT ;
-- Exit handler scoped to the main procedure body
DECLARE EXIT HANDLER FOR SQLEXCEPTION -- [2.3] & [3.2]
BEGIN
GET DIAGNOSTICS EXCEPTION 1 SQLERRM = MESSAGE_TEXT ;
SET P_ERROR_IND_OUT = 'Y' ;
INSERT INTO JM_DEBUG ( SQLTEXT )
VALUES ( 'Level_1-Exit Handler for sqlexception: Message : ' || SQLERRM ) WITH NC ;
RESIGNAL ;
END ;
-- Level_1 compound statement body starts here
SET V_REF_CURSOR_TEXT = 'SELECT avg(price) FROM ' || TRIM ( P_TABLE_NAME ) ;
PREPARE V_CUR_STMT FROM V_REF_CURSOR_TEXT ;
INSERT INTO JM_DEBUG ( SQLTEXT )
VALUES ( 'Level_1-Main Procedure Body: V_CUR_STMT prepared' ) ;
-- Level_2_1 compound statement
Level_2_1:
BEGIN
-- exit handler scoped to compound statement Level_2_1
DECLARE EXIT HANDLER FOR SQLEXCEPTION --[2.2]
BEGIN
GET DIAGNOSTICS EXCEPTION 1 SQLERRM = MESSAGE_TEXT ;
INSERT INTO JM_DEBUG ( SQLTEXT )
VALUES ( 'Level_2_1-Exit Handler for Select: Message: ' || SQLERRM ) WITH NC ;
SIGNAL COFFEES_QUERY_FAILED SET MESSAGE_TEXT = SQLERRM ;
END ;
-- continue handler scoped to compound statement Level_2_1
DECLARE CONTINUE HANDLER FOR COFFEES_UNKNOWN_AVG_PRICE -- [1.2]
BEGIN
GET DIAGNOSTICS EXCEPTION 1 SQLERRM = MESSAGE_TEXT ;
INSERT INTO JM_DEBUG ( SQLTEXT )
VALUES ( 'Level_2_1-Handler for SQLSTATE 70019: Message: ' || SQLERRM ) WITH NC ;
SET V_AVG_PRICE = 0.0 ;
END ;
-- Level_2_1 compound statement body starts here
OPEN C_GET_COFFEES ; --[2.1]
FETCH C_GET_COFFEES INTO V_AVG_PRICE ;
CLOSE C_GET_COFFEE ;
IF V_AVG_PRICE IS NULL THEN --[1.1]
```

236  *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*
SIGNAL COFFEES_UNKNOWN_AVG_PRICE;
SET MESSAGE_TEXT = 'Unknown avg price of coffee.';
END IF;
INSERT INTO JM_DEBUG (SQLTEXT) VALUES ('Level_2_1 - Body: v_avg_price = ' || TRIM ( CHAR (v_avg_price)) ) WITH NC;
END Level_2_1;
-- Level_1 body resumes here
INSERT INTO JM_DEBUG(SQLTEXT) VALUES ('Level_1-Resuming processing after end of Level_2_1 compound statement.' ) WITH NC;
SET V_SQL_STMT_EXEC1 = 'INSERT INTO ' || TRIM ( P_TABLE_NAME ) || ' VALUES(10, 'Colombian Supreme', 10, 9.95, 1000, 1000)';
Level_2_2:
BEGIN
-- exit handler scoped to compound statement Level_2_1
DECLARE EXIT HANDLER FOR OBJECT_NOT_FOUND BEGIN
GET DIAGNOSTICS EXCEPTION 1 SQLERRM = MESSAGE_TEXT;
INSERT INTO JM_DEBUG (SQLTEXT) VALUES ('Level_2-2-Handler for Insert: Message: ' || SQLERRM) WITH NC;
SIGNAL COFFEES_INSERT_FAILED SET MESSAGE_TEXT = SQLERRM;
END;
-- Level_2_2 compound statement body starts here
EXECUTE IMMEDIATE V_SQL_STMT_EXEC1;
GET DIAGNOSTICS V_ROWS_INSERTED = ROW_COUNT;
INSERT INTO JM_DEBUG (SQLTEXT) VALUES ('Level_2-2-Main Body: ' || TRIM(CHAR(V_ROWS_INSERTED)) || ' row(s) inserted in COFFEES.') WITH NC;
END Level_2_2;
-- Level_1 body resumes here
INSERT INTO JM_DEBUG (SQLTEXT) VALUES ('Level_1-Resuming processing after end of Level_2_2 compound statement.' ) WITH NC;
SET P_ERROR_IND_OUT = 'N';
END level_1;

The procedure contains four handlers:

- An exit handler defined in Level_1 compound statement:
  The scope of this handler is the entire stored procedure body.

- An exit and a continue handler defined in Level_2_1 compound statement
  The scope of these handlers is limited to the compound statement in which they were declared.

- An exit handler defined in Level_2_2 compound statement
  The scope of this handler is limited to the compound statement in which it was declared. It is not visible to the statements contained in Level_2_1.

- To facilitate the analysis, we trace the flow of the control during the execution by writing messages into a separate table named jm_debug. This is a fairly common debugging technique used by SQL PL developers.

Let us consider the test scenarios in the following sections for the stored procedure execution to see how various programming constructs interact with each other.
Test scenario 1

In this scenario, we assume that the COFFEE table initially contains no rows, and we call the procedure with the following parameters:

CALL P_NESTED_TEST('COFFEES', ' ');

The first parameter is an input value containing the name of the table to be manipulated. The second parameter is an output parameter and it returns an error indicator value ('N' for no errors, and 'Y' when errors occurred). The invocation completes successfully with error indicator set to 'N'. The jm_debug contains the entries shown in Figure 8-4.

<table>
<thead>
<tr>
<th>SQLTEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level_1-Main Procedure Body: V_CUR_STMT prepared</td>
</tr>
<tr>
<td>Level_2_1-Handler for SQLSTATE 70019: Message: Unknown avg price of coffee.</td>
</tr>
<tr>
<td>Level_2_1 - Body: v_avg_price = 0E0</td>
</tr>
<tr>
<td>Level_1-Resuming processing after end of Level_2_1 compound statement.</td>
</tr>
<tr>
<td>Level_2_2-Main Body: 1 row(s) inserted in COFFEES.</td>
</tr>
<tr>
<td>Level_1-Resuming processing after end of Level_2_2 compound statement.</td>
</tr>
</tbody>
</table>

Figure 8-4 Message file from Test scenario 1

The execution proceeds successfully until the IF statement at \[1.1\] in Example 8-13 on page 236 is reached in compound statement Level_2_1. Since the COFFEES table is empty at this time, the V_AVG_PRICE variable is set to NULL (unknown). It causes the COFFEES_UNKNOWN_AVG_PRICE signal to be fired. This error condition corresponds to a custom SQLSTATE of '70019' (see the condition declarations section in the main stored procedure body). The DB2 runtime first tries to locate a condition handler for this particular condition within the Level_2_1 compound statement. It finds a continue handler at \[1.2\]. The handler is invoked, and the V_AVG_PRICE is set. The control is returned to statement \[1.3\], which is the next statement after the SIGNAL statement that raised the exception. The execution successfully continues until the end of the main procedure body is reached. Note that compound statement Level_2_2 inserts a new row into COFFEES.

This example illustrates how to use a custom SQLSTATE and a continue handler to deal with user-defined error conditions in SQL PL. User-defined errors are certain conditions in an application that are defined as errors by the business logic rather than errors generated by DB2 Universal Database for iSeries or OS/400. In our case, the business rule is that the unknown value of the average coffee price is not allowed.

For a more detailed explanation of error handling, refer to the white paper *Take Advantage of User-Defined SQL Errors*, by Jarek Miszczyn. You can find this paper on the Web at:


Test scenario 2

This time we call the stored procedure with an intentionally corrupted first parameter:

CALL P_NESTED_TEST('CO$$EES', ' ');

The stored procedure returns with the following error messages:

SQL State: 70010
Vendor Code: -438
Message: [SQL0438] CO$$EES in SQLTUTOR type *FILE not found.
The jm_debug contains the entries shown in Figure 8-5.

| Level_1-Main Procedure Body: V_CUR_STMT prepared |
| Level_2-1-Exit Handler for Select: Message: COFFEES in SGLTUTOR type *FILE not found. |
| Level_1-Exit Handler for sqlexception: Message: COFFEES in SGLTUTOR type *FILE not found. |

*Figure 8-5  Message file from Test scenario 2*

After invocation, the execution proceeds until the OPEN cursor statement is reached at [2.1] in Example 8-13 on page 236. Since the table name is corrupted, the DB2 runtime fails to find the corresponding table object and it throws an SQL exception with SQLSTATE ‘42704’, indicating that the table was not found. Since the Level_2_1 compound statement does not contain a handler for that specific condition, a general handler for all exceptions is invoked at [2.2]. The handler writes a debug row and resignals the error with the COFFEES_QUERY_FAILED condition mapped to custom SQLSTATE ‘70010’. The control is transferred to the end of the compound statement Level_2_1 and thus returns to the main procedure body. The error condition signaled in the Level_2_1 is pending, so now the runtime tries to locate an appropriate handler in the Level_1 compound statement (main procedure body). There is no handler for a specific condition COFFEES_QUERY_FAILED, but the runtime finds the general exception handler at [2.3]. After writing a debug message into jm_debug, the handler resignals the error condition that is returned to the caller and manifests itself in the error messages shown previously.

**Test scenario 3**

Finally, we call the stored procedure again with the following parameter:

```
CALL P_NESTED_TEST('COFFEES', ' ');
```

The stored procedure returns with the following error messages:

SQL State: 23505  
Vendor Code: -803  
Message: [SQL0803] Duplicate key value specified.

The jm_debug contains the entries shown in Figure 8-6.

| Level_1-Main Procedure Body: V_CUR_STMT prepared |
| Level_2-1-Body: v_avg_price = 9.9999999999993E0 |
| Level_1-Resuming processing after end of Level_2_1 compound statement |
| Level_1-Exit Handler for sqlexception: Message: Duplicate key value specified. |

*Figure 8-6  Message file from Test scenario 3*

After invocation, the execution proceeds until the EXECUTE IMMEDIATE statement is reached at [3.1] in Example 8-13 on page 236. The first column of the COFFEES table is defined as the primary key. The row with the key value 10 in the first column already exists because it was inserted when Test scenario 1 was run. Therefore, DB2 runtime throws a duplicate key exception that corresponds to SQLSTATE '23505'. First, the DB2 runtime tries to locate a handler for this specific condition in the compound statement Level_2_2. It finds none. Then, the runtime searches for a general handler, which also does not exist in Level_2_2. Control returns to the main procedure body with the error condition pending. Now the runtime locates the general handler for SQL exceptions and invokes it at [3.2]. After writing a debug message, the handler resignals the original error thrown in the inner Level_2_2 compound statement.
Using a nested error handler to avoid locks caused by cursors left open

As described in “Avoiding orphaned locks caused by a cursors left open” on page 32, the open cursor results in a read lock held over the data space. Under the circumstances described in the reference section, there can be locks that persist as long a given job (activation group) is active. We illustrate this situation with an example.

Suppose that we run the procedure shown in Example 8-14 from the Run SQL Scripts utility of iSeries Navigator.

Example 8-14   Example with cursor left open

```sql
create procedure testopencurs()
language sql
pl: begin
declare v_cof_name varchar(32);
declare c1 cursor for
    SELECT cof_name FROM COFFEES;
declare exit handler for sqlexception
    resignal;
    OPEN c1;
    FETCH c1 into v_cof_name;
    COMMIT HOLD;
    SIGNAL SQLSTATE '70000'
    set message_text='Exit before cursor closed';
    CLOSE c1;
END p1;
```

In Example 8-14, note the following points:

- Cursor c1 is opened at 1.
- At 2, the transaction is committed with hold so that the cursors are not closed.
- At 3, an exception is signaled so that control is transferred to the exit handler at 4. The exit handler resignals the exception, and the procedure returns to the caller. The cursor c1 is still left open.

The Run SQL Script utility does not allow you to explicitly close this cursor. To remove the lock, you must close the connection, which releases the QZDASOINIT job that is associated with your Run SQL Scripts session.

To avoid this situation, you need to explicitly close the cursor in the stored procedure. The code example in Example 8-15 illustrates how this can be achieved with the nested error handler.

Example 8-15   Avoiding cursor being left open example

```sql
create procedure testopencurs()
language sql
pl: begin
declare v_cof_name varchar(32);
declare c1 cursor for
    SELECT cof_name FROM COFFEES;
```
Chapter 8. Stored procedure error handling

In Example 8-15, note the following points:

- The handler intercepts the signal at 1.
- At 2, we attempt to close the cursor. If the cursor is open, the close succeeds and the resignal is executed. If the cursor is already closed, the statement at 2 throws an SQL exception with SQL state of '24501'. This is intercepted by the inner continue handler at 3.
- After ignoring the exception, the control returns to the next statement after 2, which is resignal.
- The SET statement in the nested handler is used as a workaround for a “do nothing” statement that currently does not exist in SQL PL.

8.2.7 Use nested compound statements for better performance

You should use nested compound statements to localize exception handling and cursors. If several specific handlers are specified, code is generated to check to see if the error occurred after each statement. Code is also generated to close cursors and process savepoints if an error occurs in a compound statement. In routines with a single compound statement with multiple handlers and multiple cursors, code is generated to handle these after every SQL statement. If you scope the handlers and cursors to a nested compound statement, the handlers and cursors are only checked within the nested compound statement. The following code snippet illustrates this idea:

```sql
... GENERAL: BEGIN
 DECLARE EXIT HANDLER FOR SQLEXCEPTION
 INNERLABEL: BEGIN
 DECLARE EXIT HANDLER FOR SQLSTATE '22H11'
 DECLARE C1 CURSOR FOR SELECT CUSTOMER_NAME FROM CUSTOMER;
 OPEN C1
 CLOSE C1
 OPEN C1
 END INNERLABEL
...
```

In the previous code snippet, SQLSTATE = 22H11 is only checked in the INNERLABEL compound statement.
8.3 Error handling in external stored procedures

External stored procedures must also manage error conditions. Ideally, all errors should be trapped and corrective action should be taken when required in the stored procedure. However, there are situations when the error condition cannot be handled within the stored procedure. In these cases, the stored procedure should propagate the error condition to the calling program.

Important: If the external stored procedure completes with an error, the state of the output parameters are undefined. There is no guarantee that the returned values are valid.

To return a user-defined SQL state and error message, an external stored procedure needs to use the SQL or DB2SQL parameter style. If the SQL parameter style is specified by an external stored procedure, DB2 Universal Database for iSeries passes to the routine a number of parameters in addition to those specified on the parameter list, as explained in 6.2.1, “SQL parameter style” on page 130. The DB2SQL parameter style is an extension to the SQL parameter style in that not only the SQL extra parameters are passed to the external procedure, but also the DBINFO data structure, as explained in 6.2.2, “DB2SQL parameter style” on page 131.

The SQL parameter style has the following structure:

| IN | OUT | INOUT argument [repeated], |
| OUT Sqlstate, |
| IN Procedure name, |
| IN Specific name, |
| OUT Diagnostic message |

Two parameters are of special interest for user-defined error handling:

- Sqlstate: This output parameter can be set by the external stored procedure to signal a successful execution, warning, or error condition. For valid SQLSTATEs, refer to 8.1, “Database error reporting strategy” on page 222.
- Diagnostic message: This output parameter can be set to a customized error message.

Important: Sometimes you may be tempted to set sqlstate to the value returned to the stored procedure by the SQL runtime in order to pass it on to the calling client. This approach will not work. You may set sqlstate only to the values specified above. Otherwise, the calling program receives sqlstate 39001, which indicates an invalid sqlstate.

8.3.1 Checking the stored procedure completion status

When the stored procedure is called and the control returns to the calling program, the completion status of the stored procedure is stored in the SQLCA area. You can check for the appropriate SQLCODE, SQLSTATE, error message length, and the error message text.

The SQL parameter style allows you more flexibility. SQLSTATE can be set within the external stored procedure along with the message text.
SQL and DB2SQL parameter style

This section shows how error handling is made easier with the SQL parameter style. All errors occurring within the stored procedure can be handled within the stored procedure, depending on the business logic implemented by the stored procedure. When a stored procedure is called by a program, the calling program checks the completion status. As described in 6.3.1, “Coding for SQL parameter style” on page 132, a stored procedure can return an output parameter that contains SQLSTATE, along with the message text that describes the error that occurred.

The PRODPIC table contains pictures of the products that exist in the STOCK table. The PRODPIC table contains a PROD_PICTURE column. This column stores the picture of the product. The picture in this column can be in one of the widely accepted formats, such as GIF, JPG, or BMP. The data type of the column is BLOB. We created a stored procedure that can be used to insert a new product picture into the PRODPIC table. The stored procedure accepts two parameters, the product number, and the IFS file name that contains the picture of the product. Some of the possible errors that can occur during the stored procedure execution are listed here:

- IFS file not found
- Table STOCKPIC not found
- Other errors

We illustrate both error handling and error correction within the stored procedure. A warning message is displayed if a severe error is encountered and corrected. We return an SQL state in the form of '38yxx' if a severe error is encountered, but cannot be corrected. The severe error conditions, along with a diagnostic message text, are returned to the calling program. The possible error conditions and the corresponding SQLSTATE and message text are listed in Table 8-2.

Table 8-2  Errors

<table>
<thead>
<tr>
<th>Error condition</th>
<th>SQLSTATE</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFS file not found</td>
<td>38TNT</td>
<td>IFS file not found</td>
</tr>
<tr>
<td>STOCKPIC table not found</td>
<td>01HTC</td>
<td></td>
</tr>
<tr>
<td>Any other errors</td>
<td>38999</td>
<td>Unhandled error condition within the stored procedure</td>
</tr>
</tbody>
</table>

Examine the CREATE PROCEDURE statement for the INSPIC procedure in Example 8-16. The numbered sections are explained in the list that follows.

Example 8-16  INSERTPIC external stored procedure creation

```
CREATE PROCEDURE SPROCLIB/INSPIC(
    IN prdnbr CHAR(5),
    IN filename CHAR(50))
SPCIFIC INSPIC
LANGUAGE C
EXTERNAL NAME SPROCLIB/insertpic
MODIFIES SQL DATA
PARAMETER STYLE SQL
```

CREATE PROCEDURE notes

The following notes refer to Example 8-16:

1. This is the external stored procedure called INSPIC with two input parameters.
2. We use the SQL parameter style.
Let us examine the source of INSERTPIC referred to in the CREATE PROCEDURE statement shown in Example 8-16. This procedure uses the product number and the file name where the picture is stored as the input parameters. The picture is inserted into the PRODPIC table. The code sample in Example 8-17 illustrates how error handling can be effectively implemented with the SQL parameter style.

Example 8-17 INSERTPIC external stored procedure C source

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;
char dummy[ 5 ];
EXEC SQL
WHENEVER SQLERROR GOTO ErrorHandler;

void main(int argc, char **argv)
{

EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS BLOB_FILE pict_file;
char fname[256];
char prdnum[6];
EXEC SQL END DECLARE SECTION;
unsigned char statevar[5];
unsigned char errmc[70];
struct outmsgtxt{ short int length;
    unsigned char data[70];}
outmsgtxt_var, inmsgtxt_var;
strcpy(prdnum, argv[1]);
strcpy(pict_file.name, argv[2]);
inmsgtxt_var = *(struct outmsgtxt *)argv[8];
pict_file.name_length = strlen(pict_file.name);
pict_file.file_options = SQL_FILE_READ;

EXEC SQL
    INSERT INTO ordentlib/PRODPIC(product_number,product_picture)
    VALUES(:prdnum, :pict_file);
    if ((sqlca.sqlcode ==0))
    {
        strncpy(statevar,"00000",5);
        strncpy(argv[5],statevar,5);
        exit(0);
    }
ErrorHandler:
    if ((sqlca.sqlcode ==-204))
    {
        EXEC SQL
        CREATE TABLE PRODPIC(PRODUCT_NUMBER FOR COLUMN PRDNBR
            CHAR(5) NOT NULL WITH DEFAULT, PRODUCT_PICTURE FOR COLUMN PRDPIC
            BLOB(1 M ) ) ;
        EXEC SQL
        INSERT INTO PRODPIC (product_number,product_picture)
        VALUES(:prdnum, :pict_file);
        strncpy(statevar,"01HTC",5);
        strncpy(argv[5],statevar,5);
        exit(0);
    }
    if ((sqlca.sqlcode ==-452))
    {
        strncpy(statevar,"38FNT",5);
    }
```

The code in Example 8-17 demonstrates how to handle errors in the SQL parameter style. The `WHENEVER SQLERROR GOTO ErrorHandler;` clause is used to redirect control to the `ErrorHandler` label whenever a SQL error occurs. The error messages are prefixed with the string prefixes 00000, 01HTC, and 38FNT, representing different error conditions.
strncpy(argv[5], statevar, 5);
strncpy(errmc, "IFS file not found", 18);
strncpy(outmsgtxt_var.data, errmc, 18);
outmsgtxt_var.length = 18;
*(struct outmsgtxt *) argv[8] = outmsgtxt_var;
exit(1);
}
else
{
    strncpy(statevar, "38999", 5);
    strncpy(argv[5], statevar, 5);
    strncpy(errmc, "unhandled exception", 21);
    strncpy(outmsgtxt_var.data, errmc, 21);
    outmsgtxt_var.length = 21;
    *(struct outmsgtxt *) argv[8] = outmsgtxt_var;
    exit(1);
}

**Code sample notes**
The following notes refer to Example 8-17 on page 244.

1. The SQLSTATE and the message text output parameters are used to return the different errors from the stored procedure to the calling program.

2. If the IFS file does not exist, the insertion will fail. The program logic checks for SQLCODE= -452, which corresponds to SQLSTATE='4281A'. This SQLSTATE cannot be directly returned to the calling program. If you return SQLSTATE='4281A', it will be treated as an invalid SQLSTATE, and the calling program SQLCA.SQLSTATE will contain '39001'. We set the SQLSTATE output parameter to a user-defined value. Since the severity of the error is very high, let us set the SQLSTATE to '38FNT'. The message text will explain the error condition.

3. If the PRODPIC table does not exist in the library, the INSERT statement will fail. The program logic checks for SQLCODE= -204, which corresponds to SQLSTATE='42704'. This error condition is handled within the stored procedure, and error correction measures are taken. If the insertion fails because the table did not exist, the PRODPIC table is created within the stored procedure. Any errors occurring on the execution of the CREATE TABLE statement are returned to the calling program with an SQLSTATE of '38999' and appropriate message text.

4. After the table is created, the values are inserted into the table.

5. On successful execution, the stored procedure returns to the calling program the SQLSTATE set to "00000".

The calling program checks for the different error conditions after the SQL CALL statement using SQLWHENEVER or a conditional structure like IF-THEN_ELSE or CASE statements. The code in Example 8-18 shows the error handling done in the calling program.

*Example 8-18  Client C program calling a stored procedure and recovering error information*

```c
#include<stdio.h>
#include <string.h>
#include <stdlib.h>
#include<ctype.h>
#include <decimal.h>
#include <recio.h>
define SIZE 5
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
```
char productnum[5];
char filename[50];
EXEC SQL END DECLARE SECTION;
void main(void)
{
  int res1,res2;
  unsigned char mstatevar[5];
  EXEC SQL
    WHENEVER SQLERROR GOTO printmsg;
  EXEC SQL
    WHENEVER SQLWARNING GOTO printnomsg;
  printf("Enter the Product number:\n");
  gets(productnum);
  printf("Enter the Product picture filename with path :\n");
  gets(filename);
  EXEC SQL
  call sproclib/inspic(:productnum , :filename);
  strncpy(mstatevar,SQLSTATE,5);
  printf("The SQLSTATE returned from the Stored procedure:%s\n",mstatevar);
  exit(0);
  printmsg:
  printf("The SQLCODE returned:%i\n", sqlca.sqlcode);
  strncpy(mstatevar,SQLSTATE,5);
  res1=strncmp(mstatevar,"37999",5);
  res2=strncmp(mstatevar,"38999",5);
  if((res1 > 0) && (res2 <= 0))
    {
      printf("The user-defined SQLSTATE returned from SP:%s\n",mstatevar);
      printf(" The error message:%s\n", sqlca.sqlerrmc);
    }
  else
    {
      printf("The SQLSTATE returned from SP:%s \n",mstatevar);
    }
  exit(1);
  printnomsg:
  strncpy(mstatevar,SQLSTATE,5);
  printf("The Stored procedure returned a warning:\n");
  printf("The SQLSTATE returned from the Stored procedure:%s\n",mstatevar);
  exit(0);
}

8.3.2 GENERAL and GENERAL WITH NULLS parameter styles

External stored procedures defined with the GENERAL and the GENERAL WITH NULLS parameter styles do not have any specific way of returning error conditions. The GENERAL and GENERAL WITH NULLS parameter styles were provided as a mechanism of reusing an existing procedure, and not necessarily designed as database stored procedures. Most of the procedures report their error conditions through an output parameter, like the example in 6.3.3, “Coding the GENERAL WITH NULLS parameter style” on page 140.
8.4 Error handling in Java stored procedures

The Java language allows a great deal of flexibility in defining and throwing user-defined exceptions. The concept of database error handling in a Java stored procedure is based on the same error handling strategy used for any other kind of errors in Java, using TRY and CATCH blocks.

If a Java exception is thrown within a TRY block and one of the following CATCH statements handles it, the Java stored procedure is not interrupted, and its execution continues after the CATCH block.

If a Java Exception is not caught, it is returned to the caller. The message passed back to the caller depends on the type of exception thrown in the Java method, and essentially on whether it was an SQLException.

You can use the SQLException class to define an exception with virtually any sqlcode and sqlstate. However, upon return from a Java stored procedure, DB2 Universal Database for iSeries runtime handles only certain return codes. For consistency reasons, we recommend that you adopt a similar approach as to that presented in 8.2, “Error handling in SQL stored procedures” on page 223. While throwing the SQLException for a user-defined error condition, you can set the sqlcode to -438 and the sqlstate to a state of class '38yxx'. This way the error condition is properly recognized by the database runtime and passed back to the calling process.

You also can use SQLWarning class, which is an extension of the SQLException class, to manage warnings.

Table 8-3 is a guideline for setting the SQL return codes to values that can be properly handled by client code written in any programming language. It summarizes the possibilities of mapping between a Java exception and its corresponding SQLException returned to the caller.

<table>
<thead>
<tr>
<th>Java exception</th>
<th>SQLCODE of the Java SQLException</th>
<th>Result in a returned SQLException</th>
<th>SQLCODE of the returned SQLException</th>
<th>SQLSTATE of the returned SQLException</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.SQLException</td>
<td>&gt; 0 or = 0</td>
<td>No</td>
<td>No SQLException</td>
<td>No SQLException</td>
</tr>
<tr>
<td>java.sql.SQLException</td>
<td>&lt; 0</td>
<td>Yes</td>
<td>SQLCODE of the Java SQLException</td>
<td>SQLSTATE of the Java SQLException</td>
</tr>
<tr>
<td>Any other Exception</td>
<td>Not relevant</td>
<td>Yes</td>
<td>-443</td>
<td>'38501'</td>
</tr>
</tbody>
</table>

When the SQLCODE of the Java SQLException is 0, the client code should assume that the Java stored procedure executed without any errors or warnings. The only trace that a Java SQLException was thrown is message MCH74A0 logged in the job log of the server job that executed the stored procedure.

If the SQLCODE of the Java SQLException is positive, the message corresponding to the DB2 Universal Database for iSeries SQLCODE is logged in the job log. If the caller is a Java client, a Java SQL warning is returned with the SQLSTATE and the SQLCODE of the Java SQL Exception. This allows you to create your own Java stored procedure with, for example, an SQL code of +438 (stored procedure signaled an error); and an SQLSTATE defined at your convenience, for example, '01ABC'. The getWarnings() method in the Java client allows you to then retrieve your user-defined SQLSTATE and react accordingly.
If the SQLCODE of the SQLException is negative and is a valid DB2 Universal Database for iSeries value, the appropriate message is logged in the job log. If the client is a Java program, an SQLException is returned with SQLCODE and SQLSTATE values as set by the Java stored procedure.

Any unhandled Java exception is returned to the caller as a generic SQLException with SQL code -443 and the user-defined SQL state of 38501.

**Consistent error handling for SQL and Java stored procedures**

As discussed in 8.1, “Database error reporting strategy” on page 222, if you deal with a heterogeneous environment with SQL stored procedures, SQL or DB2SQL parameter style external stored procedures, and Java stored procedures, we strongly recommend that you adopt a consistent approach for error handling.

Let us suppose that you use the following RESIGNAL statement in an SQL stored procedure to return a “file not found” condition to the caller:

```
DECLARE EXIT HANDLER FOR file_not_found
RESIGNAL SQLSTATE '38TNT' SET MESSAGE_TEXT = 'IFS file not found.';
```

The SQLCODE is set by the database runtime to -438 on the return from the SQL stored procedure. Both the JDBC and ODBC drivers monitor this SQL return code and pass the user-defined SQL state and error message to the client application.

The equivalent Java SQLException can be defined as shown here:

```
if (e instanceof FileNotFoundException) throw
    new SQLException("IFS file not found.", "38TNT", -438);
```

Note that for the Java stored procedure, the user-defined error message is overlaid with the system error message associated with the -438 error code. If there is no system message for this error code on your iSeries server, you can define it with the following CL command:

```
ADDMMSGD MSGID(SQL0438) MSGF(QSQLMSG) MSG('Stored procedure signaled an error')
```

**Error handling example**

Section 6.2.1, “SQL parameter style” on page 130, explains that Java stored procedures on the iSeries server can only pass LOB parameters when using the DB2GENERAL parameter style. However, this does not imply that internally the Java stored procedure cannot use Clob or Blob classes, provided the JDBC 2.0 driver is used. It is also possible to handle BLOB without using the Blob class, but rather by means of an InputStream, as shown in the following example.
This illustration features a Java stored procedure, LoadPicture, used for loading files (pictures of the products) into the table PRODPIC. The procedure receives two parameters: Product number and the file path of the picture in the IFS. If the file is not found in the IFS, a Java SQLException is raised and returned to the caller as a Java SQLWarning with a specific user-defined SQLSTATE indicating the special condition. Refer to 8.3, "Error handling in external stored procedures" on page 242, for the C-embedded SQL version of this stored procedure. The code of this stored procedure is presented in Example 8-19.

Example 8-19   Java stored procedure reporting error and warning conditions

```java
import java.sql.*;
import java.io.*;

public class LoadPicture {

    public static void loadPicture (String imageId, String imageFile) throws SQLException, Exception {
        Connection con = DriverManager.getConnection("jdbc:default:connection");
        Statement s = null;
        PreparedStatement ps = null;
        File f = null;
        InputStream is = null;
        boolean tableCreated = false;
        try {
            ps = con.prepareStatement("INSERT INTO PRODPIC VALUES(?, ?)");
        } catch(SQLException ex) {
            if (ex.getErrorCode() == -204)
                // or (e.getSQLState().equals("42704"))
                {  // the table doesn't exist, we create it
                    s = con.createStatement();
                    s.executeUpdate("CREATE TABLE PRODPIC " +
                        
                        "(PRODUCT_NUMBER FOR COLUMN PRDNBR CHAR ( 5) " +
                        "" +
                        "NOT NULL WITH DEFAULT," +
                        "PRODUCT_PICTURE FOR COLUMN PRDPIC BLOB ( 1 M))");
                    tableCreated = true;
                    ps = con.prepareStatement("INSERT INTO PRODPIC VALUES(?, ?)");
                } else {
                    throw ex;
                }
        }
        f = new File(imageFile.trim());
        int filelength = (int) f.length();
        try {
            is = new FileInputStream(f);
        } catch(IOException e) {
            if (e instanceof FileNotFoundException)
                throw new SQLException("IFS file not found.", "38FNT", -438);
        }
        ps.setString(1, imageId);
        ps.setBinaryStream(2, is, filelength);
        ps.executeUpdate();
        if (tableCreated)
            throw new SQLException("Table created and insert successfull.", "01HTC", 438);
    }
}
```
The SQL statement to create the stored procedure is:

```
CREATE PROCEDURE LOADPICTURE
  (in id CHAR(5), in image CHAR(100))
LANGUAGE JAVA PARAMETER STYLE JAVA NOT FENCED
EXTERNAL NAME 'LoadPicture!loadPicture';
```

Now that the stored procedure is registered, we can call it with the JDBC client `ClientLoadPicture`. Its code is shown in 8.5.1, “Retrieving error conditions in a JDBC client” on page 251.

8.5 Retrieving user-defined errors in a client application

The ODBC, Toolbox JDBC, and native JDBC drivers support user-defined error messages. The drivers test the sqlcode field in the SQLCA area, and if it contains either -443 or -438, the SQLSTATE and the diagnostic message are retrieved from the SQLCA. Then the client application can access these values using standard error handling code.

Notes: The following notes refer to Example 8-19:

1. Imports File and FileInputStream classes.
2. Prepares the statement that inserts one record in the file. This statement fails if the table does not exist.
3. An SQLException with a negative SQLCODE of -204 is thrown if the table does not exist. We can catch this exception and test for the SQLCODE to create the table.
4. Rather than checking the SQLCODE that may be different for different databases, we can check the SQLSTATE, which is more portable. The value ‘42074’ indicates that the table is not found and corresponds to the SQLCODE -204 on the iSeries server.
5. Table created containing the pictures, if it was not found.
6. If a non-handled SQLException is thrown, it is thrown to the caller.
7. A FileInputStream is created in the IFS, based on the file path received as a parameter.
8. If this path does not exist or the file is not found, a FileNotFoundException is generated. Because this exception is not a SQLException, it results in an SQLWARNING with SQLCODE -443 being returned to the caller. This is a generic exception that does not provide the caller process with a cause of the failure. To provide the caller with more descriptive error information, we catch the generic exception and throw our own customized SQLException. We choose the negative SQLCODE -438 indicating that a serious error condition occurred in the stored procedure. We also create our own SQLSTATE ‘38FNT’, where FNT is a user-defined portion of the SQLSTATE. Now the caller application can test for the SQLSTATE returned by the stored procedure and get the information that the IFS path passed as a parameter to the stored procedure that was not valid.
9. The FileInputStream class is set up to correspond to the BLOB column in the table.
10. If the target table had been created, the stored procedure signals a warning (SQLException with a positive SQLCODE). We choose the SQLCODE +438, indicating that the stored procedure signalled an error. We create the SQLSTATE ‘01HCT’, where CT stands for create table.

The SQL statement to create the stored procedure is:

```
CREATE PROCEDURE LOADPICTURE
  (in id CHAR(5), in image CHAR(100))
LANGUAGE JAVA PARAMETER STYLE JAVA NOT FENCED
EXTERNAL NAME 'LoadPicture!loadPicture';
```

Now that the stored procedure is registered, we can call it with the JDBC client `ClientLoadPicture`. Its code is shown in 8.5.1, “Retrieving error conditions in a JDBC client” on page 251.
8.5.1 Retrieving error conditions in a JDBC client

A Java client, independent of whether it is a JDBC client or an SQLJ client, uses the TRY and CATCH blocks to manage errors, including database error conditions. There are two predefined classes for managing database errors: java.sql.SQLException and java.sql.SQLWarning. SQLException is an extension of the Exception class and SQLWarning is an extension of SQLException.

SQLException has very useful methods for retrieving related information like getErrorCode() and getSQLState(). The getErrorCode() method retrieves the vendor-specific error code, which in the DB2 Universal Database family is the SQLCODE, while the getSQLState() method retrieves the SQL92 standardized SQLSTATE.

While SQLException is thrown by stored procedures when an error condition is reached, SQLWarnings does not throw an exception. As shown in the example, SQLWarning can be retrieved with the getWarnings() method associated with the SQL statement. The following example shows how to manage both error and warning conditions.

Example 8-20 shows a JDBC client retrieving error conditions.

Example 8-20  Java client program recovering error or warning information

```java
import java.util.*;
import java.io.*;
import java.sql.*;
import com.ibm.as400.access.*;

class ClientLoadPicture {
    public static void main (String argv[]) {
        Properties props = new Properties();
        Connection con = null;
        CallableStatement cs = null;
        try {
            props.load(new BufferedInputStream(new FileInputStream("piclogon.properties")));
            String dbDriver = props.getProperty("dbDriver");
            String dbUrl = props.getProperty("dbUrl");
            String dbUser = props.getProperty("dbUser").trim();
            String dbPassword = props.getProperty("dbPassword").trim();
            String sp = props.getProperty("sp");
            String pictureID = props.getProperty("pictureID");
            String pictureFile = props.getProperty("pictureFile");
            Class.forName(dbDriver).newInstance();
            try {
                con = DriverManager.getConnection(dbUrl, dbUser, dbPassword);
                cs = con.prepareCall("CALL dummy()");
                cs.execute();
            }
```
cs = con.prepareCall("CALL " + sp + "(?,?)");
cs.setString(1, pictureID);
cs.setString(2, pictureFile);

cs.execute();
SQLWarning w = cs.getWarnings();
if (w == null)
    System.out.println("Stored procedure executed without warning.");
else
{
    System.out.println("Stored procedure executed with warning.");
    System.out.println("The warning is : " + w.toString());
    System.out.println("And an ErrorCode : " + w.getErrorCode());
    System.out.println("With a SQLState : " + w.getSQLState());
    System.out.println("Message : " + w.getMessage());
}

if (cs != null) cs.close();
if (con != null) con.close();
System.exit(0);
}
catch (SQLException e)
{
    System.out.println("Here's the e.toString() : " + e.toString());
    System.out.println("Vendor code : " + e.getErrorCode());
    System.out.println("SQLState : " + e.getSQLState());
    System.out.println("Message : " + e.getMessage());
    System.exit(1);
}
}
catch (Exception e)
{
    e.printStackTrace();
    System.exit(1);
}
Chapter 8. Stored procedure error handling

8.5.2 Retrieving error conditions from an ODBC or CLI client

Whenever any ODBC or DB2 CLI function is called, a return code is returned to the application indicating the success or failure of the operation that was attempted. If the operation fails, the application can call the SQLError() function to determine the cause of the error. The SQLError() function returns three values to the application for each error that occurred:

- The native error code
- The SQLSTATE
- The error message text

Based on the values that are returned, the application can then decide on what future course of action it should take. The list of possible codes that can be returned by the function is provided in Table 8-4.

Table 8-4  DB2 CLI function return codes

<table>
<thead>
<tr>
<th>Return codes</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_SUCCESS</td>
<td>The function completed successfully. No additional SQLSTATE information is available.</td>
</tr>
<tr>
<td>SQL_SUCCESS_WITH_INFO</td>
<td>The function completed successfully, but with a warning or other information. The application can call the SQLError() function to determine the SQLSTATE and the native error code. The SQLSTATE code has the class “01”.</td>
</tr>
<tr>
<td>SQL_NO_DATA_FOUND</td>
<td>This is mostly associated with query results. The function completed successfully but cannot find any relevant data.</td>
</tr>
</tbody>
</table>

Notes: The following notes refer to Example 8-20 on page 251:

1. The application reads the information it needs to connect to the DB2 platform and to call the stored procedure from a properties file called “piclogon.properties”. The contents of this file are shown here:

```properties
# piclogon properties
dbDriver=com.ibm.as400.access.AS400JDBCDriver
dbUser=db2admin
dbPassword=db2admin
dbUrl=jdbc:as400://AS400WS
sp=LOADPICTURE
pictureID=00004
pictureFile=/pictures/A004.jpg
```

The dbDriver, dbUrl, dbUser, and dbPassword properties are used to connect JDBC to the database. The property sp indicates the name of the stored procedure. The last two properties, pictureID and pictureFile, are the two input parameters passed to the stored procedure.

2. The stored procedure is executed after establishing its two parameters.

3. The first warning or SQLException with a positive SQLCODE that can be returned by the stored procedure is retrieved.

4. All the information contained in the warning is retrieved and displayed.

5. Any SQLException (with a negative SQLCODE) that could have occurred during the execution of the stored procedure is retrieved.

6. The information contained in this exception is retrieved and displayed.
When the called function returns any code, other than SQL_SUCCESS, the application can call the SQLError function to determine the cause of the error.

Additional SQLCODE and SQLSTATE codes
In addition to the already familiar SQLSTATE classes, the DB2 ODBC and CLI drivers can generate their own error conditions that are of class HY. That is, SQLSTATES like HYxxx correspond to errors that occurred in the DB2 ODBC or CLI driver. The SQLCODE for error generated by DB2 ODBC or CLI is -9999.

Retrieving error codes and messages
Whenever an error condition occurs, the application has to call the SQLError() function to retrieve the error information. The application must pass the following arguments to the function:

- The handle to the environment. If this is not available, you can pass the value SQL_NULL_HENV. This value is defined in the sqlcli.h header file.
- The handle to the connection. If this is not available, the application can pass SQL_NULL_HDBC. This value is defined in the sqlcli.h header file.
- The handle to the statement. If this is not available, the application can pass the value SQL_NULL_HSTMT. This value is available in the sqlcli.h header file.
- The pointer to a character buffer to contain the SQLSTATE.
- The pointer to an integer buffer to contain the native error code.
- The pointer to a character buffer to contain the error message text.
- The size of the buffer that contains the error message text. This should ideally be set to the value SQL_MAX_MESSAGE_LENGTH + 1. This value is defined in the sqlcli.h header file.
- The pointer to a small integer buffer that contains the number of bytes that the function returned after execution. See Example 8-21.

Example 8-21  CLI client program recovering error condition

<table>
<thead>
<tr>
<th>Return codes</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ERROR</td>
<td>The function call failed due to a problem. To determine the cause of the failure, the application can call the SQLError() function. The function retrieves the SQLSTATE, the native error code, and the error message text.</td>
</tr>
<tr>
<td>SQL_INVALID_HANDLE</td>
<td>The function failed. This is because an invalid environment handle, connection handle, or statement handle was sent as an argument to a function.</td>
</tr>
</tbody>
</table>

When the called function returns any code, other than SQL_SUCCESS, the application can call the SQLError function to determine the cause of the error.
PrintError( SQL_NULL_HENV, SQL_NULL_HDBC, SQL_NULL_HSTMT );
exit( -1 );
}

Nmi_ReturnCode = SQLAllocConnect( Hnd_henv, &Hnd_Hdbc );
if ( Nmi_ReturnCode != SQL_SUCCESS ) {
    PrintError( Hnd_Henv, SQL_NULL_HDBC, SQL_NULL_HSTMT );
    exit( -1 );
}

... (Processing Tasks)
...
Nmi_ReturnCode = SQLAllocStmt( Hnd_Hdbc, &Hnd_Hstmt );
if ( Nmi_ReturnCode != SQL_SUCCESS ) {
    PrintError( Hnd_Henv, Hnd_Hdbc, SQL_NULL_HSTMT );
    exit( -1 );
}

Nmi_ReturnCode = SQLExecute( Hnd_Hstmt );
if ( Nmi_ReturnCode != SQL_SUCCESS ) {
    PrintError( Hnd_Henv, Hnd_Hdbc, Hnd_Hstmt );
    exit( -1 );
}

... (Termination Tasks)
...

int PrintError( SQLHENV henv, SQLHDBC hdbc, SQLHSTMT hstmt ) {
    SQLRETURN   returncode;
    SQLCHAR     sqlstate[ SQL_SQLSTATE_SIZE ];
    SQLINTEGER  NativeErrorCode;
    SQLCHAR     MessageText[ SQL_MAX_MESSAGE_LENGTH + 1 ];
    SQLSMALLINT numbytes;
    /* --- Retrieve the SQLSTATE, Native Error Code, Message text --- */
    returncode = SQLError( henv, hdbc, hstmt, sqlstate,
                            &NativeErrorCode, MessageText,
                            sizeof( MessageText ), &numbytes );
    if ( returncode != SQL_SUCCESS ) {
        printf( "Could not retrieve error information\n" );
        return( -1 );
    }
    /* --- Display the Error Information --- */
    printf( "The SQLSTATE is %s\n", sqlstate );
    printf( "The Native Error Code is %d\n", NativeErrorCode );
    printf( "Error Message:\n" );
    printf( "%s\n", MessageText );
    return( returncode );
}

For an ODBC client code example retrieving error conditions that occurred in the stored procedure, see 8.8.4, "C++ client code using ODBC" on page 272.
8.6 Transaction management in stored procedures

When you run stored procedures, you can update several rows in one table or even update a number of rows in different tables. What happens when a failure occurs on the server side (in this case, the iSeries server) and there are rows that were supposed to be updated, but they were not?

Journaling, commitment control and the recently introduced savepoints play a major role in error handling that many of the iSeries customers are sub utilizing. With the original transaction model of the iSeries the programmers had a mechanism that allows them to confirm (commit) or reverse (roll back) database changes. In this way, if the program reaches a point in which a user-defined error condition is detected, all database changes can be voluntarily rolled back to the previous commit point or to the original state. Savepoints enhanced the transaction model by providing more granularity in the transaction control.

As a principle of design, stored procedures, and also triggers and user-defined functions, should keep independent from caller or firing processes. Because of this, we, as designers and programmers, should carefully plan for an error-handling strategy that does not ruin the caller or firing process work.

In this section we discuss the transaction management strategies that are available in stored procedures. Let us start with transaction management terminology.

8.6.1 Transaction management terminology

A transaction is a set of operations to be completed at one time as though they are a single operation. A transaction must be fully completed, or not performed at all. An example of a transaction is the transfer of funds from a savings account to a checking account. To the user, this is a single transaction. However, more than one change occurs to the database because both the savings account and checking account are updated. It is unacceptable for your checking account to be debited while your savings account is not credited.

Commitment control is a function that allows you to define and process a group of changes to resources (such as database files or tables) as a logical unit of work (LUW). A logical unit of work is defined as a group of individual changes to objects on the system that should appear as a single atomic change to the user. End users and application programmers may think of an LUW as a transaction. Commitment control ensures that either the entire group of individual changes occur on all systems participating in the LUW or that none of the changes occur.

Savepoint is a marker or milestone within a transaction to which data and schema changes can be undone. Nested savepoint is a model where a savepoint can be defined within an existing savepoint versus a linear model where savepoints are not nested. Savepoint level is the atomic context for which a rollback or release outside of the level is not allowed by end user application. Savepoints were first introduced in DB2 Universal Database for iSeries in V5R2.

Isolation level is the level of reinforcement of the transactional behavior of the database for a particular activation group, *NONE being a level in which transactional behavior is not reinforced. Other possible values are read uncommitted (UR), cursor stability (CS), repeatable read (RR) and read stability (RS). Exercise care when reading iSeries literature, because some of these terms have different meanings for authors familiar with other platforms. For more information, refer to SQL Reference, SC41-5612.
8.6.2 Transactional behavior

As long as the isolation level is set up in a value different than "NONE, we can see the scenarios shown in Figure 8-7.

![Transactional scenarios](image)

In Figure 8-7, you see the following scenarios:

- The procedure reaches a COMMIT statement. All SAVEPOINTs are released and the changes in the database becomes permanent. It is like a confirmation that the completion of the transaction has been reached. A new transaction or unit of work is initiated.
- The system crashes before reaching a COMMIT or ROLLBACK statement. In this scenario, all operations are reversed until the beginning of the transaction or unit of work, as illustrated in the first part of Figure 8-7.
- The procedure reaches a ROLLBACK statement. All SAVEPOINTs are released and the changes in the database are reversed until the beginning of the transaction or unit of work, as illustrated in the second part of Figure 8-7.
- The procedure reaches a ROLLBACK TO SAVEPOINT statement. The changes to the database are reversed until the specified savepoint as illustrated in the third part of Figure 8-7. In the figure, savepoint B was not released but savepoint C was.

All SQL programs execute as part of an application process. In OS/400, an application process is called a *job*. An application process is made up of one or more activation groups. Each activation group involves the execution of one or more programs. Programs run under a non-default activation group or the default activation group.
Nested savepoints

Nested savepoints are implemented through savepoint levels, which are name spaces for savepoint names. A savepoint level is implicitly created and ended by specific events as shown in Table 8-5.

Table 8-5  Events that initiate and terminate savepoint levels

<table>
<thead>
<tr>
<th>Savepoint level is initiated when</th>
<th>Savepoint level terminates when</th>
</tr>
</thead>
<tbody>
<tr>
<td>A new unit of work is started.</td>
<td>A COMMIT or ROLLBACK is issued.</td>
</tr>
<tr>
<td>A trigger is invoked.</td>
<td>The trigger completes.</td>
</tr>
<tr>
<td>A user-defined function is invoked.</td>
<td>The user-defined function completes.</td>
</tr>
<tr>
<td>A stored procedure is invoked, and the stored procedure was created</td>
<td>The stored procedure returns to the caller.</td>
</tr>
<tr>
<td>with the NEW SAVEPOINT LEVEL clause.</td>
<td></td>
</tr>
<tr>
<td>There is a BEGIN for an ATOMIC compound SQL statement.</td>
<td>There is an END for an ATOMIC compound SQL statement.</td>
</tr>
</tbody>
</table>

When a savepoint level ends, all active savepoints established within the current savepoint level are automatically released. Any open cursors, Data Definition Language (DDL) actions, or data modifications are inherited by the parent savepoint level and are subject to any savepoint-related statements issued within the parent savepoint level.

In scenario D in Figure 8-8, only statements into stored procedure A are rolled back. All statements in main stored procedure are committed. In scenario E, the COMMIT statement in the stored procedure A confirms all statements since the beginning of the unit of work, no matter what the savepoint level in which they were performed is. The external rollback affects all statements that occurred after that commit. In scenario F, the ROLLBACK statement in stored procedure A backs out database operations performed in procedures B and C.
8.6.3 SQL statements for controlling transactions

The DB2 Universal Database for iSeries stored procedures support the following SQL statements for the transaction management:

- COMMIT
- SAVEPOINT
- ROLLBACK and ROLLBACK TO SAVEPOINT
- RELEASE SAVEPOINT
- SET TRANSACTION

**COMMIT**

The COMMIT statement ends a unit of work and commits the database changes that were made by that unit of work.

---

*Figure 8-8 Nested savepoints*

Insert | Update | Insert | Delete | Call A | Update | Delete | COMMIT
---|---|---|---|---|---|---|---

D. Nested savepoint with partial rollback

Insert | Update | Insert | Delete | ROLLBACK

E. Which statements are really rolled back?

Insert | Update | Insert | Delete | Call A | Update | Delete | COMMIT

F. Nested savepoint with partial rollback affecting inner levels

Insert | Update | Call B | Delete | ROLLBACK

Savepoint A

Savepoint B

Savepoint A
SAVEPOINT
The SAVEPOINT statement is used to establish a milestone into the current LUW. A name for the savepoint should be supplied. You can optionally specify if that savepoint name should be unique; in that case, the savepoint cannot be reused along the stored procedure and procedures, triggers, and UDFs called or fired by.

A savepoint that is set with a SAVEPOINT that did not include the UNIQUE keyword can be reused in the same savepoint level in a subsequent SAVEPOINT statement, without having to explicitly release the original savepoint. In this case, the second savepoint replaces the original savepoint with the same name.

ROLLBACK and ROLLBACK TO SAVEPOINT
The ROLLBACK statement is used to back out the database to the beginning of the current transaction or to a specific savepoint. When the ROLLBACK TO SAVEPOINT is used, the database is backed out to the last savepoint. A specific savepoint may be specified as in the following example:

```sql
... 
INSERT INTO MYLIB.TRACE_TABLE VALUES ('FIRST INSERTED ROW');
SAVEPOINT savepoint_A;
INSERT INTO MYLIB.TRACE_TABLE VALUES ('SECOND INSERTED ROW');
SAVEPOINT savepoint_B;
INSERT INTO MYLIB.TRACE_TABLE VALUES ('THIRD INSERTED ROW');
SAVEPOINT savepoint_C;
INSERT INTO MYLIB.TRACE_TABLE VALUES ('FOURTH INSERTED ROW');
...
ROLLBACK TO SAVEPOINT savepoint_B;
...
```

In this case, table TRACE_TABLE will be inserted with the first and second rows, but the third and fourth rows will be rolled back. Savepoint_b will not be released; it will continue to exist after the ROLLBACK TO SAVEPOINT statement is executed.

RELEASE SAVEPOINT
The RELEASE SAVEPOINT statement releases a previously established savepoint name for reuse. After a savepoint name is released, a rollback to that savepoint name is no longer possible.

SET TRANSACTION
The SET TRANSACTION statement sets the isolation level for the current unit of work. Let us start by defining what the isolation level is. The isolation level used during the execution of SQL statements determines the degree to which the activation group is isolated from concurrently executing activation groups. The isolation level is specified as an attribute of an SQL program or SQL package and applies to the activation groups that use the SQL package or SQL program. DB2 Universal Database for iSeries provides the means of specifying the isolation level through the COMMIT parameter on the CRTSQLxxx, STRSQL, and RUNSQLSTM commands.

The SET TRANSACTION statement can be used to override the isolation level within a unit of work. When the unit of work ends, the isolation level returns to the value it had at the beginning of the unit of work. In the SELECT, SELECT INTO, INSERT, UPDATE, DELETE, and DECLARE CURSOR statements, you can specify the isolation level using an isolation clause. The isolation level is in effect only for the execution of the statement containing the isolation clause. An example of the SET TRANSACTION statement is shown here:

```sql
SET TRANSACTION ISOLATION LEVEL UR
```
This statement sets the isolation level to READ UNCOMMITTED, which is the equivalent to *CHG. Refer to SQL Reference, SC41-5612, for more details.

8.6.4 Transaction management in compound statements

Compound statements allow you to group other statements together in an SQL procedure. Every compound statement starts with a BEGIN clause and ends with an END clause. In the BEGIN clause, you can specify the keyword ATOMIC. This indicates that if an error occurs in the compound statement, all SQL statements in the compound statement are rolled back. If NOT ATOMIC is specified, it indicates that an error within the compound statement does not cause the compound statement to be rolled back, and it is the programmer's responsibility to code the recovery code for the procedure.

**Important:** If ATOMIC is specified, the COMMIT and ROLLBACK statements should *not* be specified in the compound statement, and the tables *must* be journaled.

If UNDO is specified in the declaration of a handler in a compound statement, then ATOMIC *must* be specified in the BEGIN clause.

Example 8-22 illustrates the use of commitment control statements within a compound SQL.

**Example 8-22  Commitment control statements within a compound SQL**

```sql
CREATE PROCEDURE CREDITP
(IN i_perinc DECIMAL(3,2),
  INOUT o_numrec DECIMAL(5,0))
LANGUAGE SQL
BEGIN
1 DECLARE proc_cusnbr CHAR(5);
DECLARE proc_cuscrd DECIMAL(11,2);
DECLARE numrec DECIMAL(5,0);
DECLARE at_end INT DEFAULT 0;
DECLARE not_found
  CONDITION FOR '02000';
DECLARE c1 CURSOR FOR
  SELECT cusnbr, cuscrd
  FROM ordapplib.customer;
DECLARE CONTINUE HANDLER FOR not_found
  SET at_end = 1;
SET numrec = 0;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
  ROLLBACK;
2 OPEN c1;
FETCH c1 INTO proc_cusnbr, proc_cuscrd;
WHILE at_end = 0 DO
  SET proc_cuscrd = proc_cuscrd +(proc_cuscrd * i_perinc);
  UPDATE ordapplib.customer
     SET cuscrd = proc_cuscrd
       WHERE CURRENT OF c1;
  SET numrec = numrec + 1;
  FETCH c1 INTO proc_cusnbr, proc_cuscrd;
END WHILE;
3 SET o_numrec = numrec;
CLOSE c1;
COMMIT;
END
```
Example 8-23 presents an equivalent code, but using BEGIN ATOMIC instead. Notice that the error handler causing a ROLLBACK and the commit statement are not present in this example.

Example 8-23 Commitment control statements using BEGIN ATOMIC

CREATE PROCEDURE CREDITP_ATOMIC
  (IN i_perinc DECIMAL(3,2),
   INOUT o_numrec DECIMAL(5,0))
  LANGUAGE SQL
BEGIN ATOMIC
  DECLARE proc_cusnbr CHAR(5);
  DECLARE proc_cuscrd DECIMAL(11,2);
  DECLARE numrec      DECIMAL(5,0);
  DECLARE at_end      INT DEFAULT 0;
  DECLARE not_found
    CONDITION FOR '02000';
  DECLARE c1 CURSOR FOR
    SELECT cusnbr, cuscrd
    FROM  ordapplib.customer;
  DECLARE CONTINUE HANDLER FOR not_found
    SET at_end = 1;
  SET numrec = 0;
  OPEN c1;
  FETCH c1 INTO proc_cusnbr, proc_cuscrd;
  WHILE at_end = 0 DO
    SET proc_cuscrd = proc_cuscrd +(proc_cuscrd * i_perinc);
    UPDATE ordapplib.customer
      SET cuscrd = proc_cuscrd
      WHERE CURRENT OF c1;
    SET numrec = numrec + 1;
    FETCH c1 INTO proc_cusnbr, proc_cuscrd;
  END WHILE;
  SET o_numrec = numrec;
  CLOSE c1;
END

Difference between V5R1 and V5R2 ATOMIC compound statement

Because savepoints were introduced in V5R2, the behavior of atomic compound statements has changed slightly. In V5R1, a BEGIN ATOMIC compound statement caused an implicit commit when the procedure reached the END statement. This mean that a stored procedure containing an atomic compound statement will automatically commit the database changes that the caller program performed before calling it. In the same way, if the atomic compound statement encounters a non-monitored error condition, it will cause an implicit ROLLBACK, backing out also those database changes performed by the caller before calling the stored procedure.
With the new savepoint support, the atomic compound statements approach is to implicitly start a new savepoint name level and set up a savepoint at the beginning. That savepoint is released when the compound statement reaches the END statement. This leaves the commitment responsibility to the caller, while still allowing the compound statement to be atomic. If the stored procedure reaches an unmonitored error condition, it will implicitly execute a ROLLBACK TO SAVEPOINT, backing out only the database changes performed inside the stored procedure.

This small behavioral change increases the usability of atomic compound statements in stored procedures, triggers, and UDFs because the side effect of committing operations outside the scope of the procedure is easily avoided now.

Consider the following stored procedure:

```sql
CREATE PROCEDURE ATOMIC01()
LANGUAGE SQL
BEGIN ATOMIC
    INSERT INTO TEST VALUES ('ATOMIC01', CURRENT TIMESTAMP, 'Row inserted in ATOMIC01');
END;

CREATE PROCEDURE CALLER()
LANGUAGE SQL
BEGIN
    INSERT INTO TEST VALUES ('CALLER', CURRENT TIMESTAMP, 'Row inserted before calling ATOMIC01');
    CALL ATOMIC01;
    INSERT INTO TEST VALUES ('CALLER', CURRENT TIMESTAMP, 'Row inserted after calling ATOMIC01');
    ROLLBACK;
END;
```

After executing stored procedure CALLER in V5R2, the TEST table will be empty, given that it was empty before execution, while in V5R1 it will not.

### 8.7 External stored procedures and commitment control

The following paragraphs describe how you can preserve the integrity of transactions where stored procedures perform data changes, either on the local site or on a remote system.

#### 8.7.1 Activation group

The scope for the transaction or UOW is the activation group. This means that if both the caller and stored procedure programs are running on the same activation group, changes performed by both programs belongs to the same transaction. A commit performed in the stored procedure will also commit changes previously performed by the caller program, something that is usually we want to avoid. This is illustrated in Figure 8-9 on page 264.
While scenario 1 is acceptable, scenarios 2 and 3 are generally considered to be the result of an ill-designed application.
When the caller application and the stored procedure run on different activation groups, they act as separate transactions. The commit or rollback in one of them does not affect the operations performed by the other, as shown in Figure 8-10.

If your stored procedure is an OPM program running at the remote system, it will always run in the default activation group and in the same commitment definition as the calling application (see Figure 8-11 and Figure 8-12).
8.7.2 Savepoints

Savepoints where first introduced in DB2 Universal Database for iSeries in V5R2, extending the functionality of the commitment control. With the savepoints, the level of control on transactions that involves stored procedures is greatly improved, as shown in 8.6.2, “Transactional behavior” on page 257. In addition to what is expressed there, it is important to highlight that the only supported interface to work with savepoints is through SQL statements.

**Note:** If you create an ILE program with default compilation attributes of option 14 on the WRKMBRPDM display, your ILE program may end up running as a separate activation group. Consult the specific ILE language programmer’s guide. Use a two-step compile process, and create the module first with option 15. Then create the program using the CRTPGM ACTGRP(*CALLER) parameter.
A new clause was added to the CREATE PROCEDURE SQL statement, NEW SAVEPOINT LEVEL. By default, when an external stored procedure is created, it is defined to run on the same SAVEPOINT level as the caller. When the clause NEW SAVEPOINT LEVEL is used, a new savepoint level is initiated. In this way, the savepoint names set in the stored procedure will not conflict with existing savepoint names in the caller application.

The COMMIT or ROLLBACK statements are allowed for a local stored procedure. However, if the stored procedure runs in a separate commitment control definition, any COMMIT and ROLLBACK statement issued within the stored procedure affects only the changes performed by the stored procedure itself. The application cannot commit and roll back those changes.

8.8 Some practical examples

Here we present three similar versions of stored procedures:

- An SQL stored procedure
- An external stored procedure
- A Java stored procedure

We also present two versions of client code, a C++ client code and a Java client code.

Because these different versions of stored procedures use a consistent error handling approach, you can see how easily you can change from one to another with transparency from the client point of view.

8.8.1 SQL stored procedure example

Let us start with an SQL stored procedure. The routine is called MODSAL, and it is used to modify an employee’s salary. The personal data for employees, such as serial number, compensation details, and department number, is stored in the EMPLOYEE table. The DEPARTMENT table, in turn, contains the department information including the department’s manager serial number. The records in EMPLOYEE and DEPARTMENT are related by the department number. The MODSAL stored procedure implements a business rule that the total compensation of an employee must not exceed the compensation of their manager. The routine’s logic checks if the rule is not compromised. If it is, an error condition is signaled to the calling process. The code snippet in Example 8-24 illustrates how to set the user-defined errors in an external stored procedure. The routine accepts two parameters: Employee number of type CHAR(5) and salary change of type DECIMAL(9,2). The numbered sections are explained in the list that follows.

Example 8-24  Setting user-defined errors in an external stored procedure

```sql
create procedure db2user.mosal ( in i_empno char(6), in i_salary dec(9,2) )
language SQL
begin atomic

declare  v_job  char(8);
declare  v_salary dec(9,2);
declare  v_bonus dec(9,2);
declare  v_comm dec(9,2);
declare  v_mgrno char(6);
declare  v_mgrcomp dec(9,2);

declare c1 cursor for
select  job, salary, bonus, comm, d.mgrno,
(select (salary+bonus+comm) from employee where empno = d.mgrno) as mgrcomp
```
from employee e, department d
where empno = i_empno and e.workdept = d.deptno;
declare exit handler for sqlstate '38S01'
  resignal sqlstate '38S01'
    set message_text = 'MODSAL: Compensation exceeds the limit.';
declare exit handler for sqlstate '02000'
signal sqlstate '38S02'
    set message_text = 'MODSAL: Invalid employee number.';

open c1;
fetch c1 into v_job, v_salary, v_bonus, v_comm, v_mgrno, v_mgrcomp;
close c1;

if (i_empno <> v_mgrno) and ((v_salary + i_salary + v_bonus + v_comm) >=
  v_mgrcomp)
  then signal sqlstate '38S01';
end if;

update employee set salary = v_salary + i_salary where empno = i_empno;
end

---

**Code sample notes**

The following notes refer to Example 8-24:

1. If the business rule is compromised, sqlstate '38S01' is signaled. The control is transferred to the error handler defined for this state.

2. This error handler defined for the '38S01' sqlstate signals the user-defined error condition. The RESIGNAL statement is used to reset the return sqlstate to '38S01'. It also sets the diagnostic message. After the RESIGNAL is fired, the stored procedure immediately returns the specified error to the caller. Upon return, sqlcode is set to -438. Unlike the external stored procedure, the entire sqlerrmc element of the SQLCA area is available for the customized message. No message truncation occurs in case of SQL stored procedures.

3. The sqlstate '02000' is returned to the SQL SP if there is no data for the employee number passed as the first parameter. This condition may be thrown either by the FETCH or searched UPDATE statement. The error handler handles this condition by signaling sqlstate '38S02' to the caller.

---

**8.8.2 External stored procedure example**

Example 8-25 presents an external stored procedure implemented in C-embedded SQL. The routine is called MODSALC. It is functionally equivalent to the SQL stored procedure MODSAL presented in 8.8.1, “SQL stored procedure example” on page 267.

**Example 8-25  External stored procedure implemented in C-embedded SQL**

```c
#pragma nosigtrunc
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqludf.h>
#include <sqltypes.h>
#include <decimal.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR GOTO ErrorHandler;

void main(int argc, char **argv)
{
```
EXEC SQL BEGIN DECLARE SECTION;
char i_empno[7]; /* input parm - employee number */
decimal(9,2) i_salary; /* input parm - salary change */
char v_job[9]; /* job description */
decimal(9,2) v_salary; /* current salary */
decimal(9,2) v_bonus; /* current bonus */
decimal(9,2) v_comm; /* current commission */
char v_mgrno[7]; /* manager's employee number */
decimal(9,2) v_mgrcomp; /* manager's total compensation */
EXEC SQL END DECLARE SECTION;
unsigned char errmc[SQLUDF_MSGTEXT_LEN + 1];
Strcpy(i_empno,(char*)argv[1]);
i_salary = *( (decimal(9,2) *)argv[2]);
/* retrieve current job and compensation data for an employee 
along with the compensation data for his/her manager */
EXEC SQL
DECLARE C1 CURSOR FOR
select job, salary, bonus, comm, d.mgrno, 
(select (salary+bonus+comm) from employee where empno = d.mgrno)
from employee e, department d
where empno = :i_empno and e.workdept = d.deptno;
EXEC SQL
OPEN C1;
EXEC SQL
fetch C1 into :v_job,:v_salary,:v_bonus,:v_comm,:v_mgrno,:v_mgrcomp;
if (sqlca.sqlcode == 100)
{
    /* signal sqlstate 38S02 */
    strcpy(argv[5],"38S02");
    strcpy(errmc,"Invalid employee number.");
    strcpy(argv[8], errmc);
    exit(1);  
}
EXEC SQL
CLOSE C1;

if((strncmp(i_empno,v_mgrno,6) != 0) &&
    ((v_salary+i_salary+v_bonus+v_comm)>=v_mgrcomp))
{
    /* signal sqlstate 38S01 */
    strcpy(argv[5],"38S01");
    strcpy(errmc,"Compensation exceeds the limit.");
    strcpy(argv[8], errmc);
    exit(1);  
}
EXEC SQL
update employee set salary = :v_salary+:i_salary
where empno = :i_empno;
exit(0);

ErrorHandler:
/* signal sqlstate 38S00 for SQL runtime errors */
strcpy(argv[5],"38S00");
sprintf(errmc,"Native SQL: code=%5d state=%5s\n", 
sqlca.sqlcode, sqlca.sqlstate);
strcpy(argv[8], errmc);
exit(1);  
}
Use the following CL commands to create the stored procedure program object:

```
CRTCMOD MODULE(DB2USER/MODSALC) SRCFILE(DB2USER/QCSRC)
CRTPGM PGM(DB2USER/MODSALC) ACTGRP(*CALLER)
```

After the program object is created, it needs to be registered as a stored procedure with this SQL statement:

```
CREATE PROCEDURE db2user.ModSalC(  IN i_empno char(6),
  IN i_salary dec(9,2))
LANGUAGE C
EXTERNAL NAME db2user.modsalc
MODIFIES SQL DATA
PARAMETER STYLE SQL
```

**Code sample notes**

The following notes refer to Example 8-25:

1. If the business rule is compromised, the stored procedure sets sqlstate to ‘38S01’. It also sets the diagnostic message to indicate the reason of the error condition.

   **Note:** In case of an external stored procedure, the full message text will probably not be accessible to the calling process. Generally, the sqlerrmc field of the SQLCA area is used by the SQL runtime to return the customized error message. However, if the SQL parameter style is specified, the SQL runtime uses a portion of sqlerrmc to return some other information (such as the procedure’s name and schema). The message text itself is placed in the sqlerrmc field as the sixth token. Due to the necessary truncation, the message that is passed back contains no more than 30 characters. You can maximize the returned message length by using short procedure names. This way more bytes in the sqlerrmc element are left for the diagnostic message. The alternate approach is to use the user-defined sqlstate value returned to the client to determine (for example, through a table lookup) what error message should be presented to the end user. In this case, the error message mapping is performed by the client application.

2. This is a “catch-all” error handler for native SQL errors. Keep in mind that the SQL exceptions that are not handled by the external procedure’s logic are not passed back to the calling process. Upon return, the sqlstate is set to ‘00000’, which means successful completion. Therefore, the external procedure needs to take appropriate action. In our case, we set the sqlstate to ‘38S00’, so we make sure that the procedure call fails with -443 sqlcode. The diagnostic message is used to return native sqlcode and sqlstate.

3. The procedure checks to see whether the employee serial number passed as the first parameter is valid. If no data is found for a given employee number, sqlstate is set to ‘38S02’, and an appropriate diagnostic message is returned to the caller.

### 8.8.3 Java stored procedure example

The code sample in Example 8-26 shows the Java implementation of the MODSAL SQL stored procedure.

**Example 8-26   Java implementation of the MODSAL SQL stored procedure**

```
import java.sql.*;
import java.math.*;

public class ModSalJ
{
  public static void modSalJ ( String i_empno, BigDecimal i_salary)
    throws SQLException
  { Connection con = DriverManager.getConnection("jdbc:default:connection");
    ```
String v_job = null;
BigDecimal v_salary = null;
BigDecimal v_bonus = null;
BigDecimal v_comm = null;
String v_mgrno = null;
BigDecimal v_mgrcomp = null;
String stmt = null;
PreparedStatement ps = null;
PreparedStatement psu = null;
try {
    stmt = "select  job, salary, bonus, comm, d.mgrno, ";
    stmt = stmt + 
    "(select (salary+bonus+comm) from employee where empno = d.mgrno) as mgrcomp";
    stmt = stmt + 
    " from employee e, department d";
    stmt = stmt + 
    " where empno = ? and e.workdept = d.deptno";

    ps = con.prepareStatement(stmt);
    ps.setString(1, i_empno);
    boolean cursor;
    ResultSet rs = ps.executeQuery();
    cursor = rs.next();
    if (cursor)
    {
        v_job = rs.getString(1);
        v_salary = rs.getBigDecimal(2);
        v_bonus = rs.getBigDecimal(3);
        v_comm = rs.getBigDecimal(4);
        v_mgrno = rs.getString(5);
        v_mgrcomp = rs.getBigDecimal(6);
        ps.close();
    }
    else
    {
        // signal sql state 38S02
        throw new
        SQLException("Invalid employee number.", "38S02", -438);  

        if (i_empno.compareTo(v_mgrno) != 0
        &&
        (((v_salary.add(i_salary)).add(v_bonus)).add(v_comm)).compareTo(v_mgrcomp)>=0)
        {
            // signal SQL state 38S01
            throw new 
            SQLException("Compensation exceeds the limit.", "38S01", -438);  

            v_salary = v_salary.add(i_salary);
            stmt = "update employee set salary=? where empno = ";
            psu = con.prepareStatement(stmt);
            psu.setBigDecimal(1, v_salary);
            psu.setString(2, i_empno);
            psu.executeUpdate();
            psu.close();
        }
        catch(SQLException ex)
        {
            // we're not handling here
            throw ex;
        }
    }
}
Code sample notes
The following notes refer to Example 8-26 on page 270:

1. The SQLException is thrown to signal that the business rule was compromised. The sqlstate is set to '38S01', and the sqlcode is set to -438. Formally, a Java procedure is an external stored procedure. Therefore, you can also set the sqlcode to -443, that is, to the error code used by the database runtime to indicate error conditions in an external routine. In our methodology, we decided to take advantage of the Java language flexibility to return -438 rather than -443. Since both codes are properly handled by the database runtime, we opted for the solution that provides an entire diagnostic message buffer.

2. If the employee number passed as parameter is invalid, an SQLException is thrown to signal that an error condition occurred in the stored procedure.

3. All potential SQL runtime errors, along with the user-defined error conditions, are caught by this CATCH block and thrown again, so that they are returned to the caller.

8.8.4 C++ client code using ODBC

Example 8-27 shows how the SQL errors returned from a stored procedure can be retrieved and presented in a C++ client that connects to the iSeries database server through an ODBC connection.

Example 8-27   SQL errors presented in a C++ client that connect to the iSeries

```cpp
#include <windows.h>
#include <SQL.h>
#include <sqltext.h>
#include <stdio.h>
#include <iostream.h>
#include <time.h>

// Define The ModifySalary Class
class ModifySalary
{
    // Attributes
    public:
        SQLHANDLE  EnvHandle;
        SQLHANDLE  ConHandle;
        SQLHANDLE  SpStmtHandle;
        SQLRETURN rc;
    // Operations
    public:
        ModifySalary();                        // Constructor
        ~ModifySalary();                       // Destructor
        SQLRETURN executeSP(char *, SQLDOUBLE);
        SQLRETURN printError( SQLHDBC, SQLHSTMT);
};

// Define The Class Constructor
ModifySalary::ModifySalary()
{
    // Initialize The Return Code Variable
    rc = SQL_SUCCESS;
    // Allocate An Environment Handle
    rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &EnvHandle);
    // Set The ODBC Application Version To 3.x
    if (rc == SQL_SUCCESS)
        rc = SQLSetEnvAttr(EnvHandle, SQL_ATTR_ODBC_VERSION,
                            (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    // Allocate A Connection Handle
```
if (rc == SQL_SUCCESS)
    rc = SQLAllocHandle(SQL_HANDLE_DBC, EnvHandle, &ConHandle);
}

// Define The Class Destructor
ModifySalary::~ModifySalary()
{
    // Free The SQL Statement Handle
    if (SpStmtHandle != NULL)
        SQLFreeHandle(SQL_HANDLE_STMT, SpStmtHandle);
    // Free The Connection Handle
    if (ConHandle != NULL)
        SQLFreeHandle(SQL_HANDLE_DBC, ConHandle);
    // Free The Environment Handle
    if (EnvHandle != NULL)
        SQLFreeHandle(SQL_HANDLE_ENV, EnvHandle);
}

SQLRETURN ModifySalary::executeSP(char * i_empno, SQLDOUBLE salary)
{
    // Declare The Local Memory Variables
    SQLRETURN    rc;
    char         empno[7];
    SQLINTEGER   cbValue;
    strcpy((char *)empno, i_empno);
    // Bind the output parameter for the stored procedure
    cbValue = SQL_NTS;
    rc=SQLBindParameter(SpStmtHandle, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                        6, 0, (SQLCHAR *) empno, sizeof(empno), (SQLINTEGER *) &cbValue);
    rc=SQLBindParameter(SpStmtHandle, 2, SQL_PARAM_INPUT, SQL_C_DOUBLE,
                        SQL_DECIMAL, 9, 2, &salary, sizeof(salary), NULL);
    // calling stored procedure
    rc = SQLExecute(SpStmtHandle);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        printError(ConHandle, SpStmtHandle);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        rc = SQLEndTran(SQL_HANDLE_DBC, ConHandle, SQL_ROLLBACK);
    else
        {rc = SQLEndTran(SQL_HANDLE_DBC, ConHandle, SQL_COMMIT);
         cout << "Stored procedure call completed successfully." << endl;}
    return(rc);
}

SQLRETURN ModifySalary::printError (SQLHDBC hdbc, SQLHSTMT hstmt)
{
    SQLCHAR buffer[SQL_MAX_MESSAGE_LENGTH + 1];
    SQLCHAR sqlstate[SQL_SQLSTATE_SIZE + 1];
    SQLINTEGER sqlcode;
    SQLSMALLINT length;
    SQLRETURN rc;

    while ((rc = SQLError(SQL_NULL_HENV, hdbc, hstmt,
    sqlstate,&sqlcode,buffer, SQL_MAX_MESSAGE_LENGTH + 1,
    &length)) == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)
    {
        cout << "SQLSTATE: " << sqlstate << endl;
        cout << "SQLCODE : " << sqlcode << endl;
    }
}
```cpp
cout << "Error msg : " << buffer << endl;
cout <<"----------------------------- " << endl << endl;
}
return(SQL_ERROR);
}

/*-----------------------------------------------------------------*/
/* The Main Function                                               */
/*-----------------------------------------------------------------*/
int main()
{
  // Declare The Local Memory Variables
  SQLRETURN rc = SQL_SUCCESS;
  SQLCHAR ConnectStr[128] = "DSN=PWDROCH;UID=db2user;PWD=db2pwd;";
  SQLCHAR SQLStmt[255];
  char i_empno[7];
  SQLDOUBLE i_salary;

  // Create An Instance Of The ModifySalary Class
  ModifySalary modifySalary;

  // Connect to the sample database
  if (modifySalary.ConHandle != NULL)
  {
    rc = SQLDriverConnect(modifySalary.ConHandle, NULL, ConnectStr, SQL_NTS,
                           NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    {modifySalary.printError(modifySalary.ConHandle,
                             ModifySalary.SpStmtHandle);
      return(rc);
    }
    cout << "Successfully connected ..." << endl;
  }
  cout << "Enter employee number :" << endl;
  cin >> i_empno;
  cout << "Enter salary delta :" <<endl;
  cin >> i_salary;
  // set autocommit off
  rc = SQLSetConnectAttr(modifySalary.ConHandle, SQL_ATTR_AUTOCOMMIT,
                         (SQLPOINTER) SQL_AUTOCOMMIT_OFF, SQL_IS_UINTEGER);

  // Allocate An SQL Statement Handle
  rc = SQLAllocHandle(SQL_HANDLE_STMT, modifySalary.ConHandle,
                      &modifySalary.SpStmtHandle);
  // Now prepare the procedure call statement
  strcpy((char *) SQLStmt, "CALL db2user.modsalC(?,?)");
  strcpy((char *) SQLStmt, "CALL db2user.modsal(?,?)");
  strcpy((char *) SQLStmt, "CALL db2user.modsalJ(?,?)");
  rc = SQLPrepare(modifySalary.SpStmtHandle, SQLStmt, SQL_NTS);
  if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
  {modifySalary.printError(modifySalary.ConHandle,
                           modifySalary.SpStmtHandle);
    return(rc);
  }
  rc = modifySalary.executeSP((char *)i_empno, i_salary);
  if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
  {modifySalary.printError(modifySalary.ConHandle,
                           modifySalary.SpStmtHandle);}  
}

// Return To The Operating System
```
Code sample notes
The following code applies to Example 8-27 on page 272:

1. If there is an error condition, the printError method is invoked to manage that error condition.

2. While there are SQL errors, retrieve them and print its most relevant information.

8.8.5 Java example client code

Example 8-28 shows the Java version of the client code. It illustrates how to handle errors returned from the stored procedure call.

Example 8-28   Handing errors returned from a stored procedure call

```java
import java.math.*;
import java.util.*;
import java.io.*;
import java.sql.*;
import com.ibm.as400.access.*;

class ModifySalary {
    public static void main (String argv[])
    {
        Properties props = new Properties();
        Connection con = null;
        CallableStatement ps;
        String SQL;

        try {
            props.load(new BufferedInputStream(new FileInputStream("modifysalary.properties")));
            String dbDriver = props.getProperty("dbDriver");
            String dbUrl = props.getProperty("dbUrl");
            String dbUser = props.getProperty("dbUser").trim();
            String dbPassword = props.getProperty("dbPassword").trim();
            String empno = props.getProperty("empno");
            System.out.println("Employee Number: " + empno);
            String salaryDeltaS = props.getProperty("salaryDelta");
            BigDecimal salaryDelta = new BigDecimal(salaryDeltaS);
            System.out.println("Salary Delta: " + salaryDelta.toString());
            System.out.println("Decimal Scale: " + salaryDelta.scale());
            String storedProc = props.getProperty("storedProc");
            System.out.println("Stored procedure : " + storedProc);
            Class.forName(dbDriver);
            con = DriverManager.getConnection(dbUrl, dbUser, dbPassword);
            System.out.println("got connection");

            try {
                SQL = "Call " + storedProc + " (?, ?)";
                ps = con.prepareCall(SQL);
                ps.setString(1, empno);
                ps.setBigDecimal(2, salaryDelta);
                ps.execute();
                System.out.println("Stored procedure call completed successfully.");
                if (ps != null) ps.close();
                if (con != null) con.close();
            }
        }
    }
```
catch (SQLException ex)
{
    while (ex != null) {
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("Message: " + ex.getMessage());
        System.out.println("Vendor code: " + ex.getErrorCode());
        ex = ex.getNextException();
        System.out.println("");
    }
}
catch (Exception e) {
    e.printStackTrace();
}
}

The ModifySalary.java attributes are managed by the following properties file:

#logon properties
dbDriver=com.ibm.as400.access.AS400JDBCDriver
#dbDriver=com.ibm.db2.jdbc.app.DB2Driver
dbUser=db2user
dbPassword=db2pwd
dbUrl=jdbc:as400://pwdroch
#dbUrl=jdbc:db2://*LOCAL
empno=000220
salaryDelta=3000.00

#storedProc=DB2USER.MODSALC
storedProc=DB2USER.MODSAL
#storedProc=DB2USER.MODSALJ

8.8.6 Results for the example programs

As mentioned earlier, the user-defined errors returned by a stored procedure are pretty consistent across all supported stored procedure types. The test results displayed by the sample client for different implementations of the MODSAL stored procedure are presented below. In the test scenario, we try to set the salary for the employee '000220' so that the total compensation violates the company regulation. This results in an SQL error.

Results for the MODALC (C-embedded SQL)

Enter employee number:
000220
Enter salary delta:
3000.00
SQLSTATE: 38S01
SQLCODE: -443
Error msg: [IBM][Client Access Express ODBC Driver (32-bit)]Compensation exceeds the lim

Note that, in this case, the original message was truncated. The SQL return code is set to -443.
Results for MODSAL (SQL PSM)
SQLSTATE: 38S01
SQLCODE : -438
Error msg : [IBM][Client Access Express ODBC Driver (32-bit)]MODSAL: Compensation exceeds the limit.

The entire diagnostic message was returned. The SQL return code is set to -438.

Results for MODSALJ (Java)
SQLSTATE: 38S01
SQLCODE : -438
Error msg : [IBM][Client Access Express ODBC Driver (32-bit)]MODSALJ: Compensation exceeds the limit.

The entire diagnostic message was returned. The SQL return code is set to -438.
This part describes the trigger support available in DB2 Universal Database for iSeries. It includes a separate chapter for SQL triggers and another one for external triggers.
Database triggers

Triggers represent one of the most powerful features of DB2 Universal Database for iSeries. This chapter introduces the concept of triggers, when they can be used, and the types of triggers supported by DB2 Universal Database for iSeries.

This chapter discusses:

- Trigger concepts
- Types of triggers
9.1 Trigger concepts

Triggers are *application independent*. They are user-written programs that are activated by the database manager when a data change is performed in the database. Triggers are mainly intended for monitoring database changes and taking appropriate actions. The main advantage of using triggers, instead of calling the program from within an application, is that triggers are activated automatically, regardless of the interface that generated the data change.

In addition, once a trigger is in place, application programmers and end users cannot circumvent it. When a trigger is activated, the control shifts from the application program to the database manager. The operating system executes your coded trigger program to perform the actions you designed. The application waits until the trigger ends and then gains control again.

Triggers may cause other triggers to be called. In the example shown in Figure 9-1, when you update TABLE1, a trigger is activated and updates database table TABLE2. This operation causes a second trigger to run.

![Figure 9-1 Trigger overview](image)

It is important to identify which database tables have to be monitored and which events should call the triggers, keeping in mind that a trigger is called every time the event happens. We recommend that you think of triggers as part of your database design, rather than as a function related to a specific application.

You may use a trigger program for the following purposes:

- Enforcing business rules, regardless of how complex they are

  You may want to ensure, for example, that whenever you enter an order in your database, the customer you are dealing with has no bad credit history. A trigger associated with the order table can perform this checking consistently and take the appropriate actions.
Data validation and audit trail
You may need to ensure that, whenever a sales representative enters an order, that sales representative is actually assigned to that particular customer. You want to also keep track of the violation attempts. Again, a trigger can be activated on the order table to perform the validation and track the violators in a separate table.

Integrating existing applications and advanced technologies
Your company sends a confirmation fax to the customers after they have accepted an order from you. Triggers can be the ideal solution to integrate your existing Order Entry application with your new facsimile support on the iSeries server. Another example is to send an e-mail confirming the order from a customer.

Preserving data consistency across different database tables
In this case, triggers can complement referential integrity and check constraint support, since they can provide a much wider and more powerful range of data validation and business actions to be performed when data changes in your database.

Triggers represent a powerful technique to ensure that your database always complies with your business needs, provides consistent checking, and takes the appropriate actions every time data is changed.

You can benefit from triggers for several reasons

- **Application independence:**
  DB2 Universal Database for iSeries activates the trigger program, regardless of the interface you are using to access the data. Rules implemented by triggers are enforced consistently by the system rather than by a single application.

- **Easy maintenance**
  If you must change the business rules in your database environment, you need to update or rewrite the triggers. No change is needed to the applications (they transparently comply with the new rules).

- **Code reusability**
  Functions implemented at the database level are automatically available to all applications using that database. You do not need to replicate those functions throughout the different applications.

- **Easier client/server application development**
  Client/server applications take advantage of triggers. In a client/server environment, triggers may provide a way to split the application logic between the client and the server system.

  In Figure 9-2, you can see how client applications can take advantage of the functions performed by the triggers at the server side. In addition, client applications do not need specific code to activate the logic at the server side. Application performance may also benefit from this implementation by reducing data traffic across communication lines.
9.2 Types of triggers in DB2 Universal Database for iSeries

There are two types of triggers available in DB2 Universal Database for iSeries for database tables. Up to 300 triggers can be defined for a single table. Prior to V5R1, only up to six triggers could be defined for a single table. The two types of triggers are:

- SQL triggers
- External triggers

9.2.1 SQL triggers

For an SQL trigger, the program performing the tests and actions is written using SQL statements. The SQL CREATE TRIGGER statement provides a way for the database management system to actively control, monitor, and manage a group of tables whenever an insert, update, or delete operation is performed. The statements specified in the SQL trigger are executed each time an insert, update, or delete operation is performed. An SQL trigger may call stored procedures or user-defined functions to perform additional processing when the trigger is executed.

Unlike stored procedures, an SQL trigger cannot be directly called from an application. Instead, an SQL trigger is invoked by the database management system upon the execution of a triggering insert, update, or delete operation. The definition of the SQL trigger is stored in the database management system and is invoked by the database management system, when the SQL table, that the trigger is defined on, is modified.
9.2.2 External triggers

For an external trigger, the program containing the set of trigger actions can be defined in any supported high-level language that creates a *PGM object. The trigger program can have SQL embedded in it. To define an external trigger, you must create a trigger program and add it to a table using the Add Physical File Trigger (ADDPFTRG) CL command. Or you can add it using iSeries Navigator. To add a trigger to a table, you must:

1. Identify the table.
2. Identify the kind of operation.
3. Identify the program that performs the desired actions.
4. Provide a unique name for the trigger or let the system generate a unique name.

9.3 Enabling and disabling a trigger

Triggers need to be enabled in order to run. However, disabling a trigger allows you to work with the table without causing the trigger to run. This can be useful in cases where a long batch processing has to be done or a large data load is going to take place. In such cases, it can be beneficial to disable the triggers associated with a table.

To enable or disable a trigger, follow these steps:

1. In iSeries Navigator, expand your server -> Database -> Libraries.
2. Right-click the table that contains the trigger that you want to enable or disable. Click Properties, as shown in Figure 9-3.

Figure 9-3 Properties table
3. In the Table Properties window, click the **Triggers** tab. Select the trigger that you want to enable or disable, and click **Enable** to enable it or **Disable** to disable the trigger, as shown in Figure 9-4.

![Figure 9-4 Disabling or enabling a trigger](image)

### 9.4 Displaying and reviewing trigger information

To assist you in visualizing trigger information, you can use:

- iSeries Navigator to view the properties of a trigger
- The Display File Description (DSPFD) command
- The Print Trigger Programs (PRTTRGPDM) command

#### 9.4.1 Using iSeries Navigator to view the properties of a trigger

After you create and define triggers to a table, one of the ways to visualize the properties of a trigger is to use iSeries Navigator:

1. In iSeries Navigator, double-click the **SAMPLEDB01** library. The right panel displays all DB2 Universal Database iSeries objects in this library.

2. Find and right-click the **SALES** table, and select **Properties** as shown in Figure 9-3.
3. In the Table Properties window (Figure 9-5), click the Triggers tab. All the triggers that the table has defined are displayed in the window.

![Table Properties window](image)

Figure 9-5  Triggers tab

Now you can select a particular trigger to see the details of it.

### 9.4.2 Displaying trigger information

The Display File Description (DSPFD TYPE(*TRG)) command provides a list of the triggers that are associated with a file. The command provides the following information:

- The number of trigger programs
- The trigger name and library
- The trigger status
- The trigger program names and libraries
- The trigger events
- The trigger times
- The trigger update conditions
- The trigger type
- The trigger mode
- The trigger orientation
- The trigger creation date and time
The number of trigger update columns
List of trigger update columns

9.4.3 Printing trigger information

Use the Print Trigger Programs (PRTTRGPGM) command to print a report listing of all the trigger programs in a specific library, job library list, user library list, all user libraries on the system, or all libraries on the system. Figure 9-6 shows how the command is used for library LIB03.

Figure 9-6  Print Trigger Programs

Figure 9-7 shows the report produced by the PRTTRGPGM command for all trigger programs found in library LIB03. Notice that all triggers belong to the SALES table and each trigger has a unique name.

Figure 9-7  Trigger Programs Report

You can use this initial report, using CHGRPTONLY(*NO), as a base to evaluate any trigger programs that already exist on your system. Then you can print the changed report, using CHGRPTONLY(*YES), regularly to see whether new trigger programs have been added to your system.
9.5 System catalog tables

Four catalog views related to triggers are defined in QSYS2.

**SYSTRIGGERS**

The SYSTRIGGERS view contains one row for each trigger in an SQL schema, as shown in Figure 9-8.

```
select trigger_name as name,
       event_manipulation as event,
       action_condition as condition,
       action_timing as time,
       trigger_mode as mode,
       trigger_program_name as program
from s Qsys2.sys_triggers
```

---

Figure 9-8 SYSTRIGGERS sample data
SYSTRIGCOL
The SYSTRIGCOL view contains one row for each column, either implicitly or explicitly referenced in the WHEN clause or the triggered SQL statements of a trigger, as shown in Figure 9-9.

```
select * from diena.systrigcol;
```

![Figure 9-9 SYSTRIGCOL sample data](image)
SYSTRIGDEP

The SYSTRIGDEP view contains one row for each object referenced in the WHEN clause or the triggered SQL statements of a trigger, as shown in Figure 9-10.

Figure 9-10 SYSTRIGDEP sample data
SYSTRIGUPD
The SYSTRIGUPD view contains one row for each column identified in the UPDATE column list, if any, as shown in Figure 9-11.

You can refer to Appendix G in SQL Reference, SC41-5612, for further details about the information contained in these views.

9.6 Authorization and adopted authorities on triggers

The authorization and adopted authorities, as discussed in 3.9, “Adopted authorities in SQL PSM” on page 36, also apply for triggers programs. However, SQL triggers will always be generated with USRPRF and DYNUSRPRF set to *OWNER and USEADPAUT set to *YES, independently of the specified values in the RUNSQLSTM command or the SET OPTION statement.

For deployment of triggers, you must consider that most of them should be able to run for all users. This, in fact, means that it is advisable to generate trigger programs with authorization set to *OWNER and using an owner profile with authorities over the resources affected by them.
9.7 Renaming and copying

Any table (including the subject table) referenced in a triggered-action can be moved or renamed. However, the triggered-action continues to reference the old name or library. An error occurs if the referenced table is not found when the triggered-action is executed. Therefore, you should drop the trigger and then re-create the trigger so that it refers to the renamed table.

If records are copied to a physical file that has an *INSERT trigger program associated with it, the trigger program is called each time a record is copied to the file. The trigger program is not called if deleted records are copied. If an error occurs while the trigger program is running, the copy operation fails. However, records that were successfully copied before the error occurred remain in the to-file.

We recommend that you use the following steps when copying records to a table that has an *INSERT trigger program associated with it:

1. Use the CHGPFTRG command or iSeries Navigator to disable all triggers associated with the target table.
2. Use the CPYF command to copy the records to the target table.
3. Use the CHGPFTRG command or iSeries Navigator to enable all triggers associated with the target table.

Note: The CHGPFTRG command will fail if the table is open.
Chapter 10. SQL triggers

This chapter describes the new support introduced in V5R1 for SQL triggers. This new support allows you to write triggers using extensions to the SQL Language as defined by the SQL standard. The greatest advantage of using SQL triggers is portability. You can often use the same SQL trigger across other relational database management systems (RDBMSs).

This chapter includes:

- Introduction to SQL triggers
- Definition of an SQL trigger
- Components and planning of an SQL trigger
- Trigger events
- Trigger modes
- Trigger times
- Correlation variables
- Transition tables
- Changing and correcting data
- Error handling
- Testing and debugging

Note: In V5R3 of iSeries Navigator, major enhancements were made to the Database interface. In this chapter, we illustrate the iSeries Navigator interface used in V5R2. For details about the enhancements done to the interface, refer to the IBM Redbook DB2 Universal Database for iSeries Administration: The Graphical Way on V5R3, SG24-6092.
10.1 Introduction to SQL triggers

Implementation of SQL triggers is based on the SQL standard. It supports constructs that are common to most programming languages. It supports the declaration of local variables, statements to control the flow of the procedure, assignment of expression results to variables, and error handling.

The SQL CREATE TRIGGER statement is similar to the support in other database products like Oracle. Therefore, it provides the portable trigger approach that we mentioned before.

The SQL CREATE TRIGGER statement provides a way for the database management system to actively control, monitor, and manage a group of tables whenever an insert, update, or delete operation is performed. The statements specified in the SQL trigger are executed each time an insert, update, or delete operation is performed. An SQL trigger may call stored procedures or user-defined functions to perform additional processing when the trigger is executed. Instead, an SQL trigger is invoked by the database management system upon the execution of a triggering insert, update, or delete operation. The definition of the SQL trigger is stored in the database management system and is invoked by the database management system, when the table that the trigger is defined on is modified.

When a trigger is created, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program object is then created using the CRTSQLCI and CRTPGM commands. The SQL options used to create the program are the options that are in effect at the time the CREATE TRIGGER statement is executed. The program is created with activation group ACTGRP(*CALLER).

At V5R1, an Internal C Compiler is shipped with the system. This provides an internal API-style interface to the C Compiler. The API interface is shipped with the system in an SQCC component. Each supported parameter of the API interface has the same function as the V4R5 Native C/400® compiler. The internal compiler allows customers to create SQL triggers without having to purchase the C Compiler, even if the user does not have the ILE C product installed.

10.2 System requirements and planning

Before you start to develop SQL triggers on the iSeries server, make sure you are running V5R1 or a higher release of OS/400.

The following program products need to be installed on the iSeries server if you are in V5R1 to be able to create an SQL trigger:

- The DB2 Universal Database Query Manager and SQL Development Kit (5722-ST1)
- Option 13 - System Openness Includes of OS/400 (5722-SS1)

Option 13 - System Openness Includes of OS/400 (5722-SS1) must be installed on the iSeries server if you are in V5R2 to be able to create an SQL trigger. The execution of an SQL trigger requires no extra products to be installed.
10.3 Structure of an SQL trigger

An SQL trigger can be created by either specifying the CREATE TRIGGER SQL statement or by using iSeries Navigator. The SQL-routine-body is the executable part of the trigger that is transformed by the database manager into a program. When an SQL routine or trigger is created, SQL creates a temporary source file (QTEMP/QSQLSRC) that will contain C source code with embedded SQL statements. An SQL trigger is created as a program (*PGM) object using the Create SQL CI (CRTSQLCI) and Create Program (CRTPGM) commands. The program is created in the library that is the implicit or explicit qualifier of the trigger name.

The SQL routine body is a single SQL statement, including an SQL control statement. When the program is created, the SQL statements other than control statements become embedded SQL statements in the program.

```
CREATE TRIGGER name-of-trigger
  ON TABLE_NAME
  FOR EACH ROW / FOR EACH STATEMENT
  MODE DB2ROW / DB2SQL
  WHEN (Condition)
  BEGIN
    Routine body of the trigger
  END
```

Notes: The following notes refer to the previous example.

1. Every trigger starts with CREATE TRIGGER and its name. The name must not match the name of an existing trigger. The name used should not begin with “SQL”.
2. TABLE_NAME specifies the name of table for which this trigger is being defined.
3. FOR EACH specifies whether the database manager will execute the trigger action for each row operation or once for each statement (even if no rows in the table are updated by the application).
4. MODE DB2ROW specifies that the trigger is fired after each row operation. MODE DB2SQL fires the trigger after all row operations.
5. Specifies a condition that evaluates to true or false. It is optional.
6. The body of the trigger starts with the clause BEGIN.
7. The routine body of the trigger defines the actions that the trigger will execute.
8. The body of the trigger ends with the END clause.

The routine body of the trigger program can consist of a single SQL statement (SELECT, UPDATE, INSERT, DELETE) or an SQL control statement.

SQL control statements may include assignment statements, declaration of local variables, and interactive statements. The SQL control statements are the basic programming constructs found in most procedural languages.

The trigger body consists of one or more SQL statements. In general, this can be any SQL statement. One restriction is that a BEFORE trigger cannot execute data modification statements such as an Update or Create. Other than that restriction, most of the same statements used in SQL procedures, and user-defined functions, are available for SQL triggers.
As mentioned earlier, when the Create Trigger statement is executed, a C program object is generated by DB2 Universal Database for iSeries to implement the trigger definition. As this program is created, DB2 Universal Database for iSeries must verify that all tables, views, aliases, user-defined types, user-defined functions, and procedures referenced in the trigger must exist as specified by the standards. The table or view that an alias refers to must also exist when the trigger is created. As the C program object is generated, the trigger body is modified in the following ways to meet the standard requirements for managing trigger dependencies when dropping a table:

- Naming mode is switched to SQL naming.
- All unqualified object references are explicitly qualified.
- All implicit column lists (for example, SELECT *, INSERT with no column list, UPDATE SET ROW) are expanded to be the list of actual column names. This is quite different from external triggers that are allowed to use unqualified object references during execution of the trigger.

### 10.3.1 Components of the SQL trigger definition

Several criteria are defined when creating an SQL trigger:

- **Subject table**: Defines the table for which the trigger is defined.
- **Trigger event**: Defines a specific SQL operation that modifies the subject table. The operation can be to DELETE, INSERT, or UPDATE a row.
- **Activation time**: Defines whether the trigger should be activated before or after the trigger event is performed on the subject table.

Therefore, a table can be associated with six types of SQL triggers:

- Before Delete trigger
- Before Insert trigger
- Before Update trigger
- After Delete trigger
- After Insert trigger
- After Update trigger

The Update SQL trigger allows you to define a trigger at a column level for the subject table.

- **Trigger granularity**: Defines whether the actions of the trigger will be performed once FOR EACH STATEMENT or once FOR EACH ROW in the set of affected rows.
  - If the application operates on a table, but does not update any rows, row-level triggers will not run. However, the statement-level trigger will run.
  - Statement-level triggers are not supported for the BEFORE time.

- **Correlation variables**: The triggered action may refer to the values in the set of affected rows; this is similar to the concept of the trigger buffer used in the system (external) triggers. In SQL triggers, this is supported through a transition variable or correlation variables. Correlation variables use the names of the columns in the subject table qualified by a specified name (for example, OLD. or NEW.) that identifies whether the reference is to the old value (BEFORE the UPDATE) or the new value (AFTER the UPDATE). The new value can also be changed using the SET correlation variable SQL statement in BEFORE UPDATE or INSERT triggers.

- **Transition tables**: This is a temporary table that contains all of the affected rows before and after trigger execution. An SQL trigger may need to refer to all of the affected rows for an SQL insert, delete, or update operation. This is true, for example, if the trigger needs to apply aggregate functions, such as MIN or MAX, to a specific column of the affected rows.
The OLD TABLE or OLD_TABLE and NEW TABLE or NEW_TABLE transition tables can be used for this purpose.

- **Trigger condition (optional):** This uses the WHEN condition (similar to WHERE clause) to control the execution of the trigger body. If the condition equates to true, then the SQL statements in the SQL trigger routine body are executed. If the condition equates to false, the SQL statements in the SQL trigger routine body are not executed, and control is returned to the database management system.

- **Trigger mode:** There are two trigger modes:
  - MODE DB2ROW
    - Triggers are activated on each row operation.
    - Valid for both BEFORE and AFTER activation time.
    - Applies to row-level triggers only.
  - MODE DB2SQL
    - Triggers are activated after all row operations.
      - Row-level: Trigger called n times after nth row processed
      - Statement-level: Trigger called once after the nth row is processed
    - Only allowed on AFTER triggers (mode is automatically changed to DB2ROW on BEFORE triggers).
    - Not as efficient as DB2ROW since each row is processed twice.

- **Trigger body:** In the body of the trigger that starts with the BEGIN clause and ends with the END clause, you can code the following SQL statements:
  - SQL procedure statement
  - SQL control statement
  - Assignment statement
  - CALL statement
  - CASE statement
  - Compound statement
  - FOR statement
  - GET diagnostics statement
  - GOTO statement
  - IF statement
  - ITERATE statement
  - LEAVE statement
  - LOOP statement
  - REPEAT statement
  - RESIGNAL statement
  - RETURN statement
  - SIGNAL statement
  - WHILE statement

For details and examples of the above statements, refer to Chapter 5, “SQL stored procedures” on page 81.
10.3.2 Simple SQL trigger example

Example 10-1 shows a simple example of an SQL trigger. The trigger is named NEW_HIRE, so that it increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table, it increases the value of column NBEMP in table COMPANY_STATS by one.

Example 10-1   Simple SQL trigger

CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;

Notes: The following notes refer to Example 10-1:

1. This defines the name of the trigger. In this case, the name is NEW_HIRE.
2. This defines the trigger event. In this case, the event is AFTER and the time is AFTER.
3. This defines how database manager executes the trigger action.
4. This is the SQL statement that will be executed. The trigger body should be composed by either a simple or compound statement.

10.3.3 Example of a trigger program using WHEN condition

Example 10-2 shows a more complicated example in which the WHEN condition is used. It also illustrates the use of correlation variables, which are discussed in 10.7.1, “Correlation variables” on page 329. The trigger will insert a record into the SALARY_CTL table whenever the salary of an employee is decreased.

Important: The SQL trigger is activated only when column SALARY is updated. System (external) triggers do not have the capability to be activated only when certain columns are updated.

Example 10-2   Compound statement SQL trigger using WITH clause

CREATE TRIGGER SALARY_TRACK
  AFTER UPDATE OF SALARY ON EMPLOYEE
  REFERENCING NEW ROW AS NROW
  OLD ROW AS OROW
  FOR EACH ROW MODE DB2SQL
  WHEN (NROW.SALARY < OROW.SALARY)
  BEGIN
    INSERT INTO SALARY_CTL (EMPNO, NEW_SALARY, OLD_SALARY, UPDATE_TIMESTAMP)
    VALUES (NROW.EMPNO, NROW.SALARY, OROW.SALARY, CURRENT_TIMESTAMP);
  END
10.4 Creating an SQL trigger

Now that you know the general structure of an SQL trigger, you are ready to create one. This section documents the steps required to edit and compile an SQL procedure. There are many ways to build an SQL trigger:

- iSeries Navigator GUI in V5R2
- iSeries Navigator SQL script utility
- Traditional Interactive Structure Query Language (STRSQL)
- Traditional 5250 programming using SEU and RUNSQLSTM utilities

### 10.4.1 Creating an SQL trigger with iSeries Navigator

iSeries Navigator provides an attractive graphical interface that allows you to perform typical database administration tasks. It allows easy access to all server administration tools, gives a clear overview of the entire database system, enables remote database management, and provides assistance for complex tasks.

In this section, you learn how to efficiently use the GUI administration tools offered by iSeries Access Express (5722-XE1) to work with SQL triggers on the iSeries. We expect that you already know how to set up the iSeries Navigator connection to your iSeries server.

The following steps show how to create an SQL trigger using the iSeries Navigator in V5R2. In this trigger program, we assume that an existing row in the SALES table can be changed. One possibility is that the number of the sales transactions (in the SALES column) can be adjusted. Therefore, the sales commission of the sales staff member must also be adjusted accordingly. The adjustment can be an addition or a subtraction of a proper multiplication of 100 (dollars) to the current value.

1. Double-click the iSeries Navigator icon on your desktop. Under My Connections, double-click the iSeries server that you are working on.
2. Double-click the Database icon and right-click. Since V5R2, more than one database may be defined on an iSeries server. Double-click the target database. Then, under Libraries, the library that contains your database. In our case, the name of the library is SAMPLEDB01.
3. After you double-click the library, it shows all the database-related objects in the right panel. Click the table that you want to define the trigger program. In our example, we click the SALES table.
4. Right-click the **SALES** table and click **Properties** as shown in Figure 10-1.

![Figure 10-1 Properties of a table](image-url)
5. In the Table Properties window (Figure 10-2) of the SALES table, click the **Triggers** tab and then click **Add SQL Trigger**.

![Figure 10-2 Adding a trigger to a table](image)
6. The Add SQL Trigger for Table window (Figure 10-3) opens. The window has three tabs:
   General, Timing, and SQL Statements.
   a. The trigger name, COMMADJ in our example, should be typed in the Trigger text box.
   b. The library for our example is SAMPLEDB01.
   c. In the Description field, type a meaningful descriptor.
   d. In our example, we select the **Update of selected columns** radio button.
   e. Under the Available columns list box, we selected **SALES_STAFF** and **SALES**.

![Add SQL Trigger for Table SAMPLEDB01.SALES - Monaco(Monaco)](image)

*Figure 10-3  Add SQL Trigger: General tab*
7. The Timing tab is shown in Figure 10-4.
   a. When to run: Timing for the trigger. Will the trigger be executed before or after the triggering operation?
   b. Run trigger: Will the trigger be executed once per row or once per triggering operation?
   c. Correlation name for old/new row, new row, or Temporary name for old/new table allow us to declare how original and new rows will be referenced in the trigger code.

   **Note:** OldValue and NewValue are called *correlation variables*. They are used in the procedural SQL code of the trigger to refer to the before-image and the after-image of the row being updated.

d. Mode: Determines the order in which the trigger will be executed.
8. The SQL Statements tab is used to type the body of the trigger as shown in Figure 10-5. In our example, even if we specified that the trigger will be fired when an update on columns sales_staff or sales is performed, it was included a test (WHEN clause) testing for differences in those columns. This is because an UPDATE statement updating columns sales_staff or sales is performed, setting exactly the same values as in the original column, will fire our trigger.

![Figure 10-5   SQL Statements tab](image)

9. Click the **Check Syntax** button to ensure that the code that you entered is syntactically correct.

10. Click **OK**, and the new trigger is created to the SALES table.
As stated at the beginning of this chapter, as of V5R3, iSeries Navigator was enhanced and the graphical interface was changed. One of the major changes is that under a certain Schema (Library), you have an icon for each type of database object, including triggers. In Figure 10-6, we illustrate the new interface and note the icon used for triggers.

Figure 10-6  Trigger icon used in V5R3 and later
10.4.2 Creating an SQL trigger with the Run SQL Scripts utility

The Run SQL Scripts utility is yet another interface that you can use on the iSeries server to create an SQL trigger. The script utility is available through the iSeries Navigator GUI. It allows you to create, edit and troubleshoot scripts of SQL statements. You can also save the scripts that you work with on your PC.

To create an SQL trigger using the Run SQL Scripts utility:

1. You can access the Run SQL Script utility in iSeries Navigator. Expand My Connections-> Databases. Right-click the desired database and select RUN SQL Scripts, as shown in Figure 10-7.

![Figure 10-7  Run SQL Script](image)
2. The Run SQL Scripts window opens. Type the complete CREATE TRIGGER statement as shown in Figure 10-8.

**Note:** It may seem that the WHEN clause is redundant with the AFTER UPDATE OF specification, but it prevents the execution of this trigger when the SALES and SALES_STAFF are updated with a value equal to the original one.

```sql
CREATE TRIGGER SAMPLEDB01.COMMADU
    AFTER UPDATE OF SALES, SALES_PERSON ON SAMPLEDB01.SALES
    REFERENCING OLD AS OLDVALUE
    NEW AS NEWVALUE
    FOR EACH ROW
BEGIN
    IF (NEWVALUE.SALES_PERSON = OLDVALUE.SALES_PERSON) THEN
        UPDATE SAMPLEDB01.STAFF
        SET COMM = COMM + (NEWVALUE.SALES - OLDVALUE.SALES) * 100
        WHERE ID = NEWVALUE.SALES_PERSON;
    ELSE
        UPDATE SAMPLEDB01.STAFF
        SET COMM = COMM + NEWVALUE.SALES * 100
        WHERE ID = OLDVALUE.SALES_PERSON;
        UPDATE SAMPLEDB01.STAFF
        SET COMM = COMM + OLDVALUE.SALES * 100
        WHERE ID = OLDVALUE.SALES_PERSON;
    END IF;
END;
```

Figure 10-8  Creating an SQL trigger with the Run SQL Scripts utility
3. To run the CREATE TRIGGER statement, from the Run menu, select **All**. If the syntax of your SQL statement is correct, the SQL trigger is created in the current library on your iSeries server. Check for the completion status in the run history panel of the Run SQL Scripts windows. The last message displayed in the panel should read:

```
Statement ran successfully
```

**Note:** If the run history panel does not supply sufficient information about the execution of the SQL statements, you can view the iSeries server job log to obtain more specific information. From the View menu, select **Job Log**. A job log window opens as shown in Figure 10-9.

![Figure 10-9  Job log window](image_url)

To view second-level messages in the job log, double-click the message you want to view. A window opens showing all of the information for that message.

To save the script that contains the source code for the COMMAJ trigger program, click **File -> Save As** from the script utility menu bar. The Save As window is displayed. In the Save in list combo, open the directory you want to use as your SQL script repository. In our case, we used the `c:\sg24_6503\` directory. Enter `COMMAJ` in the file name input field. Then click **Save** to return to the Run SQL Script window.
To view messages in the job log with debug mode, click **Options -> Include Debug Messages in Job Log**, as shown in Figure 10-10.
This can be useful in a case where the CREATE TRIGGER statement fails to compile. In Figure 10-11, you can see a more detailed job log that has debug messages included.

![Job Log - AS23](image)

**Figure 10-11  Job log with debug message**

### 10.4.3 Creating SQL triggers with traditional interfaces

Another way to create an SQL trigger is to use the traditional 5250 interface and to invoke the interactive SQL screen. On the command line, you must type:

```
STRSQL
```

After the interactive SQL screen opens, type the CREATE TRIGGER command as it was done using the Run SQL Scripts utility. Then press Enter. If the syntax is correct, the trigger is created on the specified library.

Another way to create an SQL trigger is to use SEU as explained in the following steps:

1. Create a library if you do not have one already.
2. Create a source physical file. This is the file where all the SQL source members are to be stored.
3. Start a Source Entry Utility (SEU) edit session.
4. Enter the SQL trigger source code.
5. Create the SQL trigger using the Run SQL Statement (RNSQLSTM) command to issue a CREATE TRIGGER command. This creates a C program object that runs when the trigger is created. If there are problems in generating the trigger, there is a listing that shows the syntax error of the source.
Let us see how to implement this scenario. First, create a library and a source file, and start an editing session. Follow these steps:

1. To create a library called LIBCINTIA, type the following command at the 5250 emulation prompt:
   
   ```bash
   CRTLIB LIB(LIBCINTIA)
   ```

2. To create a source physical file called SOURCE, type the command:
   
   ```bash
   CRTSRCPF FILE(LIBCINTIA/SOURCE) RCDLEN(92)
   TEXT('Source physical file for SQL trigger')
   ```
   
   The CRTSRCPF command creates a source physical file SOURCE in the library LIBCINTIA.

3. To start an editing session and create a source member, COMMADJ, type the command:
   
   ```bash
   STRSEU SRCFILE(LIBCINTIA/SOURCE) SRCMBR(COMMADJ) TYPE(TXT) OPTION(2)
   ```
   
   Entering OPTION(2) indicates that you want to start a session for a new member. The STRSEU command creates a new member, COMMADJ, in the SOURCE file in the LIBCINTIA library and starts an edit session.

4. Type the source code of the CREATE TRIGGER statement using Source Entry Utility (SEU).

5. Use the RUNSQLSTM command to create the SQL trigger. We recommend that you use the Debugging view option *LIST and listing output *PRINT. It is useful for debugging and testing purposes.

---

**Figure 10-12  Creating the SQL trigger using RUNSQLSTM**
6. If there are syntax errors in your source, a message similar to the one shown in Figure 10-13 is issued.

   Command Entry                     ASM23
   Request level:                   11

   Previous commands and messages:
   > RUNSQLSTM SRCFILE(LIBCINTIA/SOURCE) SRCMBR(COMMADJ) COMMIT(*NONE)
   RUNSQLSTM command failed.

   Type command, press Enter.
   ===> RUNSQLSTM SRCFILE(LIBCINTIA/SOURCE) SRCMBR(COMMADJ) COMMIT(*NONE)

   F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
   F11=Display full   F12=Cancel   F13=Information Assistant   F24=More keys

   Figure 10-13   RUNSQLSTM command fails

   Note: If the RUNSQLSTM command fails (see the message in Figure 10-13) to create
   the SQL trigger, go to the listing of the program by typing the Work with Spooled File
   (WRKSPLF) command at the command prompt.

   It is important to note that when you are creating an SQL trigger, the OS/400 creates a
   temporary source file that contains C source code with embedded SQL statements. The
   actual trigger is created as a *PGM object using the CRTSQLCI and CRTPGM
   commands.

   Three listing are generated as shown in Figure 10-14 on page 315:
   
   1. The first listing is the source listing of the generated SQL ILE C program.
   2. This is the listing of the pre-compiled SQL ILE C program.
   3. This is the listing of the source of the SQL trigger. You have to look at this listing in case
      of a failure to create the trigger. You can look at it by typing 5 next to the listing. If the
      SQL trigger has syntax errors, the first two listings are not generated.

Notes: The following notes refer to the previous example.

1. Type the name of the source file (SOURCE) and the library (LIBCINTIA) that you
   created before.
2. The name of the member source file that you typed (COMMADJ).
3. This parameter specified the type of source debug information to be provided by the
   SQL pre-compiler. The possible values are:
   - *STMT: Allows the compiled module object to be debugged using program
     statement numbers and symbolic identifiers.
   - *NONE: The debug view is not generated.
   - *LIST: Generates the listing view for debugging the compiled module object.
4. If you want the pre-compiled listing, type *PRINT.
7. In the preceding listing (Figure 10-15), there is a syntax error that probably generated the other ones. Correct the errors by going to an editing session and executing the RUNSQLSTM command again.

```
18 WHERE ID = OLDVALUE.SALES_STAFF;
19 SELECT COALESCE(COMM, 0) INTO NEWCOMM FROM SAMPLEDB01.STAFF
20 WHERE ID = NEWVALUE.SALES_STAFF;
21 SET NEWCOMM = NEWCOMM + ( 100 * NEWVALUE.SALES ) ;
22 UPDATE SAMPLEDB01.STAFF SET COMM = NEWCOMM
23 WHERE ID = NEWVALUE.SALES_STAFF ;
24 END ;
```
10.4.4 Verifying the SQL trigger properties

After the SQL trigger is successfully created, verify its properties by using the iSeries Navigator interface:

1. In iSeries Navigator, double-click the SAMPLEDB01 library. The right panel displays all DB2 Universal Database iSeries objects in this library.

2. Find and right-click the SALES table, and select Properties as shown in Figure 10-1 on page 302.

3. In the Table Properties window (Figure 10-16), click the Triggers tab. All the triggers that the table has defined are displayed in the window.

4. Select the trigger that you want to see, which in our case is COMMADJ, and click the Properties button.
5. The COMMADJ Properties window (Figure 10-17) that opens has two tabs:
   - The General tab specifies the description, event, when to run, how often to run, operative or no operative, enabled or disabled, type of mode, and creation time stamp.
     * Multiple trigger execution: If multiple triggers are defined for a trigger event and activation time, they run in the creation timestamp order. The trigger's creation time stamp determines which trigger runs at which point.
     * Enabled status: This is a user-controlled activation status of the trigger.
       * If the trigger status is *ENABLED, the trigger activates when the proper event occurs.
       * If the trigger status is *DISABLED, the trigger will not activate when the proper event occurs.
     * Operative status: This a DB2 Universal Database for iSeries controlled status of the trigger.
       * If the trigger can run, this is set to Yes.
       * If the trigger cannot run, this is set to No. The associated subject table cannot be opened. You must drop an inoperative trigger and recreate it.

**Note:** You may have to recreate the subsequent triggers in the execution order after the re-creation of the inoperative trigger, to maintain the original execution order for the creation timestamp order.
The SQL Statements tab contains the code for the SQL trigger that you define in the trigger. Here you find the names of the correlation variables and transition tables if the triggers use them. These topics are covered later in this chapter.

Note: The DSPFD CL command can be used to display information about SQL triggers as well.

10.5 Deleting or replacing an SQL trigger

When you create a trigger program, this trigger program must be unique to register the SQL trigger in the catalog. Since the CREATE TRIGGER statement does not have a replace option. If you want to re-create or replace an existing trigger, use the DROP TRIGGER statement first.

To delete or drop an SQL trigger from the iSeries server, use either of the following options:

- In iSeries Navigator, in the right panel, right-click the table name that contains the trigger and select Properties as shown in Figure 10-1 on page 302. Click the Triggers tab and select the trigger that you want to delete. Then click the Delete button as shown in Figure 10-18.

![Figure 10-18 Deleting a program trigger](image)

Note: The DSPFD CL command can be used to display information about SQL triggers as well.
From any SQL interface, as iSeries Navigator's RUN SQL Script Center, traditional green-screen STRSQL command, among others, perform the following command:

```sql
DROP TRIGGER schema.trigger-name
```

Here `schema` is the library, collection, or schema where the trigger was registered, and `trigger-name` is the respective trigger name.

DB2 Universal Database for iSeries does not allow an SQL trigger to be replaced with another trigger of the same name, same library, or same event/time components. Any attempt to do so results in a message similar to the example in Figure 10-19.

![Figure 10-19 Error message when attempting to replace an SQL trigger](image)

The only way to replace an SQL trigger with another SQL trigger of the same name is to delete the original one and recreate it. This results in a new creation time stamp for it and may affect the order in which triggers will be executed.

You may have to recreate the subsequent triggers in the execution order after the re-creation of the replaced trigger, to maintain the original execution order, which is the creation timestamp order.

Important: It is important to define the order of the trigger execution when there is more than one trigger for an event.

It is possible to use the Add Physical File Trigger (ADDPFTRG) command to replace an SQL trigger with a system (external) trigger of the same name, same library, and same event/time components. This preserves the original creation time stamp of the SQL trigger. However, the new trigger will be a system (external trigger) that may perform quite a different action. To avoid this possibility, we recommend that you use different naming conventions for SQL and system (external) triggers.

### 10.6 Trigger component details

Figure 10-20 shows the CREATE TRIGGER syntax diagram so you can see the options supported by the SQL trigger interface. Although a different interface, SQL triggers for the most part provide equivalent functionality with the current external triggers, as well as a few features not available to external triggers.
Like a table, a trigger has a second part name that includes a schema (library) name. The trigger name must be unique within a schema.
SQL triggers support the same triggering events as external triggers—the update, deletion, or insertion of a row in a table. In addition, an SQL trigger allows an update trigger event to be applied to just a specific column (or a subset of the table columns) instead of every column in the row. The ability to scope a trigger event to a specific column is only supported by SQL triggers. This scoping is accomplished by adding an `OF` clause to the `UPDATE` event specification, for example, `"...UPDATE OF address, city, state, zip"`. This SQL-only option can provide a performance boost by allowing a trigger to only be executed when an individual column or columns are changed instead of every time that any column in the row is changed.

The same activation (trigger) times are supported by both SQL and external triggers. The activation time is always either before or after its triggering event takes place in the database. Triggers are often referred to by the combination of the event and time properties as in a BEFORE INSERT trigger or AFTER UPDATE trigger. As mentioned earlier, multiple triggers can have the same triggering event and time. Due to the standards, a BEFORE SQL trigger does not allow data modifications to be performed by the trigger logic. The SQL standard prevents a BEFORE trigger from ever activating another BEFORE trigger. That activation is prevented by the NO CASCADE clause. Therefore, data modification statements, such as INSERT, UPDATE, DELETE, any CREATE, are not allowed in a BEFORE trigger.

SQL triggers provide an extra control that is not available to external triggers. The SQL statement that causes the trigger to be fired (activated) can affect one row or multiple rows in your database. For example, an SQL DELETE statement can be used to delete one row or multiple rows from a table. When you define an SQL trigger, you can specify whether the trigger is to be called only once for such an SQL statement (FOR EACH STATEMENT) or once for each row that is modified (FOR EACH ROW). This is referred to as the granularity or orientation of the trigger. The two types of triggers are referred to as statement-level triggers or row-level triggers. External triggers are always defined as row-level triggers. If one row is deleted by the DELETE statement, then a statement-level trigger would be called once and a row-level trigger would be called once; when 10 rows are deleted, the statement-level trigger is called once, but a row-level trigger would be called 10 times. If zero rows are deleted, a statement-level trigger is still called once, but a row-level trigger would not be called at all. The statement level option can only be specified on AFTER triggers. If more than one trigger exists per event, the execution of statement and row-level triggers is intermixed based on each creation order.

Another SQL-only option for triggers is the trigger mode. Two trigger modes are available to SQL triggers: DB2SQL and DB2ROW. DB2SQL is the mode supported by the other DB2 Universal Database products. DB2ROW is the mode that has always been used by external triggers. The trigger mode causes a difference in the execution of the trigger.

The DB2ROW mode causes the trigger to be fired after each row change. Not surprisingly, this mode can only be specified on row-level triggers. In contrast, the DB2SQL mode waits until all of the row changes are made and then calls the trigger. This difference is not the same as the Row versus Statement level granularity. We give an example of a delete row-level trigger using both modes to better understand the differences. If the triggering delete statement deletes 10 rows, then the DB2ROW version of the trigger is called after each row is deleted, a total of 10 times. The DB2SQL version of this delete trigger is also called 10 times. However, with the DB2SQL mode the trigger is not called until all 10 rows are deleted. The trigger data for each row deletion is saved. Then after all the rows were deleted, this saved data is used to call the trigger for each deleted row. This means that the DB2SQL mode is less efficient than DB2ROW since each row is effectively processed twice. DB2SQL mode is only supported on AFTER triggers. If DB2SQL is specified on a BEFORE trigger, it is automatically converted to the DB2ROW mode.
Despite the differences in mode behavior, the final outcome of a trigger, in general, is the same if the DB2SQL and DB2ROW modes were used with the same trigger. The one exception to that is a self-referencing trigger. If a DB2ROW trigger performs an aggregate operation, such as a count against the trigger's underlying table, then it can get a different value than a DB2SQL trigger. The DB2SQL aggregate operation is not executed until after all row operations have completed; the DB2ROW aggregate function is executed against the table after each row operation. If the trigger logic did a count of the rows in the underlying table, the DB2SQL trigger returns the same count value (total number of rows = 10) during each execution. The DB2ROW mode row count is different on each activation of the trigger (total number of rows = 1 on the first execution, total number of rows = 2 on the second execution, and so on).

### 10.6.1 Trigger time

The activation time is always either before or after its triggering event place in the table. Triggers are often referred to by the combination of the event and time properties as in a BEFORE insert or AFTER.

**BEFORE**

This specifies that the trigger is a BEFORE trigger. The database manager executes the triggered-action before it applies any changes caused by an insert, delete, or update operation on the subject table. It also specifies that the triggered-action does not activate other triggers because the triggered-action of a BEFORE trigger cannot contain any updates.

BEFORE triggers may not modify tables, but they can be used to verify input column values and modify column values that are inserted or updated in a table. In Example 10-3, the trigger is used to set the fiscal quarter for the corporation prior to inserting the row into the target table.

*Example 10-3  SQL BEFORE trigger using correlation variables*

```sql
CREATE TRIGGER sampledb07.TransactionBeforeTrigger BEFORE INSERT ON sampledb07.TransactionTable
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE newmonth SMALLINT;
  SET newmonth = MONTH (new_row.DateOfTransaction);
  IF newmonth <4 THEN
    SET new_row.FiscalQuarter=3;
  ELSEIF newmonth <7 THEN
    SET new_row.FiscalQuarter=4;
  ELSEIF newmonth <10 THEN
    SET new_row.FiscalQuarter=1;
  ELSE
    SET new_row.FiscalQuarter=2;
  END IF;
END
```
AFTER

This specifies that the trigger is an AFTER trigger. The database manager executes the triggered-action after it applies any changes caused by an insert, delete, or update operation on the subject table.

Example 10-4 maintains an accumulated sales commission for a sales staff when someone makes a sales transaction. This is done when a new row is added into the SALES table. A value of 100 (dollars) is multiplied by the accomplished number of transactions, and the result is added and updated into the COMM column of the STAFF table.

Example 10-4 SQL AFTER trigger using correlation variables

```sql
CREATE TRIGGER SAMPLEDB01.COMMCALE
AFTER INSERT ON SAMPLEDB01.SALES
REFERENCING NEW AS NEWSALESROW
FOR EACH ROW
MODE DB2ROW
BEGIN
  DECLARE NEWCOMM DECIMAL ( 7 , 2 ) ;
  SET NEWCOMM = 0 ;
  SELECT QSYS2.COALESCE ( SAMPLEDB01.STAFF.COMM, 0 ) INTO
    NEWCOMM FROM SAMPLEDB01.STAFF WHERE
    SAMPLEDB01.STAFF.ID = NEWSALESROW.SALES_STAFF ;
  SET NEWCOMM = NEWCOMM + ( 100 * NEWSALESROW.sales) ;
  UPDATE SAMPLEDB01.STAFF
  SET COMM = NEWCOMM
  WHERE SAMPLEDB01.STAFF.ID = NEWSALESROW.SALES_STAFF ;
END
```

10.6.2 Trigger modes

Two trigger modes are available to SQL triggers are:

- DB2SQL
- DB2ROW

MODE DB2SQL

This mode is supported by other DB2 Universal Database products. This trigger mode causes a subtle difference in the execution of the trigger:

- The DB2SQL mode waits until all of the row changes are made and then calls the trigger.
- The DB2SQL mode triggers are activated after all of the row operations occur.
- The DB2SQL mode specifies that the trigger program will not run for any row until after all rows affected by the operation are modified. All rows appear to the trigger program to have their new values.

MODE DB2SQL triggers also run in creation timestamp order. If there are multiple MODE DB2SQL triggers defined for an event, execution of FOR EACH ROW and FOR EACH STATEMENT triggers can intermix.

For non-SQL I/O, the MODE DB2SQL statement triggers are executed on each row. (In this case, statement-level triggers operate the same way as row-level triggers. For High Level Language (HLL), there is no concept of multiple row updates; each I/O is for a single record, so consider I/O as a “statement”.)
For embedded SQL, the MODE DB2SQL statement triggers work as stated, once for each SQL statement that is executed. The MODE DB2SQL statement trigger is only executed once for each statement, even though multiple rows may be updated.

Figure 10-21 shows the message ID that you receive when the DB2SQL mode is used for a BEFORE trigger.

**MODE DB2ROW**

The DB2ROW mode causes the trigger to be fired after each row change. Not surprisingly, this mode can only be specified on row-level triggers. Rows that are not yet modified by the operation appear to the trigger program to have their original values, while rows already modified show the new values.

MODE DB2ROW triggers are activated on each row operation. If multiple MODE DB2ROW triggers are defined for an event, they run in creation timestamp order.

**Comparison between DB2ROW and DB2SQL**

The best way to understand the differences in behavior is with an example. Let us suppose we are interested in registering the new salary position of employees when their salaries change. Let us say that the salary position is measured as the percentage of employees that have a salary less than or equal to the new salary of the employee.
The salary position is calculated by INSERT and UPDATE triggers and stored in SALARYPOSITION table. The UPDATE trigger is shown in Example 10-5.

**Example 10-5 SQL trigger showing the difference between DB2ROW and DB2SQL**

```sql
CREATE TRIGGER salaryposition
    BEFORE UPDATE OF SALARY ON EMPLOYEE
    REFERENCING NEW ROW AS newrow
    FOR EACH ROW MODE DB2ROW
BEGIN
    DECLARE v_emp_count INTEGER;
    DECLARE v_le_sal_emp_count INTEGER;
    -- count number of employees
    SET v_emp_count = (SELECT COUNT(*) FROM EMPLOYEE);
    -- count how many employees have a salary less than or equal to
    -- the new salary for this employee
    SET v_le_sal_emp_count = (SELECT COUNT(*) FROM EMPLOYEE
        WHERE SALARY <= newrow.SALARY);
    -- establishes the salary position for the updated employee
    INSERT INTO SALARYPOSITION (EMPNO, SALARY, SALPOS)
        (newrow.EMPNO, newrow.SALARY,
        100.00 * CAST(v_le_sal_emp_count AS DEC(5,2)) / CAST(v_emp_count AS DEC(5,2)));
END;
```

Now if we perform an UPDATE EMPLOYEE SET SALARY = 20000, we obtain results like those shown on the left side of Figure 10-22. If the trigger is changed for a similar version but DB2SQL mode, the results of this operation are like those shown on the right side of Figure 10-22.

![Figure 10-22 Comparing DB2ROW and DB2SQL](image-url)
In **DB2ROW** mode, the operation sequence for this example is:

Modify row 1
Calls trigger for row 1
Modify row 2
Calls trigger for row 2
...
Modify last row
Calls trigger for last row

Then when it modifies the first row and calls the trigger, there will be some rows with a salary below and others above $20,000, and consequently it will calculate the corresponding salary position.

In **DB2SQL** mode, the operation sequence for this example is:

Modify row 1
Modify row 2
...
Modify row n
 Calls trigger for row 1
Calls trigger for row 2
...
Calls trigger for row n

Then when the trigger is called for the first row, all rows are updated with a salary of $20,000. This means that for every employee, their salary position will be 100 percent, because the others will have a salary below or equal to their salary.

You cannot say that one mode is better than the other. It depends on how the business defines its needs. And there will be cases in which a **DB2ROW** more accurately reflects the business needs, and others in which a **DB2SQL** mode will fit better. Probably for most cases, it does not make any difference. If this is the case, you must consider that **DB2ROW** performs better.

**Note:** As a general rule, try to avoid a trigger design in which the final result depends on the mode (**DB2ROW** or **DB2SQL**) or the order of execution.

Also note that external triggers are always created with the **FOR EACH ROW** and **DB2ROW** mode.

### 10.6.3 Trigger granularity

There are two granularities for SQL triggers:

- **FOR EACH ROW**
- **FOR EACH STATEMENT**

This component is an extra control that is not available to external triggers. This defines whether the actions of the trigger will be performed once **FOR EACH ROW** or once **FOR STATEMENT**.

**FOR EACH ROW**

This specifies that the database manager executes the triggered-action for each row of the subject table that the triggering operation modifies. If the triggering operation does not modify any rows, the triggered-action is not executed.
Depending on the mode used, activation time, and triggering operation, the trigger program may or may not have access to the correlation variables or transition tables, as shown in Table 10-1.

**Table 10-1 FOR EACH ROW allowable combinations of correlation variables, transition tables**

<table>
<thead>
<tr>
<th>MODE</th>
<th>Activation time</th>
<th>Triggering operation</th>
<th>Correlation variables allowed</th>
<th>Transition tables allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2ROW</td>
<td>BEFORE</td>
<td>DELETE</td>
<td>OLD</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2ROW</td>
<td>BEFORE</td>
<td>INSERT</td>
<td>NEW</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2ROW</td>
<td>BEFORE</td>
<td>UPDATE</td>
<td>OLD, NEW</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2ROW</td>
<td>AFTER</td>
<td>DELETE</td>
<td>OLD</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2ROW</td>
<td>AFTER</td>
<td>INSERT</td>
<td>NEW</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2ROW</td>
<td>AFTER</td>
<td>UPDATE</td>
<td>OLD, NEW</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>BEFORE</td>
<td>DELETE</td>
<td>OLD</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>BEFORE</td>
<td>INSERT</td>
<td>NEW</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>BEFORE</td>
<td>UPDATE</td>
<td>OLD, NEW</td>
<td>NONE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>AFTER</td>
<td>DELETE</td>
<td>OLD</td>
<td>OLD TABLE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>AFTER</td>
<td>INSERT</td>
<td>NEW</td>
<td>NEW TABLE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>AFTER</td>
<td>UPDATE</td>
<td>OLD, NEW</td>
<td>OLD TABLE, NEW TABLE</td>
</tr>
</tbody>
</table>

**FOR EACH STATEMENT**

This specifies that the database manager executes the triggered-action only once for the triggering operation. A FOR EACH STATEMENT trigger is activated even when no rows are affected by the triggering statement.

**Restriction:** FOR EACH STATEMENT can only be specified for an activation time of AFTER and for a DB2SQL mode.

For native database changes, there is no difference between row level and statement level since native databases can only perform single row operations.

When you execute MODE DB2SQL triggers, FOR EACH STATEMENT always runs after FOR EACH ROW if they are defined on the same table for the same event. The correct concept is that a trigger's creation time stamp determines which trigger runs at which point. Depending on the activation time and triggering operation, the trigger program may or may not have access to the correlation variables or transition tables, as shown in Table 10-2.

**Table 10-2 FOR EACH STATEMENT allowable combinations of correlation variables, transition tables**

<table>
<thead>
<tr>
<th>MODE</th>
<th>Activation time</th>
<th>Triggering operations</th>
<th>Correlation variables allowed</th>
<th>Transition tables allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2SQL</td>
<td>AFTER</td>
<td>DELETE</td>
<td>NONE</td>
<td>OLD TABLE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>AFTER</td>
<td>INSERT</td>
<td>NONE</td>
<td>NEW TABLE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>AFTER</td>
<td>UPDATE</td>
<td>NONE</td>
<td>OLD TABLE, NEW TABLE</td>
</tr>
</tbody>
</table>
10.7 Accessing triggering data

When an external trigger is activated, it often uses the data from the triggering database request. For example, an insert trigger may put the data for the new row onto a data queue. This data was made available to external triggers via the trigger buffer. SQL triggers have access to the same data, but through a different interface known as row transition variables and transition tables. The REFERENCING clause is used to specify the correlation name for the row transition variables (or correlation variables) and to specify the table name of the transition tables.

Correlation variables closely match the record image values passed in the external trigger buffer. The old row variable maps to the old record image by representing the value of the modified row before the triggering event. The new row variable maps to the new record image by representing the value of the modified row after the triggering event. The SQL trigger, however, makes it much easier to access the old and new values. The trigger only has to qualify the column name with the defined correlation name. Example 10-6 uses a new correlation variable to copy the newly inserted column values into a travel expense audit table.

Example 10-6   Accessing triggering data on SQL triggers - Correlation variables

```
CREATE TRIGGER big_spenders
AFTER INSERT ON expenses
REFERENCING NEW AS n
FOR EACH ROW MODE DB2ROW
WHEN (n.totalamount > 10000)
INSERT INTO travel_audit
VALUES(n.empno, emplname, n.deptno, n.totalamount, n.enddate);
```

Notes: The following notes refer to Example 10-6:

1. Specify the correlation name for the row transition variables (or correlation variables) and specify the table name of the transition table.

2. The clause WHEN allows you to conditionally execute your trigger logic.

The transition table variables represent hypothetical, temporary read-only tables. The old table variable contains all of the modified rows as they appeared before the triggering event. The new table variable contains all of the modified rows as they appeared after the triggering event. If you have an after update trigger that was triggered by an UPDATE statement that updates 10 rows in the table, then the transition tables accessed in the update trigger contain 10 rows. The transition tables may be used like any other table in the trigger definition as long they are not modified. Example 10-7 uses a transition table in an update trigger to keep track of the number of account changes. Transition tables can only be used with AFTER triggers.

Example 10-7   Accessing triggering data on SQL triggers - Transition tables on per row trigger

```
CREATE TRIGGER accounttrigger
AFTER UPDATE ON accounts
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
BEGIN
DECLARE chgcnt INT;
SET chgcnt = (SELECT count(*) FROM newtable);
INSERT INTO account_changes
VALUES('I',CURRENT TIMESTAMP,CURRENT USER, chgcnt);
END
```

---

Notes:

The following notes refer to Example 10-6:

1. Specify the correlation name for the row transition variables (or correlation variables) and specify the table name of the transition table.

2. The clause WHEN allows you to conditionally execute your trigger logic.
Chapter 10. SQL triggers

10.7.1 Correlation variables

The triggered action may refer to the values in the set of affected rows. This concept is similar to the concept of the trigger buffer used in external triggers. In SQL triggers, this is supported through the use of transition variables or correlation variables. Transition variables use the names of the columns in the subject table qualified by a specified name that identifies whether the reference is to the old value (prior to the update) or the new value (after the update). The new value can also be changed using the SET transition-variable statement in before update or insert triggers.

The REFERENCING clause is used to specify the correlation name for the row transition variables (or correlation variables) and to specify the table name of the transition tables.

REFERENCING

This specifies the correlation names for the transition variables and the table names for the transition tables. Correlation names identify a specific row in the set of rows affected by the triggering SQL operation. Table identifiers identify the complete set of affected rows. Each row affected by the triggering SQL operation is available to the triggered action by qualifying columns with correlation-names specified as follows:

Notes: The following notes refer to Example 10-7:

1. Specify the correlation name for the row transition variables (or correlation variables) and specify the table name of the transition table.
2. Use a transition table in a update trigger to keep track of the number of account changes.
OLD ROW AS correlation-name

Specifies a correlation name that identifies the values in the row prior to the triggering SQL operation. It also specifies the old value correlation name. These rows contain the state of the row before the trigger runs. The old correlation name can only be specified if the trigger event is DELETE or UPDATE.

NEW ROW AS correlation-name

Specifies a correlation name that identifies the values in the row as modified by the triggering SQL operation and any SET statement in a BEFORE trigger that already executed. These rows contain the state of the row after the trigger runs. The new correlation name can only be specified if the trigger event is INSERT or UPDATE. The maximum length of correlation names is 128 characters. These correlation names are optional.

10.7.2 Transition tables

An SQL trigger may need to refer to all of the affected rows for an SQL INSERT, UPDATE, or DELETE operation. This is true, for example, if the trigger needs to apply aggregate functions, such as MIN or MAX, to a specific column of the affected rows. The OLD_TABLE and NEW_TABLE transition tables can be used for this purpose. In Example 10-8, the trigger applies the aggregate function MAX to all of the affected rows of the table StudentProfiles.

Example 10-8   Accessing triggering data on SQL triggers - Transition tables on per statement trigger

CREATE TRIGGER SAMPLEDB07.UpdateCollegeBoundStudentsProfileTrigger
AFTER UPDATE ON SAMPLEDB07.StudentProfiles
REFERENCING NEW_TABLE AS ntable
FOR EACH STATEMENT MODE DB2SQL
BEGIN
  DECLARE maxStudentYearInSchool SMALLINT ;
  SET maxStudentYearInSchool =
    (SELECT MAX (StudentsYearInSchool)FROM ntable);  
  IF maxStudentYearInSchool >
    (SELECT MAX (YearInSchoolMax)FROM
      SAMPLEDB01.CollegeBoundStudentsProfile)THEN
    UPDATE SAMPLEDB07.CollegeBoundStudentsProfile SET YearInSchoolMax =
      maxStudentYearInSchool;
  END IF;
END

In Example 10-8, the trigger is executed a single time following the execution of a triggering UPDATE statement because it is defined as a FOR EACH STATEMENT trigger. You need to consider the processing overhead required by the database management system to populate the transition tables when you define a trigger that references transition tables.

Notes: The following notes refer to Example 10-8:

1. ntable is the new transition table for all the triggering rows.
2. Apply aggregate functions, such as MAX or MIN, to a specific column of the affected rows.

Let us suppose that we need to feed up a summary table containing changes in salaries performed against the EMPLOYEE table, aggregated by WORKDEPT and JOB. We will construct three triggers, for INSERT, DELETE and UPDATE. In Example 10-9, we show only the UPDATE trigger.
Example 10-9  Using transition tables in SQL trigger

CREATE TRIGGER SAMPLEDB01.CostOfChangesInSalary
AFTER UPDATE OF SALARY, COMM, BONUS, JOB, WORKDEPT ON SAMPLEDB01.EMPLOYEE
REFERENCING OLD_TABLE AS O_TBL
    NEW_TABLE AS N_TBL
FOR EACH STATEMENT
BEGIN
    INSERT INTO SAMPLEDB01.INCOME_CHG_SMM
    (WORKDEPT, JOB, CHG_SALARY, CHG_BONUS, CHG_COMM, AVG_CHG_SALARY,
    AVG_CHG_BONUS, AVG_CHG_COMM, AFFECTED_ROWS, CHG_TMS, CHG_USER)
    WITH AA AS (SELECT
                WORKDEPT, JOB, SUM(SALARY) TOT_SALARY,
                SUM(BONUS) TOT_BONUS, SUM(COMM) TOT_COMM,
                COUNT(*) AFFECTED_ROWS
        FROM
            O_TBL
        GROUP BY
            WORKDEPT, JOB),
    BB AS (SELECT
                WORKDEPT, JOB, SUM(SALARY) TOT_SALARY,
                SUM(BONUS) TOT_BONUS, SUM(COMM) TOT_COMM,
                COUNT(*) AFFECTED_ROWS
        FROM
            N_TBL
        GROUP BY
            WORKDEPT, JOB)
    SELECT
        AA.WORKDEPT, AA.JOB,
        COALESCE(BB.TOT_SALARY, 0) - COALESCE(AA.TOT_SALARY, 0),
        COALESCE(BB.TOT_BONUS, 0) - COALESCE(AA.TOT_BONUS, 0),
        COALESCE(BB.TOT_COMM, 0) - COALESCE(AA.TOT_COMM, 0),
        (COALESCE(BB.TOT_SALARY, 0) - COALESCE(AA.TOT_SALARY, 0)) / AA.AFFECTED_ROWS,
        (COALESCE(BB.TOT_BONUS, 0) - COALESCE(AA.TOT_BONUS, 0)) / AA.AFFECTED_ROWS,
        (COALESCE(BB.TOT_COMM, 0) - COALESCE(AA.TOT_COMM, 0)) / AA.AFFECTED_ROWS,
        AA.AFFECTED_ROWS,
        CURRENT_TIMESTAMP,
        USER
    FROM
        AA LEFT JOIN BB ON (AA.WORKDEPT = BB.WORKDEPT AND AA.JOB = BB.JOB)
    UNION ALL
    SELECT
        BB.WORKDEPT, BB.JOB,
        COALESCE(BB.TOT_SALARY, 0),
        COALESCE(BB.TOT_BONUS, 0),
        COALESCE(BB.TOT_COMM, 0),
        COALESCE(BB.TOT_SALARY, 0) / BB.AFFECTED_ROWS,
        COALESCE(BB.TOT_BONUS, 0) / BB.AFFECTED_ROWS,
        COALESCE(BB.TOT_COMM, 0) / BB.AFFECTED_ROWS,
        BB.AFFECTED_ROWS,
        CURRENT_TIMESTAMP,
        USER
    FROM
        BB EXCEPTION JOIN AA ON (BB.WORKDEPT = AA.WORKDEPT AND BB.JOB = AA.JOB);
END;
10.7.3 Changing values in the firing row

Allow Repeated Change (ALWREPCHG) is another external trigger option that you are probably wondering whether SQL triggers support. SQL triggers are always created with the ALWREPCHG(*YES) value. External triggers use this option primarily to enable the modification of data in the new record image and have those changes used in the actual underlying database change. SQL triggers provide a similar capability through a different mechanism. Unlike external triggers specified with *YES, SQL triggers are not allowed to update or delete the triggering row (known as a destructive data change).

With a BEFORE INSERT or UPDATE SQL trigger, you use only the SET or SELECT INTO statements to change the value of the new correlation variables. Then those updated values are used by DB2 Universal Database on the insert or update request. The new correlation variables can also be changed by passing them as OUT or INOUT parameters on a stored procedure call. The following examples use the SET statement to ensure that the state value is in uppercase and uses an external stored procedure call to assign a unique order number.

The ALWREPCHG(*NO/*YES) parameter of the Add Physical File Trigger (ADDPFTRG) command controls repeated changes under commitment control. Changing from the default value to ALWREPCHG(*YES) allows the same record or updated record associated with the trigger program to repeatedly change.

The Allow Repeated Change ALWREPCHG(*YES) parameter on the Add Physical File Trigger (ADDPFTRG) command also affects trigger programs defined to be called before INSERT and UPDATE database operations. If the trigger program updates the new record in the trigger buffer and ALWREPCHG(*YES) is specified, the actual INSERT or UPDATE operation on the associated physical file uses the modified new record image. This option can be helpful in trigger programs that are designed for data validation and data correction. Because the trigger program receives physical file record images (even for logical files), the trigger program may change any field of that record image.

Example 10-10 SQL trigger modifying the firing row

```
CREATE TRIGGER order_completion
    BEFORE INSERT orders
    REFERENCING NEW AS n
    FOR EACH ROW MODE DB2ROW
BEGIN
    SET n.ord_state = UPPER(n.state);
    CALL GenOrderNumber(n.ord#);
END
```

Notes: The following notes refer to Example 10-10:

1. Use the SET statement to ensure that the state value is in uppercase.
2. Use an external stored procedure call to assign a unique order number.

10.8 Error handling

When we talk about error handling in SQL triggers, we need to distinguish between:

- Monitoring and managing error conditions that occur inside an SQL trigger
- Signaling or firing an error from an SQL trigger to avoid the completion of an operation
Managing error conditions that occur inside an SQL trigger is similar to the process described in 8.2, “Error handling in SQL stored procedures” on page 223, by using:

- Condition handlers (except UNDO handlers)
- SQLCODE and SQLSTATE variables
- GET DIAGNOSTIC statement

The differences are in the way you signal or cause an error from a trigger and the way in which the application program retrieves information about such an error. In a stored procedure, an application program calls the stored procedure and receives directly any error condition signaled by the called procedure. In the case of triggers, application programs do not call triggers. Instead, they call database operations such as INSERT, UPDATE, and DELETE, and DB2 Universal Database for iSeries calls the trigger. Errors signaled by a trigger are caught by the database manager. Then the database manager signals an SQLCODE -723 that corresponds to an SQLSTATE 09000 to the application program, as shown in Figure 10-23.

The application program can recover the fired SQLSTATE code by examining the text in the error message. Another important difference is that RETURN statement and UNDO handlers are not allowed in SQL triggers.

Figure 10-23   Comparing error signaling in triggers and stored procedures
10.8.1 Signaling errors from a trigger

We may be interested in firing an error from an SQL trigger to avoid the completion of an INSERT, UPDATE, or DELETE operation, to reinforce a business rule. Example 10-11 shows a column-level trigger that uses the SIGNAL statement to stop any update of salary that exceeds the employee's manager salary.

Example 10-11 SQL trigger signaling error

```sql
CREATE TRIGGER salarycheck
    BEFORE UPDATE OF SALARY ON EMPLOYEE
    REFERENCING NEW AS n
    FOR EACH ROW MODE DB2ROW
BEGIN
    DECLARE v_manager CHAR(6);
    DECLARE v_max_salary DEC(11,2);
    DECLARE v_adm_dept CHAR(3);
    DECLARE v_error_msg VARCHAR(256);
    block: BEGIN
        -- determines the employee's manager
        SELECT MGRNO, ADMRDEPT INTO v_manager, v_adm_dept
        FROM DEPARTMENT WHERE DEPTNO = n.WORKDEPT;
        IF n.EMPNO = v_manager THEN
            -- EMPLOYEE IS A MANAGER... LOOKS FOR ITS MANAGER IN THE ADM DEPT
            IF n.WORKDEPT = v_adm_dept THEN
                -- EMPLOYEE IS GENERAL MANAGER. NO CONTROL APPLIES
                LEAVE block;
            ELSE
                SET v_manager = (SELECT MGRNO FROM DEPARTMENT
                    WHERE DEPTNO = v_adm_dept);
            END IF;
        END IF;
        SET v_max_salary = (SELECT SALARY FROM EMPLOYEE
            WHERE EMPNO = v_manager);
        -- EMP'S salaries must be less that MGR's salary
        IF n.SALARY >= v_max_salary THEN
            SET v_error_msg = 'SALARY EXCEEDS MANAGER''S SALARY';
            SIGNAL SQLSTATE VALUE 'SA001' SET MESSAGE_TEXT = v_error_msg;
        END IF;
    END;
END;
```

Code comments

The following comments refer to Example 10-11:

1. Because the RETURN statement in an SQL trigger is not allowed, we use a compound SQL Statement to make it easier to leave the trigger in any place in the code.
2. If the modified employee record corresponds to the general manager, no action is required, and we leave the trigger.
3. If the error message is too long, it will be truncated.
4. Since the new salary is equal to or exceeds its manager salary, the operation must be avoided by signaling an error condition.
For those of you that remain in V5R1, there is a circumvention to the nested compound statements limitation.

**Example 10-12  SQL trigger signaling error - V5R1**

```sql
CREATE TRIGGER salarycheck
    BEFORE UPDATE OF SALARY ON EMPLOYEE
    REFERENCING NEW AS n
    FOR EACH ROW MODE DB2ROW
BEGIN
    DECLARE v_manager CHAR(6);
    DECLARE v_max_salary DEC(11,2);
    DECLARE v_adm_dept CHAR(3);
    DECLARE v_error_msg VARCHAR(256);

    -- pseudoloop to make easy to leave the trigger from any point in the code.
    pseudoloop: LOOP
    -- determines the employee's manager
    SELECT MGRNO, ADMRDEPT INTO v_manager, v_adm_dept
    FROM DEPARTMENT WHERE DEPTNO = n.WORKDEPT;
    IF n.EMPNO = v_manager THEN
        -- EMPLOYEE IS A MANAGER... LOOKS FOR ITS MANAGER IN THE ADM DEPT
        IF n.WORKDEPT = v_adm_dept THEN
            -- EMPLOYEE IS GENERAL MANAGER. NO CONTROL APPLIES
            LEAVE pseudoloop;
        ELSE
            SET v_manager = (SELECT MGRNO FROM DEPARTMENT
                                WHERE DEPTNO = v_adm_dept);
        END IF;
    ELSE
        SET v_max_salary = (SELECT SALARY FROM EMPLOYEE
                                WHERE EMPNO = v_manager);
        -- EMP'S salaries must be less that MGR's salary
        IF n.SALARY >= v_max_salary THEN
            SET v_error_msg = 'SALARY EXCEEDS MANAGER\'S SALARY'
            SIGNAL SQLSTATE VALUE 'SA001' SET MESSAGE_TEXT = v_error_msg;
        END IF;
    END IF;
    LEAVE pseudoloop;
END LOOP;
END
```

Any update statement that breaks these business rules fail with the message shown in Figure 10-24.

![Figure 10-24  Error message for update statement that breaks the business rules](image)

**10.8.2 Recovering errors fired by triggers**

Since triggers are not called directly from application programs, error messages fired by triggers are received with a common SQLCODE of -723 and SQLSTATE of 09000. The original SQLSTATE code signaled or fired inside the trigger can be recovered from the application program looking at the text in the received error message.

```sql
[SQL0723] SQL trigger SALARYCHECK in SAMPLEDB02 failed with SQLCODE -438 SQLSTATE SA001.
Cause . . . . . :   An error has occurred in a triggered SQL statement in trigger SALARYCHECK in library SAMPLEDB02. The SQLCODE is -438, the SQLSTATE is SA001, and the message is SALARY EXCEEDS MANAGER'S SALARY. Recovery . . . :   Refer to the joblog for more information regarding the detected error. Correct the error and try the request again.
```

---

Chapter 10. SQL triggers  335
The sample application program in Example 10-13 shows how to retrieve the signaled error code.

Example 10-13  Java client recovering trigger error

```java
import java.sql.*;  //SQL APIs.
import java.util.*; //standard classes like Properties.
import java.math.*; //BigDecimal class.
public class ErrorHandlingExample {
    private static final String SYSTEM      = "localhost";
    private static final String USER        = "*current";
    private static final String PASSWORD    = "*current";
    private static final String DATA_LIBRARY = "SAMPLEDB02";
    private static Connection dbConnect;
    public static void main(String[] args) {
        try {
            if (args.length != 2)
                throw new Exception("Wrong number of parameters");
            String empno = args[0];
            BigDecimal salary = new BigDecimal(args[1]);

            // Register AS/400 JDBC driver
            Class.forName("com.ibm.as400.access.AS400JDBCDriver");

            // Establishing environmental properties
            Properties jdbcProperties = new Properties();
            jdbcProperties.put("user", USER);
            jdbcProperties.put("password", PASSWORD);
            jdbcProperties.put("naming", "sql");
            jdbcProperties.put("errors", "full");
            jdbcProperties.put("date format", "iso");
            jdbcProperties.put("extended dynamic", "true");
            jdbcProperties.put("package", "JAVAtext");

            // Connect
            dbConnect = DriverManager.getConnection("jdbc:as400:" + SYSTEM + "/" + DATA_LIBRARY, jdbcProperties);

            // define a statement
            PreparedStatement stmt = dbConnect.prepareStatement(
                "UPDATE SAMPLEDB02.EMPLOYEE SET SALARY = ? WHERE EMPNO = ?");

            // set up parameter markers
            stmt.setBigDecimal(1, salary);
            stmt.setString(2, empno);

            // execute the specified statement
            System.out.println("Updating " + empno + " salary to " + salary.toString());
            stmt.executeUpdate();
            if (stmt.getUpdateCount() == 0)
                System.out.println("Employee " + empno + " not found");

            // disconnect
            if (stmt != null) stmt.close();
            if (dbConnect != null) dbConnect.close();
        } catch (SQLException e) {
            System.out.println("=======================================");
            if (e.getSQLState().compareTo("09000") == 0) {
                System.out.println("Error in a trigger
");
                String message = e.getMessage();
                String fired_state = message.substring(message.indexOf("SQLSTATE") + 9,
                System.out.println("\n" + fired_state);
            }
        }
    }
}
```
message.indexOf("SQLSTATE")+14);
        System.out.println("The user defined SQLState is: " + fired_state);
        System.out.println("nSQL Message: " + message);
    }
    else
        System.out.println("SQL Exception: " + e);
        System.exit(1);
}

    catch (Exception e) {
        System.out.println("Usage:");
        System.out.println(
            "   java ErrorHandlingExample <employee number> <new salary>"");
        System.exit(1);
    }
}
}

**Code comments**

The following notes refer to Example 10-13:

1. Review how many rows are updated.
2. Catch any SQL errors, including trigger errors.
3. Trigger errors have SQLSTATE of ’09000’. Instead of checking SQLSTATE, it is possible to check for SQLCODE -723, but checking SQLSTATE makes the code more portable.
4. The SQLSTATE fired or signaled in the trigger can be found into the text of the error message.

The code shown in Figure 10-25 is the result of executing the previous program:

```
java Chapter_11/ErrorHandlingExample 000030 125000
Updating 000030 salary to 125000
=======================================
Error in a trigger

The user defined SQLState is: SA001

SQL Message: [SQL0723] SQL trigger SALARYCHECK in SAMPLEDB02 failed with SQLCODE -438 SQLSTATE SA001. Cause . . . . . : An error has occurred in a triggered SQL statement in trigger SALARYCHECK in library SAMPLEDB02. The SQLCODE is -438, the SQLSTATE is SA001, and the message is SALARY EXCEEDS MANAGER'S SALARY. Recovery . . . : Refer to the joblog for more information regarding the detected error. Correct the error and try the request again.
```

*Figure 10-25  Result of running the program shown in Example 10-13*

Refer to Chapter 7, “Java stored procedures” on page 173, for more details about error handling.

### 10.9 Inoperative triggers

An *inoperative trigger* is a trigger that is no longer available to be activated. If a trigger becomes inoperative, no INSERT, UPDATE, or DELETE operations are allowed on the subject table. A trigger becomes inoperative if one of the following conditions is true:
The SQL statements in the triggered-action reference the subject-table, the trigger is a self-referencing trigger, and the table is duplicated using the system CRTDUPOBJ CL command.

The SQL statements in the triggered-action reference tables or views in the from library and the objects are not found in the new library when the table is duplicated using the system CRTDUPOBJ CL command.

The table is restored to a new library using the system RSTOBJ or RSTLIB CL commands, the triggered-action references the subject-table, and the trigger is a self-referencing trigger.

An inoperative trigger must first be dropped before it can be recreated by issuing a CREATE TRIGGER statement. Note that dropping and recreating a trigger affects the activation order of a trigger if multiple triggers for the same triggering operation and activation time are defined for the subject table.

10.10 Moving into production (save and restore)

When an SQL table is restored, the definitions for the SQL triggers that are defined for the table are also restored. The SQL trigger definitions are automatically added to the SYSTRIGGERS, SYSTRIGDEP, SYSTRIGCOL, and SYSTRIGUPD catalogs. The program object that is created from the SQL CREATE TRIGGER statement must also be saved and restored when the SQL table is saved and restored. The saving and restoring of the program object is not automated by the database manager. The precautions for self-referencing triggers should be reviewed when restoring SQL tables to a new library.

**Important:** Restoring a collection to an existing library or to a collection that has a different name does not restore the journal, journal receivers, or IDDU dictionary (if one exists). If the collection is restored to a collection with a different name, the catalog views in that collection will only reflect objects in the old collection. The catalog views in QSYS2, however, will appropriately reflect all objects.

10.11 Resolution of unqualified object references

Object qualification in SQL triggers is similar to object qualification in SQL stored procedures, as explained in 5.8, “Implicit object qualification and authorization resolution” on page 117. Look at Example 10-14.

**Example 10-14  Unqualified object references**

```sql
CREATE TRIGGER TRG1
AFTER INSERT ON TBL1
LANGUAGE SQL
BEGIN SQL
DECLARE stmt VARCHAR(256);
INSERT INTO TBL2 VALUES ('WHAT SCHEMA AM I USING?');
SET stmt = 'INSERT INTO TBL2 VALUES (''WHAT SCHEMA AM I USING IN DYNAMIC SQL?''');
PREPARE s1 FROM stmt;
EXECUTE s1;
CALL PROC1();
END;
```
Using the SQL naming convention

The following notes refer to Example 10-14:

1. The implicit qualifier is the authorization identifier of the statement (the user ID executing the statement). If the operation is executed by the USER1 user, the SQL trigger will be created in library USER1 and will be USER1.TGR1.

   If the CREATE TRIGGER operation is executed through an ODBC or JDBC connection and the default library is set, the implicit qualification will be the first library defined in the default library list. For example, if we are using the Run SQL Scripts Utility and the default library is set up as shown in Figure 10-26, the implicit qualification for TRG1 will be TPSTAR.

   ![Figure 10-26   Run SQL Script: Default library list](image)

2. The implicit qualifier is the authorization identifier of the statement at creation time. If the CREATE TRIGGER operation is executed by the USER1 user, it looks at table TBL1 in USER1 schema (collection or library).

   In case of ODBC or JDBC connection, if a default library list is specified, DB2 Universal Database for iSeries looks for TBL1 in the library list.

3. The implicit qualifier is the authorization identifier of the statement at run time, not at creation time. If TRG1 is fired by an operation caused by the USER1 user, it looks at table TBL2 in USER1 schema (collection or library).

   In case of ODBC or JDBC connection, if a default library list is specified, DB2 Universal Database for iSeries looks for TBL2 in the library list.

4. The implicit qualifier is the authorization identifier of the statement at run time, not at creation time. If TRG1 is fired by an operation caused by the USER1 user, it looks at table TBL2 in USER1 schema (collection or library).

5. The implicit qualification is the authorization identifier at run time. In case of ODBC or JDBC connection, if a default library list is specified, it looks for the first procedure with the same signature that can be found in the specified libraries.

   In the example, if we have TRG1 procedures in libraries SAMPLEDB02 and DLEMA, and the user DLEMA executes an operation that fires TRG1, with no default libraries, TRG1 calls DLEMA.PROC1. However, if the default libraries are set as shown in Figure 10-26, SAMPLEDB02.PROC1 is called.

Using the system naming convention

The following notes refers to Example 10-14.

1 and 2. The implicit qualifier is the current library at creation time.

3, 4 and 5. The implicit qualifier is the current library at run time. If TBL2 is not in a current library, the library list will be searched.
When a trigger is created, it is tightened with its qualification to its table. At run time, the fired trigger will not depend on naming convention, library list, or authorization identifier. It depends on the conditions where the trigger was created.

Using the DFTRDBCOL parameter
At creation time, we can specify a default schema by using the DFTRDBCOL parameter in the RUNSQLSTM, CRTSQLxxx, or SET OPTION. When DFTRDBCOL is specified, the implicit qualification is the schema that is specified by the DFTRDBCOL parameter, independently of the naming convention used, except for dynamic SQL.

CREATE TRIGGER TRG1
AFTER INSERT ON TBL1
LANGUAGE SQL
SET OPTION DFTRDBCOL = SCHEMA1
BEGIN
DECLARE stmt VARCHAR(256);
INSERT INTO TBL2 VALUES ('WHAT SCHEMA AM I USING?');
SET stmt = 'INSERT INTO TBL2 VALUES (''WHAT SCHEMA AM I USING IN DYNAMIC SQL?'');
PREPARE s1 FROM stmt;
EXECUTE s1;
CALL PROC1();
END;

In the example, implicit qualifications for 1, 2, 3, and 5 are SCHEMA1, but 4 is qualified at runtime depending on the naming convention used. You can find more details about implicit qualification in SQL Reference, SC41-5612.

10.12 Transaction isolation and recovery

All triggers, when they are activated, set the isolation level to the same isolation level as the triggering application. This isolation level can be overwritten by placing a SET TRANSACTION statement within the trigger body. Careful consideration should be given before deciding to run the application and trigger at a different isolation level. You should especially try to avoid the situation where one part is running under commitment control and the other part is not running under commitment control. This mismatch usage of commitment control can make it difficult to recover the database changes made by both the application and the trigger. When both the application and the trigger are running with isolation other than No Commit, the trigger changes are committed or rolled back whenever the triggering application issues a commit or rollback. Other combinations of commitment control are supported, but they are not recommended.

All triggers, when they are activated, perform a SET TRANSACTION statement so that all of the operations by the trigger are performed with the same isolation level as the application program that caused the trigger to be run. The user may put her own SET TRANSACTION statements in an SQL-control-statement in the SQL-trigger-body of the trigger. If the user places a SET TRANSACTION statement within the SQL-trigger-body of the trigger, then the trigger runs with the isolation level specified in the SET TRANSACTION statement, instead of the isolation level of the application program that caused the trigger to run.

If the application program that caused a trigger to be activated is running with an isolation level other than No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger are run under commitment control and are not committed or rolled back until the application commits its current unit of work. If the application that caused the trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), then the operations of a trigger are written to the database immediately, and cannot be rolled back. If both system triggers defined by the Add Physical File Trigger...
(ADDPFTRG) CL command and SQL triggers defined by the CREATE TRIGGER statement are defined for a table, we recommend that the system triggers perform a SET TRANSACTION statement so that they are run with the same isolation level as the original application that caused the triggers to be activated. We also recommend that the system triggers run in the Activation Group of the calling application. If system triggers run in a separate Activation Group (ACTGRP(*NEW)), then those system triggers will not participate in the unit of the work for the calling application, nor in the unit of work for any SQL triggers.

**Important:** SQL triggers always run in the caller's Activation Group.

If the triggering application is running with commitment control, the operations of an SQL trigger and any cascaded SQL triggers will be captured into a sub-unit of work. If the operations of the trigger and any cascaded triggers are successful, the operations captured in the sub-unit of work are committed or rolled back when the triggering application commits or rolls back its current unit of work. Any system trigger that runs in the same Activation Group as the caller, and performs a SET TRANSACTION to the isolation level of the caller, will also participate in the sub-unit of work. If the triggering application is running without commit control, then the operations of the SQL triggers will also run without commitment control.

If an application that causes a trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), and it issues an INSERT, UPDATE, or DELETE statement that encounters an error during the execution of the statement, SQL triggers will still be activated following the error for that operation. However, a number of changes will already be performed. If the triggering application is running with commitment control, the operations of any triggers that are captured in a sub-unit of work will be rolled back when the first error is encountered, and no additional triggers will be activated for the current INSERT, UPDATE, or DELETE statement.

### 10.12.1 Savepoints

Since the introduction of savepoints in V5R2, a better level of control is given to designers and programmers for database changes caused by trigger programs. Savepoints in triggers and stored procedures are supported in DB2 Universal Database for iSeries and allows us to establish a savepoint anywhere into the trigger and rollback to that savepoint, without having to roll back the complete operation, including changes made by the triggering application. This enhanced our possibilities of designing triggers that keeps their independence with firing applications without renouncing to have commitment control and be able to roll back operations into the trigger.

### 10.12.2 ATOMIC

The ATOMIC behavior has changed between V5R1 and V5R2. In V5R1, the ATOMIC compound statement was based on COMMIT, which is very intrusive to be recommended for a trigger, because a COMMIT in a trigger should end committing changes in the firing application.

Now with V5R2 supporting savepoints, ATOMIC compound statements establish a savepoint at the beginning and release the savepoint at the end. This changes the situation and gives flexibility to this option.

Anyway, independently of the version you are using, if ATOMIC is specified in the SQL-trigger-body of the trigger, and the application program that caused the ATOMIC trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger will not run under commitment control.
10.13 Additional considerations

This section focuses on important topics that are specific to SQL triggers.

10.13.1 Adding columns to a subject table referenced in the triggered action

If a column is added to the subject table AFTER triggers are defined, the following rules apply:

- If the trigger is an UPDATE trigger that was defined without an explicit column list, then an update to the new column will cause the activation of the trigger.
- If the SQL statements in the triggered-action refer to the triggering table, the new column is not accessible to the SQL statements until the trigger is recreated. The OLD_TABLE and NEW_TABLE transition tables will contain the new column, but the column cannot be referenced unless the trigger is recreated.
- If a column is added to any table referenced by the SQL statements in the triggered-action, the new column is not accessible to the SQL statements until the trigger is recreated.

10.13.2 Datetime considerations

If OLD ROW or NEW ROW is specified, the date or time constants and the string representation of dates and times in variables that are used in SQL statements in the triggered-action must have a format of ISO, EUR, JIS, USA, or must match the date and time formats specified when the table was created if it was created using DDS and the CRTPF CL command. If the DDS specifications contain multiple different date or time formats, the trigger cannot be created.

10.13.3 SQL trigger program object

When an SQL trigger is created, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program object is then created using the CRTSQLCI and CRTPGM commands. The SQL options used to create the program are the options that are in effect at the time the CREATE TRIGGER statement is executed. The program is created with ACTGRP(*CALLER). The DB2 Universal Database Query Manager and SQL Development Kit product must be installed on the system when the SQL trigger is created.

10.13.4 Authority of SQL triggers

The trigger program object authorities are:

- When SQL naming is in effect, the trigger program will be created with the public authority of *EXCLUDE and adopt authority from the schema qualifier of the trigger-name if a user profile with that name exists.
- If a user profile for the schema qualifier exists, then the owner of the trigger program will be the user profile for the schema qualifier. Note that the special authorities *ALLOBJ and *SECADM are required to create the trigger program object in the schema qualifier library if a user profile exists that has the same name as the schema qualifier, and the name is different from the authorization ID of the statement.
- If a user profile for the schema qualifier does not exist, then the owner of the trigger program will be the user profile or group user profile of the job executing the SQL CREATE TRIGGER statement.
The group user profile will be the owner of the trigger program object, only if OWNER(*GRPPRF) was specified on the user's profile who is executing the statement. If the owner of the trigger program is a member of a group profile, and if OWNER(*GRPPRF) was specified on the user's profile, the program will run with the adopted authority of the group profile.

- When system naming is in effect, the trigger program will be created with public authority of *EXCLUDE and adopt authority from the user or group user profile of the job executing the SQL CREATE TRIGGER statement.

10.14 Testing and debugging

When you are developing any kind of software, it is important to have a debugging tool. Debugging allows you to detect, diagnose, and eliminate run-time errors in a program. This section shows you debugging alternatives to test SQL Persistent Stored Modules (PSMs).

Remember that when you create an SQL PSM, it is really creating an ILE C program underneath. For this reason, one of the alternatives for debugging SQL triggers is by using the ILE source debugger for testing. V5R2 DB2 Universal Database for iSeries simplified the debug of SQL stored procedure, functions and triggers with the SQL *SOURCE debug view and by the use of the Toolbox for Java iSeries System Debugger. For a complete explanation of the graphical debugger, refer to 3.10.1, “Graphical debugger” on page 38.

**SET OPTION-statement**

This specifies the options that will be used to create the trigger. For example, to create a debuggable trigger, you can include the following statement:

```
SET OPTION DBGVIEW = *STMT
```

**Important:** The options CLOSQLCSR, CNULRQD, DFTRDBCOL, DYNDFTCOL, and NAMING are not allowed in the CREATE TRIGGER statement.

The options DATFMT, DATSEP, TIMFMT, and TIMSEP cannot be used if OLD ROW or NEW ROW is used.

10.14.1 The ILE source debugger

Testing and debugging SQL triggers in a client/server environment can be more difficult than with a traditional iSeries server.

The ILE source debugger is used to detect errors in, and eliminate errors from, program objects and service programs. By using debug commands with any ILE program, you can:

- View the program source or change the debug view.
- Set and remove conditional and unconditional breakpoints.
- Step through a specified number of statements.
- Display or change the value of fields, structures, and arrays.
- Equate a shorthand name with a field, expression, or debug command.

Many debug commands are available for use with the ILE source debugger. These debug commands and their parameters are entered on the debug command line shown at the bottom of the Display Module Source and Evaluate Expression displays. These commands can be entered in uppercase, lowercase, or mixed case.

**Note:** The debug commands on the debug command line are not CL commands.
The debug commands are described in the following list:

- **ATTR**: Permits you to display the attributes of a variable. The attributes are the size and type of the variable.
- **BREAK**: Permits you to enter either an unconditional or conditional breakpoint at a position in the program being tested. Use the BREAK line-number WHEN expression to enter a conditional breakpoint.
- **CLEAR**: Permits you to remove conditional and unconditional breakpoints.
- **DISPLAY**: Allows you to display the names and definitions assigned by using the EQUATE command.
- **EQUATE**: Allows you to assign an expression, variable, or debug command to a name for shorthand use.
- **EVAL**: Allows you to display or change the value of a variable or to display the value of expressions, records, structures, or arrays.
- **QUAL**: Allows you to define the scope of variables that are displayed in subsequent EVAL commands.
- **STEP**: Allows you to run one or more statements of the procedure being debugged.
- **FIND**: Searches forwards or backwards in the module currently displayed for a specified line number or string or text.
- **UP**: Moves the displayed source window toward the beginning of the view for the number of lines entered.
- **DOWN**: Moves the displayed source window toward the end of the view for the number of lines entered.
- **LEFT**: Moves the displayed source window to the left.
- **RIGHT**: Moves the displayed source window to the right by the number of characters entered.
- **TOP**: Positions the view to show the first line.
- **BOTTOM**: Positions the view to show the last line.
- **NEXT**: Positions the view to the next breakpoint in the source currently displayed.
- **PREVIOUS**: Positions the view to the previous breakpoint in the source currently displayed.
- **HELP**: Shows the online help information for the available source debugger commands.

### 10.14.2 Preparing the SQL trigger for debugging

A program or module must have debug data available if you are to debug it. Debug data is created during compilation. On the RUNSQLSTM command, specify whether the generated C module is to contain debug data. This is done by using the DBGVIEW parameter on the RUNSQLSTM command, as shown in Figure 10-27.
Press the Page Down key to view the DBGVIEW parameter, as shown in Figure 10-28.
The Debugging view parameter specifies the type of source debug information to be provided by the SQL compiler. The possible values are:

- **NONE**: The debug view is not generated.
- **STMT**: Allows the compiled module object to be debugged using program statement numbers and symbolic identifiers.
- **LIST**: Generates the listing view for debugging the compiled module object.

You must specify **STMT** or **LIST** if you want debugging data to be saved in the program. After RUNSQLSTM successfully creates the trigger, you are ready to test it.

### 10.14.3 Testing the SQL trigger

When testing the SQL trigger, if it updates, inserts, or deletes records from tables, you can use an Interactive SQL session to verify that it works properly. Another useful tool is to use Query for AS/400 to verify the results.

The easiest way to test the execution of the trigger is to use ILE C code debugging. While debugging and testing your program, ensure that your library list is changed to direct the programs to a test library containing test data so that any existing real data is not affected.

To start a debugging session, use the STRDBG command as shown in Figure 10-29.

```
Start Debug (STRDBG)
Type choices, press Enter.
Program .................... COMMADJ  Name, *NONE
Library ..................... LIBCINTIA Name, *LIBL, *CURLIB
  + for more values
Default program .............. *PGM  Name, *PGM, *NONE
Maximum trace statements .... 200  Number
Trace full ................. *STOPTRC *STOPTRC, *WRAP
Update production files ...... *YES *NO, *YES
OPM source level debug ...... *NO *NO, *YES
Service program .............. *NONE Name, *NONE
Library ..................... *NONE Name, *LIBL, *CURLIB
  + for more values

More...
F3=Exit  F4=Prompt  F5=Refresh  F10=Additional parameters  F12=Cancel
F13=How to use this display  F24=More keys
```

Figure 10-29  Start debug session

It is important to note that when your session is in debug mode, the job log of the session saves a lot of information related to the SQL statements being executed. The application developer can use this information for problem detection and performance tuning.
10.14.4 Testing the SQL trigger in a client/server environment

Testing and debugging SQL triggers in the client/server environment may be a little bit more tricky than in the traditional iSeries environment. This section describes how to test and debug SQL triggers in a distributed environment. Complete the following steps from your Windows desktop:

1. Click **Start -> Programs -> IBM AS400 iSeries Access Express -> ODBC Administration.**

2. In the ODBC Data Source Administrator display (Figure 10-30), click the **Tracing** tab and click the **Start Tracing Now** button.

![Figure 10-30 Starting the ODBC trace](image)

3. Switch to the iSeries server session. To find the QZDASOINIT job serving your client, run the following CL command:

   `WRKOBJLCK OBJ(xxxxxxxx) OBJTYPE(*USRPRF)`

   Here, `xxxxxxx` is the user profile you use to log on the iSeries server.
4. The Work with Object Locks screen (Figure 10-31) is displayed. Type 5 in the Option field of job QZDASOINIT and press Enter.

```
Work with Object Locks
System: ASM23
Object: CINTIA Library: QSYS Type: *USRPRF

Type options, press Enter.
4=End job 5=Work with job 8=Work with job locks

<table>
<thead>
<tr>
<th>Opt</th>
<th>Job</th>
<th>User</th>
<th>Lock</th>
<th>Status</th>
<th>Scope</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QPADEV0003</td>
<td>CINTIA</td>
<td>*SHRRD</td>
<td>HELD</td>
<td>*JOB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*SHRRD</td>
<td>HELD</td>
<td>*JOB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*SHRRD</td>
<td>HELD</td>
<td>*JOB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>QZDASOINIT</td>
<td>QUSER</td>
<td>*SHRRD</td>
<td>HELD</td>
<td>*JOB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*SHRRD</td>
<td>HELD</td>
<td>*JOB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*SHRRD</td>
<td>HELD</td>
<td>*JOB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bottom
```

Figure 10-31 Finding the database server job

5. In the Work with Job display (Figure 10-32), write down the fully qualified job name. Then choose option 10.

```
Work with Job
System: ASM23
Job: QZDASOINIT User: QUSER Number: 001307

Select one of the following:

1. Display job status attributes
2. Display job definition attributes
3. Display job run attributes, if active
4. Work with spooled files
10. Display job log, if active or on job queue
11. Display call stack, if active
12. Work with locks, if active
13. Display library list, if active
14. Display open files, if active
15. Display file overrides, if active
16. Display commitment control status, if active

More...

Selection or command

F3=Exit F4=Prompt F9=Retrieve F12=Cancel
```

Figure 10-32 Work with job
6. In the Display Job Log screen (Figure 10-33), press Enter to return to the command prompt. Then run the STRSRVJOB CL command.

![Display Job Log](image)

```
= Display Job Log =

Job . . : QZDASOINIT  User . . : QUSER  Number . . : 001307

Job 001307/QUSER/QZDASOINIT started on 09/05/01 at 21:50:40 in subsystem QUSRWRK in QSYS. Job entered system on 09/05/01 at 21:50:40.
Printer device PRT01 not found.
Errors on CHGJOB command for job 001307/QUSER/QZDASOINIT.
Printer device PRT01 not found.
User CINTIA from client 9.5.62.72 connected to server.
File QAUGDBGs created in library QTEMP.
Member QAUGDBGs added to file QAUGDBGs in QTEMP.
File QSQLSRC created in library QTEMP.
File QSQLSRC in library QTEMP changed.
Member TEST added to file QSQLSRC in QTEMP.

Press Enter to continue.
```

---

7. The Start Service Job screen is displayed, as shown in Figure 10-34. Start the ILE C Source Debugger for your server job using the CL STRDBG command.

![Start Service Job](image)

```
= Start Service Job (STRSRVJOB) =

Type choices, press Enter.

Job name . . . . . . . . . . . . > QZDASOINIT  Name
User . . . . . . . . . . . . . . > QUSER  Name
Number . . . . . . . . . . . . > 001307  000000-999999

Additional Parameters

Duplicate job option . . . . . *SELECT  *SELECT, *MSG

Press Enter to continue.
```

---
The Start Debug display (Figure 10-35) is displayed.

![Start Debug display](image)

**Figure 10-35   Start Debug**

### 10.15 SQL trigger examples

This section illustrates some examples of special-case triggers. Among the cases, you will find:

- Self-referencing triggers
- Triggers invoking external programs
- SQL triggers invoking a Java stored procedure or a UDF written in Java

#### 10.15.1 Self-referencing triggers

A self-referencing trigger has any reference to the table over which it is defined. For example, you can have an update trigger over table A that, during execution, updates a different row in the same table A.

**Example**

Suppose we have a table called SalesPerson with the following columns:

- SalesPerson_Code
- SalesoftheMonth
- SalesManager_Code (which is another SalesPerson_Code)

We define an Update trigger over the SalesoftheMonth column so that each time we update the SalesoftheMonth column for a sales person, it updates the manager's SalesoftheMonth and so on in a loop.

The trigger checks if SalesManager_Code is not NULL; the only row that has this value is the President of the Company.
Example 10-15  Self referencing SQL trigger

```sql
CREATE TRIGGER LIBCINTIA.TEST
  AFTER UPDATE OF SALESOFTHEMONTH ON SAMPLEDB07.SALESPERSON
  REFERENCING OLD AS OLDROW
  NEW AS NEWROW
  FOR EACH ROW
  MODE DB2ROW
  WHEN (OLDROW.SALESMANAGER_CODE IS NOT NULL)
BEGIN
  DECLARE SALES INTEGER ;
  SET  SALES = NEWROW.SALESOFTHEMONTH - OLDROW.SALESOFTHEMONTH ;
  UPDATE SAMPLEDB07.SALESPERSON
  SET SALESOFTHEMONTH = SALESOFTHEMONTH + SALES
  WHERE SALESPERSON_CODE = NEWROW.SALESMANAGER_CODE ;
END ;
```

10.15.2 SQL trigger invoking external programs

Sometimes it is useful to call external programs from SQL triggers for such operations as:

- Obtaining the user profile that is executing the firing operation
- Performing a non database operation
- Accessing a system API or CL command
- Reusing existing code

You can call external programs using the CALL statement, as explained in 3.4.4, “Calling procedures” on page 29.

In the following example, a bank having applications from multiple vendors needs to propagate changes from a branch table in one application to equivalent branch tables in the others. To do that, this bank selects the application that has the best branch management functionality and developed triggers for insert, update, and delete that propagates changes. A stored procedure applies the changes in the other tables. The insert, update, and delete triggers are as shown in Example 10-16.

Example 10-16  SQL triggers invoking stored procedures

```sql
CREATE TRIGGER SAMPLEDB01.BRANCH_IN
  BEFORE INSERT ON SAMPLEDB01.BRANCH
  REFERENCING NEW ROW AS N
  FOR EACH ROW MODE DB2ROW
CALL SAMPLEDB01.BRNCH('I', N.BRANCH_NO, N.BRANCH_NAME, N.BRANCH_MGR,
  N.ZONE_NO, N REGION_NO, N.STATUS, N.TELEPHONE, N.ADDRESS, N.ENABLERS);

CREATE TRIGGER SAMPLEDB01.BRANCH_DEL
  BEFORE DELETE ON SAMPLEDB01.BRANCH
  REFERENCING OLD AS O
  FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE OLD_BRANCH_NO NUMERIC(4);
  SET OLD_BRANCH_NO = O.BRANCH_NO;
  CALL SAMPLEDB01.BRNCH('D', OLD_BRANCH_NO, '', 0, 0, 0, '', '', '', ');
END;

CREATE TRIGGER SAMPLEDB01.BRANCH_UPD
  BEFORE UPDATE ON SAMPLEDB01.BRANCH
  REFERENCING OLD ROW AS O NEW ROW AS N
  FOR EACH ROW MODE DB2ROW
```
BEGIN
    IF O.BRANCH_NO <> N.BRANCH_NO THEN
        SIGNAL SQLSTATE '38HB1' SET MESSAGE_TEXT  = 'BRANCH NUMBER CHANGE NOT ALLOWED';
    END IF;
    CALL SAMPLER01.BRNCH('U', N.BRANCH_NO, N.BRANCH_NAME, N.BRANCH_MGR,
    N.ZONE_NO, N.REGION_NO, N.STATUS, N.TELEPHONE, N.ADDRESS, N.ENABLEDERS);
END;

Notice that in the delete trigger, the correlation variable O.BRANCH_NO was not passed directly as a parameter to the called stored procedure. This is because OLD correlation variables cannot be modified, so the preprocessor prevents their use in calling stored procedures, even if the parameter has been declared as an output parameter in the stored procedure.

In the case of AFTER triggers, the correlation variables referring to the new row cannot be modified, so they cannot be used directly as parameters in stored procedures invocation.

In these cases, the way to use correlation variables as parameters in stored procedure calling is to assign their value to a local variable.

Calling stored procedures from triggers can be used, among others, for:
- Reusing existing code that cannot be directly declared as triggers.
- Distributing responsibilities among different development groups. In the example above, the responsibility for triggers in that bank belongs to DBAs, but the stored procedure for bringing the changes to branch tables in different applications belong to a specific development group.
- Using the same code in multiple triggers. External triggers allow you to use the same trigger program in multiple trigger declarations, but SQL triggers do not. If multiple triggers do the same job, you can use just one external trigger program or you can create a stored procedure and call it from multiple SQL triggers.

10.15.3 SQL trigger invoking Java stored procedures or UDFs

There is nothing like a Java Trigger in DB2 Universal Database for iSeries. But if you have a particular interest in using Java code inside a trigger, and particularly in an SQL trigger, you can create a Java stored procedure or a Java user-defined function and call that Java code from a trigger.

Note: For more information about user-defined functions, refer to DB2 UDB for AS/400 Object Relational Support, SG24-5409.

For example, suppose that in the SAMPLE database, we need to avoid inserting rows in SALES table when SALES_DATE is a holiday. The code in Example 10-17 corresponds to a Java UDF that tests if a given date is a holiday for a specific country.

Example 10-17  Testing if a given date is a holiday for a specific country

```java
import java.sql.*;
public class Java_UDFs {

    /**
     * Method that verifies if a date in a given country is holiday
     * Creation date: (9/26/2001 11:09:36 AM)
     * @param testedDate java.sql.Date
     * @return integer 0 if it is not a holiday, > 0 if it is a holiday
     */
```
* @param country java.lang.String
*/

public static int isHoliday(Date testedDate, String country) throws SQLException {
    try {
        Connection con = DriverManager.getConnection("jdbc:default:connection");
        String sql = "SELECT COUNT(*) FROM DLEMA.HOLIDAY " +
                        "WHERE EFF_DT = ? " +
                        "AND COUNTRY = ? ";
        PreparedStatement stmt = con.prepareStatement(sql);
        stmt.setDate(1, testedDate);
        stmt.setString(2, country);
        ResultSet rs = stmt.executeQuery();
        int returnValue = -1;
        if (rs.next())
            returnValue = rs.getInt(1);
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
        if (con != null) con.close();
        return returnValue;
    }
    catch(SQLException e) {
        if (e.getSQLState().equals("42704")) {
            // holiday table does not exists
            throw new SQLException("HOLIDAY table does not exists", "38NHD", -438);
        } else {
            throw new SQLException("Unmanaged SQL Error in isHoliday", "38UHD", -438);
        }
    }
    catch(Exception e) {
        // do something to manage the error
        throw new SQLException("Unmanaged Java Error in isHoliday", "38UJV", -438);
    }
}

We created a UDF for this Java method using the following SQL statement:

```
CREATE FUNCTION ISHOLIDAY(
    EFF_DATE DATE,
    COUNTRY VARCHAR(20))
RETURNS INTEGER
EXTERNAL NAME 'Java_UDFs.isHoliday'
PARAMETER STYLE JAVA
LANGUAGE JAVA DETERMINISTIC READS SQL DATA;
```

Notes: The following notes refer to the previous example.

1. It defines a UDF using the isHoliday method of the Java_UDFs class. A Java UDF function is very similar to a Java stored procedure.

2. This is a deterministic UDF, which means that if it receives equal parameters, it is going to return exactly the same value.

Then we create the following SQL trigger, in which we use the isHoliday Java UDF to prevent the insertion of rows when SALES_DATE is a holiday:

```
CREATE TRIGGER HOLIDAYCHECK
BEFORE INSERT ON SALES
REFERENCING NEW AS newRow
FOR EACH ROW MODE DB2ROW
```
BEGIN
  DECLARE holiday INTEGER;
  SET holiday = ISHOLIDAY(newRow.sales_date, 'US');
  IF holiday >0 THEN
    SIGNAL SQLSTATE VALUE '38HLY';
  END IF;
END;

Notes: The following notes refer to the previous example.
1. Function ISHOLIDAY is the Java UDF that determines if a given date is holiday for a specific country.
2. If it is a holiday, the trigger signals a user-defined SQLSTATE '38HLY'.

Now, if we try to insert a sales record for the first of January, we receive the error message shown in Figure 10-36.

SQL trigger HOLIDAYCHECK in SAMPLEDB02 failed with SQLCODE -438 SQLSTATE 38HLY. Cause . . . . . : An error has occurred in a triggered SQL statement in trigger HOLIDAYCHECK in library SAMPLEDB02. The SQLCODE is -438, the SQLSTATE is 38HLY, and the message is . Recovery . . . : Refer to the joblog for more information regarding the detected error. Correct the error and try the request again.

Figure 10-36   Error message

Notes: The following notes refer to Figure 10-36:
1. SQLCODE -723 corresponds to SQLSTATE 09000: An error occurred in a Trigger, preventing the completion of the operation.
2. The trigger causing the error is identified. In this case, it corresponds to our trigger.
3. The error message signaled by the trigger is also specified. Note that in this example, the error message signaled by the trigger is 38HLY, our error code for trying to insert a row with a holiday sales date.

For more information about a consistent error handling strategy, see 8.1, “Database error reporting strategy” on page 222.

10.15.4 Accessing a Global Temporary Table from an SQL trigger

SQL triggers use the same procedural language as an SQL procedure and therefore have the same limitation of not being able to directly process a result set. While SQL triggers can also use this Global Temporary Table technique, extra setup step is required. When an SQL trigger is created, the SQL standard requires that all of the objects referenced by the trigger must already exist. In our examples so far, if the FreeEmp_Results Global Temporary Table does not exist in the same job or connection that is attempting to create the trigger, the trigger creation fails.

To solve the issue of this trigger enforcement, you can copy and execute one of the DECLARE GLOBAL TEMPORARY TABLE statements from the Get_Free_Employees stored procedure of Example 10-18. Alternatively you can call the stored procedure to make available the required FreeEmp_Results temporary table before executing the Create Trigger statement. Example 10-18 shows a coding example.
CREATE TRIGGER Free_Employee_Check
BEFORE INSERT ON project FOR EACH ROW
BEGIN
  DECLARE free_emp_countt INT;
  CALL Get_Free_Employees('ALL');
  SET free_emp_count = (SELECT COUNT(*) FROM session.freeemp_results);
  IF (free_emp_count = 0) THEN
    SIGNAL SQLSTATE VALUE '38001' SET MESSAGE_TEXT = 'Error-No Employees Available';
  END IF;
END;

We now hope that you understand how SQL Global Temporary Tables can be a powerful tool in your programming toolbox when sets of data need to be shared across stored procedures and program calls.

10.15.5 Instead Of Trigger

Instead Of Trigger was first introduced in March 2005 on V5R3. IBM created the Instead Of Trigger to enhance the behavior for Insert, Update, and Delete operations against SQL views. An Instead Of Trigger is an SQL trigger that can be used to change the semantics of an INSERT, UPDATE, and DELETE operation against a view. An Instead Of Trigger allows certain views that due to their nature are read only, to perform INSERT, UPDATE and DELETE operations on them.

A good example of using Instead Of Trigger is on a table where columns are being encrypted. An insert operation on a view would not be possible and Instead Of Trigger can be use to address this. Look at the following examples:

► Example 1:
  CREATE VIEW my_logins(system, login, paswd) AS
  SELECT system, login, decrypt_char(passwd) FROM regusers WHERE userid=USER

  The VIEW in this example cannot be used to INSERT, UPDATE, or DELETE on the underlying table or tables due to the column passwd is being encrypted. By creating an Instead Of Trigger as in the following samples, an INSERT/UPDATE is allowed for the view.

► Example 2 - INSERT:
  CREATE TRIGGER insert_my_logins INSTEAD OF INSERT ON my_logins
  REFERENCING NEW AS n FOR EACH ROW MODE DB2SQL
  INSERT INTO regusers VALUES (USER, n.system, n.login, Encrypt(n.passwd))

► Example 3 - UPDATE:
  CREATE TRIGGER update_my_logins INSTEAD OF UPDATE ON my_logins
  REFERENCING OLD AS o NEW AS n FOR EACH ROW MODE DB2SQL
  UPDATE regusers SET system=n.system, login=n.login, passwd=Encrypt(n.passwd) WHERE system=o.system AND login=o.login AND userid=USER

Instead Of Triggers can only be defined over an SQL view to perform an INSERT, UPDATE, or DELETE request on behalf of the view. When an Instead Of Trigger is fired, the triggering SQL statement against the view is replaced by the trigger logic, which performs the operation on behalf of the view. A DB2 Instead Of Trigger is activated after the triggering statement is issued to the base view. Only one Instead Of Trigger of can be defined for each triggering action (INSERT, UPDATE, and DELETE) on a view.
External triggers

External triggers represent one of the most powerful features of DB2 Universal Database for iSeries. This chapter describes several technical aspects of external triggers. It also provides examples and guidelines to show how you can take advantage of external triggers in your application environment.

This chapter discusses:

- Defining a trigger
- Interfacing triggers and applications
- Considerations for commitment control
- Designing trigger programs
- Advanced trigger features
- Application and design considerations
- Recommendations
11.1 Defining a trigger

On the iSeries server, a trigger program can be developed using any supported high-level language compiler. You can include SQL statements or any other high-level language. You can also code a trigger using the CL language. See Figure 11-1.

After the trigger is developed, it can be associated with a physical file/table. The definition has a file-level scope: If you define a trigger on a multi-member physical file, such as a yearly sales file having a member for each month, the trigger is activated whenever data is modified in any member.

When adding a trigger program to a database table, you need to specify:

- **Trigger event**: This is the I/O operation that activates the trigger:
  - Insert
  - Update
  - Delete
  - Read (this applies to external triggers only)

- **Trigger time**: Determines whether the trigger is activated before or after the trigger event takes place. We describe later in detail how this parameter influences the behavior of DB2 Universal Database for iSeries. See 12.5, “Constraints and triggers: Ordering the actions” on page 440.
  - Before operation: Before an update, an insert, or delete operation
  - After operation: After an update, an insert, or delete operation

- **Trigger program**: The program that is activated for this type of I/O operation.
Replace trigger:
– Option (*YES): The trigger program is replaced if there is any other trigger with the same specification.
– Option (*NO): The trigger program is added if there are no other triggers with the same specification.

Allow repeated change: Specifies whether repeated changes to a record within a trigger are allowed. This parameter takes effect only when it runs under commitment control. The options are:
– *NO: Repeated changes to a record within a trigger are not allowed.
– *YES: Repeated changes to a record within a trigger are allowed.

When ALWREPCHG(*YES) is specified for the BEFORE INSERT and UPDATE triggers, the record image can be changed in the trigger buffer.

Trigger condition: This parameter is relevant to UPDATE triggers only. The options are:
– *CHANGE: The trigger runs only if the update operation has actually changed the data. If the update operation leaves the record as it was, the trigger is not activated.
– *ALWAYS: The trigger is always called, even if no field in the record has been changed.

Trigger: This parameter is relevant to all triggers. The options are:
– *GEN: The system generates a trigger name.
– trigger-name: Specify the name of the trigger. The trigger name must be unique to the library. The trigger name is used to distinguish triggers with the same time and event values. You can specify a maximum of 128 characters without delimiters or 258 characters with quotation mark (") delimiters. The case is preserved when lowercase characters are specified.

Triggers can be added to database tables by using a CL command or by using Operations Navigator.

11.1.1 ADDPFTRG

The Add Physical File Trigger (ADDPFTRG) command associates a trigger program with a physical file. When this association is established, DB2 Universal Database for iSeries calls the trigger program when a change operation is performed against the physical file, a member of the physical file, and any logical file created over the physical file or views created by SQL. See Figure 11-2 and Figure 11-3.
When you add the trigger to the physical file, the file description is updated to reflect that a trigger has been associated with the file. You can recompile, restore, rename, copy, and delete the program, and the file description is not affected. For example, when you update the trigger program, you do not need to remove the trigger and add it again to the physical file. You can take advantage of this flexibility in case you need to change your business rules. Simply recompile the trigger program. You do not have to modify any applications or change data in your database. All applications accessing this database file immediately comply with the new rules.

However, if you specify *LIBL when you add the trigger, the actual library name is resolved and stored in the file description (Figure 11-4).
In this example, the file description reports the actual library where the trigger is currently stored. By using DSPFD TYPE(*TRG), notice that the trigger library is explicitly reported (Figure 11-5).
You can define up to 300 trigger programs for the same database table: *BEFORE and *AFTER insert, delete, update, or read operations.

11.1.2 Using iSeries Navigator to add an external trigger

Using iSeries Navigator, you can define system (external) triggers and SQL triggers. In addition, you can enable, disable, or delete a trigger.

To add a trigger, follow these steps:
1. In the iSeries Navigator window, expand your server->Database.
2. Choose the database you are working with and expand its libraries.
3. Click the library that contains the table to which you want to add the trigger.
4. Right-click the table to which you want to add the trigger and select Properties.
5. In the Table Properties window, click the Triggers tab.
6. Select Add external trigger to add an external (system) trigger.

The following iSeries Navigator windows show the additional steps for adding a system (external) trigger to a database. Figure 11-6 shows the General tab information required for adding the trigger. Notice the similarity with the data required on the ADDPFTRG command (Figure 11-2 on page 360 and Figure 11-3 on page 360).

![Add External Triggers for Table: SAMPLEDB01.SALES](image)

**Figure 11-6 Add External Triggers: General tab**
Figure 11-7 shows the information required in the Events tab for a trigger. Notice that a trigger name and a library name for the trigger are additional information needed for a trigger. If this is not provided, the system generates a trigger name and stores it in the same library as the library of the file to which the trigger is being added.
Figure 11-8 shows that the INSERT AFTER trigger has been added to the list of triggers for table SALES in library SAMPLEDB01. It also shows a list of all the triggers for the table.

11.2 Trigger program structure

When a trigger is activated, the system automatically provides the program with the following parameter list:

- **Trigger buffer**: The trigger buffer has two logical parts:
  - **Static area**:
    - A trigger template that contains the physical file name, member name, trigger event, trigger time, commit lock level, and CCSID of the current change record and relative record number
    - Offsets and lengths of the record areas and null byte maps
  - **Dynamic area**:
    - Areas for the old record and old null byte map, new record, and new null byte map

This area occupies (in decimal) offset 0 through 95.
▶ Trigger buffer length: The length of the trigger buffer provided by DB2 Universal Database for iSeries

By defining these parameters in your trigger programs, you can take the appropriate actions based on the kind of data change that has occurred and the characteristics of the job that fired the trigger. Our code samples show how you can use the information passed through the trigger parameter list. See 11.4, “Designing trigger programs” on page 380. Table 11-1 describes the trigger buffer structure.

**Note:** The way we defined the trigger buffer for COBOL and RPG implies that, if you change the record length of the associated database file, the trigger program must be modified to run correctly. Alternatively, you can perform a move operation to bring the record images into your work field variables. There is another technique called soft coding the trigger buffer. In this case, when the structure of the file changes, you only have to recompile the trigger program to access the new layout. We show this technique in 11.4.4, “Soft coding the trigger buffer example” on page 416.

<table>
<thead>
<tr>
<th>Decimal Offset</th>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Physical file Name</td>
<td>char(10)</td>
<td>The physical file being changed.</td>
</tr>
<tr>
<td>10</td>
<td>Physical file library name</td>
<td>char(10)</td>
<td>The library in which the physical file resides.</td>
</tr>
<tr>
<td>20</td>
<td>Physical file member name</td>
<td>char(10)</td>
<td>The name of the physical file member.</td>
</tr>
<tr>
<td>30</td>
<td>Trigger Event</td>
<td>char(1)</td>
<td>The event that caused the trigger program to be called; the possible values can be “1” (Insert), “2” (Delete), “3” (Update), “4” (Read).</td>
</tr>
<tr>
<td>31</td>
<td>Trigger Time</td>
<td>char(1)</td>
<td>Can be “1” (After) or “2” (Before).</td>
</tr>
<tr>
<td>32</td>
<td>Commit level</td>
<td>char(1)</td>
<td>Reports the commit lock level of the interface that activated the trigger “0” (NONE), “1” (CHG), “2” (CS), “3” (ALL).</td>
</tr>
<tr>
<td>33</td>
<td>Reserved</td>
<td>char(3)</td>
<td>Reserved.</td>
</tr>
<tr>
<td>36</td>
<td>CCSID of data</td>
<td>binary(4)</td>
<td>The CCSID of the data in the new or the original records; the data is converted to the job CCSID by the database.</td>
</tr>
<tr>
<td>40</td>
<td>Relative record number</td>
<td>binary(4)</td>
<td>Relative Record Number of the record to be updated or deleted (*BEFORE triggers) or the relative record number of the record that was inserted, updated, deleted, or read (*AFTER triggers).</td>
</tr>
<tr>
<td>44</td>
<td>Reserved</td>
<td>char(4)</td>
<td>Reserved.</td>
</tr>
<tr>
<td>48</td>
<td>Original Record offset</td>
<td>binary(4)</td>
<td>The location of the original record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the original value of the record does not apply to the operation; for example, an insert operation.</td>
</tr>
<tr>
<td>52</td>
<td>Old record length</td>
<td>binary(4)</td>
<td>The maximum length is 32766 bytes.</td>
</tr>
<tr>
<td>56</td>
<td>Old record null map offset</td>
<td>binary(4)</td>
<td>The location of the null byte map of the original record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the original value of the record does not apply to the change operation, for example, an insert operation.</td>
</tr>
<tr>
<td>60</td>
<td>Old record null map length</td>
<td>binary(4)</td>
<td>The length is equal to the number of fields in the physical file.</td>
</tr>
</tbody>
</table>
### Trigger Buffer Definitions

<table>
<thead>
<tr>
<th>Decimal Offset</th>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>New record offset</td>
<td>binary(4)</td>
<td>The location of the new record. The offset value is from the beginning of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the trigger buffer. This field is not applicable if the new value of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>record does not apply to the change operation, for example, a delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operation.</td>
</tr>
<tr>
<td>68</td>
<td>New record length</td>
<td>binary(4)</td>
<td>The maximum length is 32766 bytes.</td>
</tr>
<tr>
<td>72</td>
<td>New record null map offset</td>
<td>binary(4)</td>
<td>The location of the null byte map of the new record. The offset value is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>from the beginning of the trigger buffer. This field is not applicable if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the new value of the record does not apply to the change operation, for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>example, a delete operation.</td>
</tr>
<tr>
<td>76</td>
<td>New record null map length</td>
<td>binary(4)</td>
<td>The length is equal to the number of fields in the physical file.</td>
</tr>
<tr>
<td>80</td>
<td>Reserved</td>
<td>char(16)</td>
<td>Reserved.</td>
</tr>
<tr>
<td>*</td>
<td>Original record</td>
<td>char(*)</td>
<td>A copy of the original physical record before being updated, deleted, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read. The original record applies only to update, delete, and read</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operations.</td>
</tr>
<tr>
<td>*</td>
<td>Original record null byte map</td>
<td>char(*)</td>
<td>This structure contains the NULL value information for each field of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>original record. Each byte represents one field. The possible values for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>each byte are: “0” (Not NULL), “1” (NULL).</td>
</tr>
<tr>
<td>*</td>
<td>New record</td>
<td>char(*)</td>
<td>A copy of the record that is being inserted or updated in a physical file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>as a result of the change operation. The new record only applies to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>insert or update operations.</td>
</tr>
<tr>
<td>*</td>
<td>New record null byte map</td>
<td>char(*)</td>
<td>This structure contains the NULL value information for each field of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new record. Each byte represents one field. The possible values for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>each byte are: “0” (Not NULL), “1” (NULL).</td>
</tr>
</tbody>
</table>

**Important**: When support for the LOB data types (BLOB, CLOB, and DBCLOB) was added in V4R4, it came with a minor restriction. That is triggers could not be defined over a table with LOB columns. In V5R1, this restriction was lifted.

One interesting side-effect of this enhancement is that it changed the size of the trigger buffer passed as input to all external trigger programs due to system infrastructure changes. Trigger programs that either presumed the overall trigger buffer length will not change (for example, placing the entire entry in a permanent/unchanging data queue), or hard-coded their usage of the trigger buffer parameters instead of correctly using the offsets and lengths passed in the trigger buffer, will be impacted. However, for trigger programs that were properly coded to allow for a change in the trigger buffer length (second parameter) and access the trigger buffer data (for example, before and after images and null byte maps), using the offsets and lengths should continue to function as expected when they are executed on a V5R1 system.

The following sections describe the trigger buffer definitions for several programming languages (RPG, COBOL, and C).
### 11.2.1 Trigger buffer for RPG

Figure 11-9 shows an example of how you can define the trigger buffer.

```sql
*===================================================================*
* Definition of the structure to be passed into the trigger program = buffer *
*===================================================================*

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPARM1</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>I</td>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>I</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>I</td>
<td>31</td>
<td>40</td>
</tr>
<tr>
<td>I</td>
<td>32</td>
<td>50</td>
</tr>
<tr>
<td>I</td>
<td>33</td>
<td>60</td>
</tr>
<tr>
<td>I</td>
<td>34</td>
<td>70</td>
</tr>
<tr>
<td>I</td>
<td>35</td>
<td>80</td>
</tr>
<tr>
<td>I</td>
<td>36</td>
<td>90</td>
</tr>
</tbody>
</table>

*===================================================================*
* LENG = buffer length *
*===================================================================*

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPARM2</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>40</td>
</tr>
</tbody>
</table>
```

**Notes:** The following notes refer to Figure 11-9:
- Trigger event
- Trigger commit level
- Relative record number
- Old record length
- New record length
- Old record image
- Old record null map
- New record image
- New record null map
11.2.2 Trigger buffer for COBOL

Figure 11-10 shows how you can define the trigger buffer in a COBOL program.

Listing: Trigger buffer for COBOL

```cobol
*===================================================================*
* PARM 1 = Trigger buffer                                       *
*===================================================================*
LINKAGE SECTION.
 01 PARM-1.
    03 FILE-NAME       PIC X(10).
    03 LIB-NAME        PIC X(10).
    03 MEM-NAME        PIC X(10).
    03 TRG-EVENT       PIC X. 1
    03 TRG-TIME        PIC X.
    03 CMT-LCK-LVL     PIC X. 2
    03 FILLER          PIC X(3).
    03 DATA-AREA-CCSID PIC 9(8) BINARY.
    03 RRN             PIC (8) BINARY 3
    03 FILLER          PIC X(4).
    03 DATA-OFFSET.
      05 OLD-REC-OFF    PIC 9(8) BINARY.
      05 OLD-REC-LEN    PIC 9(8) BINARY 4
      05 OLD-REC-NULL-MAP PIC 9(8) BINARY.
      05 OLD-REC-NULL-LEN PIC 9(8) BINARY.
      05 NEW-REC-OFF    PIC 9(8) BINARY.
      05 NEW-REC-LEN    PIC 9(8) BINARY 5
      05 NEW-REC-NULL-MAP PIC 9(8) BINARY.
      05 NEW-REC-NULL-LEN PIC 9(8) BINARY.
      05 FILLER         PIC X(16).
    03 RECORD-JUNK.
      05 OLD-RECORD      PIC X(112). 6
      05 OLD-NULL-MAP    PIC X(9). 7
      05 NEW-RECORD      PIC X(112). 8
      05 NEW-NULL-MAP    PIC X(9). 9

*===================================================================*
* PARM 2 = Trigger length                                       *
*===================================================================*
01 PARM-2.
  03 TRGBUF-LEN       PIC X(2).
```

Figure 11-10  Trigger buffer for a COBOL program

Notes: The following notes refer to Figure 11-10:
- Trigger event
- Trigger commit level
- Relative record number
- Old record length
- New record length
- Old record image
- Old record null map
- New record image
- New record null map
11.2.3 Trigger buffer for C

In the QSYSINC library, there is an include file for most of the system APIs that can be used in a C program. The file name is H. If the QSYSINC library is not available in your system, install the System Openness Includes option of OS/400. Figure 11-11 shows the trigger buffer definition for a C program taken from the TRGBUF member.

```c
typedef _Packed struct Qdb_Trigger_Buffer {
    char  file_name[10];
    char  library_name[10];
    char  member_name[10];
    char  trigger_event[1];
    char  trigger_time[1];
    char  commit_lock_level[1];
    char  reserved_1[3];
    int   data_area_ccsid;
    int   Current_Rrn;
    char  reserved_2[4];
    int   old_record_offset;
    int   old_record_len;
    int   old_record_null_byte_map;
    int   old_record_null_byte_map_len;
    int   new_record_offset;
    int   new_record_len;
    int   new_record_null_byte_map;
    int   new_record_null_byte_map_len;
} Qdb_Trigger_Buffer_t;
```

*Figure 11-11  Trigger buffer for a C program*
While using C, we are not allowed to define explicitly the old record image, the old null map record image, the new record image, and the new null map record image that we must include in RPG and COBOL programs. This is because of the different memory allocation techniques of the C language. If you use C, set a pointer to the record images using the offsets. You can access the contents of these areas.

11.2.4 Using the trigger buffer

The following discussion refers to the most important fields in the trigger buffer, as marked in the previous examples:

- **Trigger Event**: This field gives you the possibility of determining the event that called the trigger. This information is particularly valuable when a trigger is defined for different events. You may want to identify which record image to use, depending on the event that has activated the trigger. Remember that the system always initializes the offset fields, even if one of them may address meaningless data. Table 11-2 shows the record images that you actually receive, depending on the event.

<table>
<thead>
<tr>
<th>Trigger event</th>
<th>Images</th>
</tr>
</thead>
<tbody>
<tr>
<td>*INSERT</td>
<td>NEW RECORD</td>
</tr>
<tr>
<td>*UPDATE</td>
<td>NEW/OLD RECORD</td>
</tr>
<tr>
<td>*DELETE</td>
<td>OLD RECORD</td>
</tr>
</tbody>
</table>

- **Trigger CMTLVL**: This is the commit lock level of the application that caused the trigger to run. Generally speaking, we do not know whether the interface that activates the trigger program is running under commitment control. This parameter can be used in triggers when you want to set the same commit lock level as the transaction that fired the trigger.

There are several ways to set an isolation level for triggers, depending on the language that you are using. For SQL triggers, you can use the SET TRANSACTION SQL statement. Remember that if you have both SQL and native data access in your program, SET TRANSACTION only affects SQL statements. Access to data through native interfaces is not affected by SET TRANSACTION. For complete information about commitment control and commit lock levels, see Backup and Recovery, SC41-5304.

For native trigger programs, different considerations apply. Using C, you can dynamically open a file with or without commitment control. ILE RPG provides a dynamic commitment definition for physical files; you can associate an RPG variable to the COMMIT keyword on the F specification. Most OPM languages do not provide a way to dynamically define commitment control for files. There are several ways to circumvent this limitation. For example, you may define the database files in your program twice, once without and once with commitment control.
Open the correct file definition according to the value retrieved in the trigger buffer, for example, using COBOL/400®:

```
IDENTIFICATION DIVISION.

FILE-CONTROL.
  SELECT FILDEFA ASSIGN TO DATABASE-FILEX
  ORGANIZATION IS INDEXED
  ACCESS IS RANDOM
  RECORD KEY IS EXTERNALLY-DESCRIBED-KEY.
  SELECT FILDEFB ASSIGN TO DATABASE-FILEX
  ORGANIZATION IS INDEXED
  ACCESS IS RANDOM
  RECORD KEY IS EXTERNALLY-DESCRIBED-KEY.

I-O-CONTROL.
  COMMITMENT CONTROL FOR FILDEFA.

PROCEDURE DIVISION.

IF CMT-LCK-LVL = '0' THEN
  OPEN I-O FILDEFB
ELSE
  OPEN I-O FILDEFA.
```

You can also see the example provided in “Invoice trigger example in ILE RPG” on page 406, where an ILE RPG program handling dynamic commitment control is shown. For a full discussion of the commitment control options of the various programming languages, consult the specific language user's guide and reference.

Example 11-1 shows part of an RPG trigger program with embedded SQL statements showing how to set the correct commit lock level.

```
Example 11-1  RPG trigger program to set correct commit lock level

*==================================================================*
* Extracts the commitment control level of the invoking         *
* application and set the same isolation level for the         *
* trigger program                                              *
*==================================================================*
*
SELECT
  C     CMTLCK        WHENEQ    '0'
C/EXEC SQL
  C+ SET TRANSACTION ISOLATION LEVEL NO COMMIT
C/END-EXEC
C     CMTLCK        WHENEQ    '1'
C/EXEC SQL
  C+ SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED, READ WRITE
C/END-EXEC
*
C     CMTLCK        WHENEQ    '2'
C/EXEC SQL
  C+ SET TRANSACTION ISOLATION LEVEL READ COMMITTED, READ WRITE
C/END-EXEC
C     CMTLCK        WHENEQ    '3'
C/EXEC SQL
  C+ SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
C/END-EXEC
* 
```
For detailed information about the isolation level and SET TRANSACTION, see SQL Programming Guide, SC41-5611.

- **Trigger relative record number**: This is a relative physical address of a row in the database table. You can use it to quickly retrieve a specific row from a table:
  
  ```sql
  select * from ordentlib.orderhdr where rrn(ordentlib.orderhdr) = :relnum
  ```

- **Trigger old record image**: When a trigger is activated by an update or a delete operation, the old record image is found in this parameter field. In COBOL and RPG, remember to define a storage area with the same length as the database record length.

- **Trigger old null record map**: This is the map of the null record fields of your database file. This character array has the same length as the number of fields in the database file associated to the trigger program. DB2 Universal Database for iSeries can set each character to 1 (NULL field) or 0 (not NULL).

- **Trigger new record image**: When a trigger is activated by an update or an insert operation, the new record image is found in this parameter field. In COBOL and RPG, remember to define a storage area with the same length as the database record length.

- **Trigger new null record map**: This is the map of the null record fields of your database file. This character array has the same length as the number of fields in the database file associated with the trigger program. DB2 Universal Database for iSeries can set each character to 1 (NULL field) or 0 (not NULL).

If you are using variable length fields, the length of the record images provided by DB2 Universal Database for iSeries is the maximum length allowed for the database records. Variable length character fields are padded with blanks, and a two-byte binary field with the actual data length is added in front of every VARCHAR field.

### 11.3 Trigger feedback to application programs

When implementing your trigger program, you have to consider that triggers cannot pass parameters back directly because trigger programs are activated by the database manager and are given an input-only parameter list. If a failure occurs while the trigger program is running, an appropriate escape message must be signalled before the trigger terminates. The message can be the original message that is signalled by the system or a user-defined message retrieved from a message file by the trigger program.

If no error message is signalled to the calling program after a trigger has failed, the database manager assumes that the trigger completed successfully and the operation that activated the trigger is completed as well.

We differentiate between the two cases here:

- **System generated error**, such as a failure encountered accessing a locked record

  In this case, the system generates an exception that looks for an exception handler in the trigger. If none is found, the exception traverses the invocation stack in reverse order searching for an appropriate exception handler. If an exception handler does not handle the exception, the exception is processed by the system database module that is performing the I/O operation that fired the trigger. The I/O operation will fail.

- **Failures detected by the trigger program**

  This situation is common in data validity checking. For example, consider the case of an insert trigger performing checking on the records being inserted. When a record is inserted, the OS/400 module QDBPUT is invoked (Figure 11-16 on page 375). Notice that the trigger is displayed in the invocation stack after the OS/400 module QDBPUT.
If the trigger determines that invalid data is being inserted, the insert operation has to be rejected. We can achieve this by sending an escape message to the call stack entry where QDBPUT is running. For this purpose, use the QMHSNDPM API to signal an escape message to QDBPUT. You may choose to send a user-defined message back to the application that fired the trigger.

In both cases, the I/O operation fails, and the application receives an error code. As a result of the escape message signalled by triggers, depending on the language you use, there are different return codes:

- **SQL application:**
  
  SQLCODE = - 443

  This corresponds to the message:

  SQL0443, “Trigger program or external procedure detected an error”.

- **COBOL language:**

  File Status = 90

- **RPG language:**

  The indicator is turned on, and you receive an RPG1299 message.

- **CL language:**

  Message CPF502B, “Error occurred in trigger program” is received.

- **C application:**

  The **errno** variable is set to EIORECERR.

The I/O feedback area is also updated. The field reporting the exception identifier is set to “CPF502B”.

User messages sent by the triggers through the QMHSNDPM API and the CPF502B message are always found in the job log. If the application that is activating the trigger happens to run in an interactive job, the trigger may send a message to the display. We show an example of this technique in “Audit trail trigger example in COBOL SQL (OPM)” on page 387.

**Note:** If a trigger error occurs in an SQL application, a message key of the original error is stored in the SQLERRD(4) field of the SQLCA communication area. The QMHRTVPM API can be used to return the message description for this message key.

If you provide exception handling routines in your trigger programs, remember that, once an exception has been handled, the trigger will end normally. If you need to reject the change operation that fired the trigger, you have to signal an escape message from your exception handler to the appropriate call stack entry.

It is interesting to see how a trigger failure is reported back to the most common interfaces for data access on the iSeries server, such as Data File Utility (DFU) and Interactive SQL. Notice that the actual message sent to these interfaces is generic in both cases. The user message sent by the trigger can be found in the job log.
An Interactive SQL insert operation failing after a trigger signalled an escape message is shown in Figure 11-12.

```
> UPDATE ORDENTL/ORDERHDR SET CUSTOMER_NUMBER = '00005' WHERE
  ORDER_NUMBER = '12312'
  Trigger program or external procedure detected an error.
```

*Figure 11-12  Escape message signalled by a trigger*

The SQL interface sends the appropriate generic SQL message (SQL0443), Trigger program or external procedure detected an error.

The message is sent to the job log by the trigger. In this case, a user-defined message, Salesperson not allowed to deal with the customer, was previously sent by the QMHSNDPM API (Figure 11-13).

```
3 > strsql
  Run in debug mode for performance information.
  Salesperson not allowed to deal with customer
  Error occurred in trigger program.
  Error occurred in trigger program.
  Trigger program or external procedure detected an error.
```

*Figure 11-13  User-defined message signalled by a trigger*

Similarly, if you use DFU, the operation that you are performing fails, and the generic CPF502B is reported to the interface (Figure 11-14).

```
WORK WITH DATA IN A FILE                       Mode . . . . :   ENTRY
  Format . . . . :   ORDERHDR                    File . . . . :   ORDERHDR
ORDER_NUMBER:    00001
CUSTOMER_NUMBER: 00005
ORDER_DATE:      2001-01-01
ORDER_DELIVERY:  2001-01-01
ORDER_TOTAL:           34000
SALESREP_NUMBER: ITSCID24
```

*Figure 11-14  DFU session*
After you press the Enter key on this display, you see the exception shown in Figure 11-15.

![End Data Entry](image)

It is worthwhile to go into more detail to show how the process of signalling a message to the database interface works. The example in Figure 11-16 relates to our application. We are reproducing the invocation stack of the application right after the trigger has been activated.

![Job Call Stack](image)

**Notes:** The following notes refer to Figure 11-16:

1. Application program Order Header Entry inserts a record.
2. This is the OS/400 module for the insert operation.
3. This is the trigger program.
In the invocation stack, each entry is assigned a call message queue as soon as the call stack entry is displayed in the job stack. The call message queue is destroyed when the procedure or program leaves the invocation stack. Some of the messages on the call message queue are also recorded in the job log. As we previously mentioned, when the trigger is called, the application that caused its invocation waits until the trigger comes to an end. The trigger becomes part of the application flow and is shown in the invocation stack of the job as though it had been called by the application itself.

The statements in Figure 11-17 from a COBOL program show which parameters to pass to the QMHSNDPM API to send a message to the DB module performing the I/O operation.

The parameters indicated by 4 are initialized with the message identification and the message file associated with this message identification. You may use a system message, or you may create your own message file to signal a message through trigger programs.

If you create your own message file, make sure your messages are created by setting DMPLST = *NONE. Otherwise, your applications generate dump spooled files when they receive these messages.

The parameter indicated by 5 identifies the program message queue entry. If you use the value * (asterisk), you address the message queue of the procedure currently in execution. We specified “1” for the program stack counter indicated by 6 so that the message is sent to the previous call stack entry, which is QDBPUT. See System API Programming, SC41-5800, for a description of the usage of this API.

You may need to call your HLL trigger from a CL program. For example, you need to perform an OVRDBF ..... SHARE(*YES) before the actual trigger logic is executed. The CL code is shown in Figure 11-18.
After the trigger is activated, the job call stack is displayed as shown in Figure 11-19.

![Job Call Stack](image)

**Figure 11-19 Invocation stack with a CL trigger**

You need to specify the correct call stack entry name to the QMHSNDPM API that is initializing the parameter, indicated by 5 in Figure 11-17, with the CL program name CLPGM.

Similarly, if you are writing an ILE trigger, every ILE compiler introduces the Program Entry Procedure (PEP) in front of the user main program (Figure 11-20).

![Job Call Stack](image)

**Figure 11-20 Invocation stack with an ILE trigger**

In Figure 11-20, you can see the call stack entry after an ILE RPG trigger has been activated. The call stack entry shown in bold is the PEP of this specific ILE RPG program. The PEP message queue name for an RPG program has the format:

_**QRNP_PEP_**  ...  **Library**

In our example, the full name is _**QRNP_PEP_T4249IADT**_. Also, see the example in “Audit trail trigger in ILE RPG - T4249IADT” on page 392. If you use ILE C, the PEP name is always _**C_pep**_, and is case sensitive (see “Audit trail trigger in ILE C - T4249CCAT” on page 421).
11.3.1 Commitment control and triggers

To ensure the best level of data consistency, use commitment control in your applications. If your database design includes triggers, be aware of the implications of using commitment control for the resources accessed by the trigger programs. To avoid data integrity potential exposures, triggers and applications should share the same commitment definition. In this case, all the changes performed by triggers are committed or rolled back by the application itself. The safest way to ensure that this happens is by compiling your triggers with ACTGRP(*CALLER). Triggers and applications should also share the same lock level. See 11.2, “Trigger program structure” on page 364, for more information about handling the commit lock level in trigger programs.

If triggers run in a separate commitment control definition, they must commit or roll back their changes, since the application cannot do that. There are potential record-locking and consistency exposures in this situation. If the trigger terminates normally without committing its changes, the application cannot release the locks on those records. Having different commitment definitions for triggers and applications should be done only if strictly necessary, as we show in our example (see “Audit trail trigger in ILE RPG - T4249IADT” on page 392).

We now discuss what happens to the database changes when a trigger encounters a failure and ends abnormally. We must consider how DB2 Universal Database for iSeries deals with the database changes that activate the trigger and with the database changes made by the trigger itself.

There are several scenarios depending on whether triggers and applications are using commitment control:

- Both the trigger and application use commitment control
  
  In this case, a failure during the trigger program execution is going to cause the automatic rollback of all changes that were made by the trigger program. The originating change operation is also rolled back.

  All the changes related to the database operation that fired the trigger are treated as part of an atomic transaction; the system ensures that all of them are either rolled back together or remain. When the atomic transaction is rolled back, any other database change previously made by the application is not affected. Consider the scenario in Figure 11-21.

  The application shown in Figure 11-21 performs database changes, with one of them firing a trigger. The trigger itself is executing database changes, and an error occurs at the DELETE statement (indicated by the arrow). DB2 Universal Database for iSeries automatically rolls back all the changes enclosed by the dotted line, but does not affect the other operations made by the application. The application determines whether to commit those changes.
Figure 11-21  Atomic transactions with triggers

- Only the application uses commitment control
  In this case, all the changes performed by the trigger are not rolled back, but the change operation that fired the trigger is rejected. Data integrity may be violated in this situation.

- Only the trigger uses commitment control
  If the trigger activation group ends for an unexpected exception for which you did not provide an exception handler, all the changes made by the trigger are automatically rolled back. Remember that, if you handle exceptions inside the trigger, this terminates normally and you need to explicitly issue a rollback operation to bring all the changes back. The originating change is cancelled only in the case of a BEFORE trigger.

  When an OPM trigger fails, the changes made by the trigger are not automatically rolled back. An explicit rollback should always be issued by the trigger to avoid lock exposure. The originating change is cancelled only in the case of a BEFORE trigger.

- Trigger and application do not use commitment control
  In this case, the changes made by the trigger are not rolled back; the originating change is cancelled in case of a BEFORE trigger.

**Important:** If your triggers modify database data, we suggest that you use commitment control in both applications and triggers since this is the safest way to ensure the integrity of your data.
Table 11-3 summarizes the behavior of DB2 Universal Database for iSeries in case of an unmonitored trigger failure.

**Table 11-3 Triggers and commitment control definition**

<table>
<thead>
<tr>
<th>Application program</th>
<th>Trigger program</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT=YES</td>
<td>COMMIT=YES</td>
<td>The originating change performed by the application and the changes made by the trigger are rolled back together.</td>
</tr>
<tr>
<td>COMMIT=YES</td>
<td>COMMIT=NO</td>
<td>The change that activated the trigger is rolled back. The changes made by the trigger are not rolled back.</td>
</tr>
<tr>
<td>COMMIT=NO</td>
<td>COMMIT=YES</td>
<td>After an unhandled exception, the changes made by the trigger are rolled back automatically if the activation group ends. For OPM triggers, an explicit rollback operation should be issued. The originating change is rolled back only in case of a BEFORE trigger.</td>
</tr>
<tr>
<td>COMMIT=NO</td>
<td>COMMIT=YES</td>
<td>In the case of an AFTER trigger, all changes are not rolled back. In the case of a BEFORE trigger, only the originating change is rolled back.</td>
</tr>
</tbody>
</table>

11.4 Designing trigger programs

This section describes some specific implementations of trigger programs that fit our Order Entry application scenario. We show the main advantages of using triggers, instead of coding all the functions as part of a specific application.

11.4.1 Order Entry application scenario

The main flow of the application is described in Chapter 2, “Stored procedures, triggers, and user-defined functions: Order entry application” on page 11. Here, we focus our attention on some particular functions that can be implemented by trigger programs. Before going into a detailed discussion of each of these trigger programs, it is useful to have a brief overview of what functions have been developed and a list of the samples included in this IBM Redbook. We have included three trigger programs in our scenario:

- **Verify Salesperson/Customer Association**
  Verifies whether a salesperson is allowed to deal with a specific customer, and logs any attempt to violate the restrictions.
  We developed several versions of this program in different languages:
  - T4249CADT: SQL COBOL (“Audit trail trigger in COBOL SQL - T4249CADT” on page 388)
  - T4249IADT: Native ILE RPG (“Audit trail trigger in ILE RPG - T4249IADT” on page 392)
  - T4249CCAT: Native ILE C (“Audit trail trigger in ILE C - T4249CCAT” on page 421)

This example is discussed in detail in 11.4.2, “Audit trail trigger example programs” on page 381. That section also reports a COBOL version of the application program that fires the trigger (T4249CADT). This program creates a new order and inserts the new Order Header in the database.
Final Order Check and Invoice Printing

Performs several authority checks on updates of the Order Header and prints the invoice when the grand total is updated. Here is the list of the examples in the different languages:

- T4249CINV: Native COBOL ("Update trigger on Order Header - T4249CINV" on page 400)
- T4249IINV: Native ILE RPG ("Invoice trigger in ILE RPG - T4249IINV" on page 407)
- T4249CCIV: ILE C ("Invoice trigger in ILE C - T4249CCIV" on page 412)

In the Order Entry application scenario, this trigger is activated by the program that is responsible for finalizing the order (FNLORD). An RPG version of this program is also reported in this section (T4249FNLO; see “Finalize order program - T4249FNLO” on page 397).

Check Customer Credit Limit

Prevents issuing an order if the customer credit limit is exceeded and sends a fax to customers who reached 90 percent of their credit limit. If the customer is a special customer and has reached 90 percent of his credit limit, his credit limit is increased by 30 percent. This is an example of a trigger program that changes the record that activated the trigger.

- T4249CCTA: ILE RPG ("ILE RPG trigger program to send a fax - T4249CTA" on page 418)
- T4249CCTA1: ILE C ("Changing the trigger buffer example" on page 425)
- T4249CCTA2: ILE C ("Calling the trigger program recursively" on page 427)

11.4.2 Audit trail trigger example programs

Whenever a new order is created, we need to make sure that the sales representative who is taking the order has the right to deal with that particular customer. In our sales department, each team of sales representatives has its own pool of customers assigned to them. The sales representative uses our application to place the orders. The first display that the sales representative works with is the Order Entry input display (Figure 11-22).

![Order Entry display](image-url)
For the sake of simplicity, the order number is an input field. In a real application, it is generated by the system. In our scenario, we prevent duplicate orders, thanks to a unique constraint on the corresponding database field. It is useful to also report the DDS layout of this display file (Figure 11-23).

```
** This is the Order Entry Display File
**
** This covers the take call order header entry
**

    REF(ORDENTREF)
    INDARA
    TEXT('ORDER ENTRY')
    ASSUME
    OVERLAY
    CA03(15 'END OF PROGRAM')
    2 35 'ORDER ENTRY '
    DSPATR(BL)
    5 3 'ORDER NUMBER: '
    ORHNBR R B 5 20 CHECK(ME)
    ERRMSG('ORDER ALREADY EXISTS' 99)
    DSPATR(HI)
    6 3 'CUST. NUMBER: '
    CUSNBR R B 6 20 CHECK(ME)
    ERRMSG('CUSTOMER NOT FOUND' 98)
    97
    98
    ERRMSG('ERROR - SEE + JOBLOG')
    DSPATR(HI)
    7 3 'ORDER DATE: '
    8 3 'DELIVERY DATE: '
    ORHDTE 10 B 7 20
    ORHDLY 10 B 8 20
    DSPATR(HI)
    DSPATR(HI)
    R EXITLINE
    TEXT('EXIT LINE')
    23 3 'ENTER = ACCEPT"
    F3 = EXIT'
```

*Figure 11-23 DDS for the display file*

If we develop this application in a traditional environment without using any of the advanced features of DB2 Universal Database for iSeries, we code all of the consistency checks within the application itself.

In particular, the Order Header input program should check the Customer/Salesrep relationship file to make sure the current user ID that identifies the sales representative is actually allowed to place an order for that customer. The pseudo-code of this traditional version of the Insert Order Header program is shown in Figure 11-24.
In Figure 11-24, all of the data validation for logical consistency is performed by the application program. By using the DB2 Universal Database for iSeries advanced features, we can easily delegate the integrity verification (as indicated by 1) to the system, implementing referential integrity between the CUSTOMER file and the Order Header (ORDERHDR) file.

We also need to keep track of any violation attempt against our sales organization policy. This check is performed in the line marked by 2 by the traditional application. This may lead to some security exposures, since this business rule can be circumvented by using a different data input interface, such as Interactive SQL, DFU, or a different application. Also, if we need to change the rules, we probably need to modify all the existing applications dealing with the Order Header file. For this reason, we decided to implement this check at the database level by developing an insert trigger program. The layout of the new application is shown in Figure 11-25.

In Figure 11-25, all the data verification is performed by the database shown by 3 when the insert takes place. Notice that the resulting application is much more compact and simple. We only need to check the return code after the insert operation has completed. All the logic for business rules enforcement has been moved outside the application itself and is executed in any case, even if we use a different data interface.

In the following example, you can see a COBOL SQL implementation of this procedure. The operation that activates the trigger and the referential integrity check is marked with 4. Immediately after the SQL insert, the application checks the SQLCODE for errors and reports the correct message to the user.

```
Display input map;
Read map;
Initialize grand total to 0;
if CUSTOMER_NUMBER not in CUSTOMER file
   send error message;
else
   retrieve USER PROFILE;
   if USER_PROFILE is not associated to CUSTOMER_NUMBER
      send error message
      log violation attempt
   else
      insert ORDERHDR
      if duplicate_key
         send error message
      else
         return;
```

```
Display input map;
Read map;
Initialize grand total to 0;
Insert record in ORDERHDR;
if I/O error
   show error;
else
   return;
```
Order Header entry program - T4249CINS

PROCESS OPTIONS.
IDENTIFICATION DIVISION.
PROGRAM-ID. T4249CINS.
  AUTHOR. PROGRAMMER NAME.
  INSTALLATION. ITSC LABORATORY.
  DATE-COMPILED.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
  SOURCE-COMPUTER. IBM-AS400.
  OBJECT-COMPUTER. IBM-AS400.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT T4249OHRD ASSIGN TO WORKSTATION-T4249OHRD
   ORGANIZATION IS TRANSACTION
   FILE STATUS IS STATUS-ERR.

**********************************************************
DATA DIVISION.
FILE SECTION.
FD T4249OHRD
   LABEL RECORD ARE STANDARD.
  01 DSP01.
     COPY DDS-ALL-FORMATS OF T4249OHRD.
***********************************************************
WORKING-STORAGE SECTION.
  01 DSPFIL-INDICS.
     COPY DDS-ALL-FORMATS-INDIC OF T4249OHRD.
  77 IND-ON       PIC 1 VALUE B"1".
  77 IND-OFF     PIC 1 VALUE B"0".

  01 JOBA-AREA.
     03 BYTES-RTN    PIC 9(8) BINARY VALUE 0.
     03 BYTES-AVAIL  PIC 9(8) BINARY VALUE 0.
     03 JOBNAME      PIC X(10).
     03 USERNAME     PIC X(10).
     03 JOBNUMBER    PIC X(6).
*================================================================*
* Parameters for retrieve job attributes - USERID *
*================================================================*
  01 RTV-JOBA.
     03 RTV-JOB-VAR   PIC X(50).
     03 RTV-JOB-LEN   PIC 9(8) BINARY VALUE 50.
     03 RTV-JOB-FMT   PIC X(8) VALUE "JOBIO400".
     03 RTV-JOB-NAME  PIC X(26) VALUE "*".
     03 RTV-JOB-ID    PIC X(16) VALUE " ".

  01 STATUS-ERR   PIC XX.
  01 ORDNUM       PIC X(5).
  01 CUSTOMER     PIC X(5).
  01 ODATE        PIC X(10).
  01 ODLY         PIC X(10).
  01 OTOTAL       PIC S9(9)V9(2) COMP-3.
EXEC SQL
  INCLUDE SQLCA
END-EXEC.

LINKAGE SECTION.

01 CUSTNBR    PIC X(5).
01 ORDNBR     PIC X(5).
01 RTCODE     PIC X.

*====================================================================*
* This program has three output parameters: Customer numb.*
* Order number and Return code. The return code can be: *
* Rtcode = 0 - OK      Rtcode = 2 - F3                    *
*====================================================================*

PROCEDURE DIVISION USING CUSTNBR, ORDNBR, RTCODE.

DECLARATIVES.

TRANSACTION-ERROR SECTION.
  USE AFTER STANDARD ERROR PROCEDURE T4249OHRD.

WORK-STATION-ERROR-HANDLER.
  GOBACK.
END DECLARATIVES.

MAIN-LINE SECTION.
  OPEN I-O T4249OHRD.
  PERFORM INITIzA$-HEADER.

*====================================================================*
* Call API to get job attributes and move the *
* output parameter into the work area           *
*====================================================================*

CALL "QUSRJOBI" USING RTV-JOB-VAR,
RTV-JOB-LEN,
RTV-JOB-FMT,
RTV-JOB-NAME,
RTV-JOB-ID.

MOVE RTV-JOB-VAR TO JOBA-AREA.
MOVE "0" TO RTCODE.
MOVE 0 TO INSERTOK.
MOVE IND-OFF TO IN15 IN ORDER-I-INDIC.
WRITE DSP01 FORMAT IS "EXITLINE".
PERFORM ORDER-ENTRY UNTIL
  IN15 IN ORDER-I-INDIC EQUAL IND-ON OR
  INSERTOK EQUAL 1.

IF IN15 IN ORDER-I-INDIC = IND-ON THEN
  MOVE "2" TO RTCODE
ELSE
  IF INSERTOK = 1 THEN
    MOVE "0" TO RTCODE.
*====================================================================*
*We are not closing the file, because we are overlapping screens*
*====================================================================*

  CLOSE T4249OHRD.
  GOBACK.
ORDER-ENTRY.
PERFORM WRITE-READ-ORDER.

MOVE ORHNBR OF ORDER-I TO ORDNUM.
MOVE CUSNBR OF ORDER-I TO CUSTOMER.
MOVE ORHDTE OF ORDER-I TO ODATE.
MOVE ORHDLY OF ORDER-I TO ODLY.
MOVE ZEROS TO OTOTAL.
MOVE CUSTOMER TO CUSTNBR.
MOVE ORNUM TO ORDNBR.

IF IN15 IN ORDER-I-INDIC NOT EQUAL IND-ON THEN

* The programs inserts an order in ORDERHDR file.
*
EXEC SQL
INSERT INTO ORDENTL/ORDERHDR VALUES(:ORDNUM, :CUSTOMER,
END-EXEC

IF SQLCODE EQUAL 0 THEN
MOVE 1 TO INSERTOK
ELSE
*==========================================================*
* After the insert operation, you should monitor the       *
* following SQLCODEs:                                      *
*   SQL0530(-530) - Referential Integrity violation        *
*   SQL0803(-803) - Order Header already exists            *
*   SQL0443(-443) - Trigger program signalled an exception *
*==========================================================*

IF SQLCODE EQUAL -530 THEN
MOVE IND-ON TO IN98 OF ORDER-O-INDIC
MOVE SPACES TO ORHNBR OF ORDER-O
MOVE CUSTOMER TO CUSNBR OF ORDER-O
ELSE
IF SQLCODE EQUAL -803 THEN
MOVE IND-ON TO IN99 OF ORDER-O-INDIC
ELSE
MOVE IND-ON TO IN97 OF ORDER-O-INDIC.
*================================================================*

INITIAZ-HEADER.

MOVE SPACES TO ORHNBR OF ORDER-O.
MOVE SPACES TO CUSNBR OF ORDER-O.
MOVE "0001-01-01" TO ORHDTE OF ORDER-O.
MOVE "0001-01-01" TO ORHDLY OF ORDER-O.

WRITE-READ-ORDER.

WRITE DSP01 FORMAT IS "ORDER" INDICATORS ARE ORDER-O-INDIC.
MOVE IND-OFF TO ORDER-I-INDIC ORDER-O-INDIC.
READ T4249OHDR RECORD INDICATORS ARE ORDER-I-INDIC.

The trigger program that is fired by this application function checks that the sales representative has actually been assigned to that particular customer. If this condition is not satisfied, the trigger logs an entry in an audit file.
The pseudo-code of the trigger is shown in Figure 11-26.

```
// Figure 11-26  Pseudo-code for the audit trail trigger

Retrieve USERPROFILE;
  Read SalesRep/Customer file;
  if USERPROFILE is associated with CUSTOMER
    return;
  else
    log entry in AUDIT file;
    send escape message to application;
    commit;
    return;
```

The commit operation, highlighted in bold in Figure 11-26, needs some discussion. We want to be sure that, whenever a violation occurs, the trigger logs the attempt. For this reason, the trigger should run in a separate commitment definition, so that the commit operation, highlighted in bold, affects only the insert operation in the audit file. This way, the interface originating the violation cannot roll back this log entry. For this special requirement of this particular application, we want the trigger to run in a different commitment definition. In general, we recommend that you run your applications and triggers in the same commitment definition. The application should commit or roll back all the changes at the end of the logical transaction (see 11.3.1, “Commitment control and triggers” on page 378).

However, if you code your triggers in an OPM language, you cannot force them to use a separate commitment definition. In this case, avoid using commitment control in triggers, unless you do not want the application to roll back the changes made by your triggers.

If you are implementing these kinds of triggers in the Integrated Language Environment®, create the program to run in a named activation group. If your trigger is an SQL trigger, commitment control is automatically started at the activation group level for you. But, if you are using the native interface, you should start commitment control for the named activation group. See “Audit trail trigger example in ILE RPG” on page 392 and “Audit trail trigger example in ILE C” on page 397.

**Audit trail trigger example in COBOL SQL (OPM)**

This version of the trigger program must perform uncommitted changes. To create the program shown in the following example, use the command:

```
CRTSQLCBL PGM(ORDENTLIB/T4249CADT)
SRCFILE(ORDENTLIB/QLBLSRC)
COMMIT(*NONE)
```

It is interesting to point out that this version of the trigger is enabled to send a message to the display if the job that activated the trigger is interactive. This technique might be useful if you want to provide the end user with a better understanding of what happened during the trigger execution. Generally, when a trigger fails, applications send the user a generic message referencing the job log. The trigger program checks for the job type highlighted in bold and opens an appropriate display file to show the message only in the case of an interactive job.
Audit trail trigger in COBOL SQL - T4249CADT

PROCESS OPTIONS.
IDENTIFICATION DIVISION.
PROGRAM-ID. T4249CADT.
AUTHOR. IBM.
INSTALLATION. ITSO.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-AS400.
OBJECT-COMPUTER. IBM-AS400.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT DSPMSGD ASSIGN TO WORKSTATION-DSPMSGD
   ORGANIZATION IS TRANSACTION.
**********************************************************
DATA DIVISION.
FILE SECTION.
FD DSPMSGD
   LABEL RECORD ARE STANDARD.
   01 DSP01.
   COPY DDS-ALL-FORMATS OF DSPMSGD.
***********************************************************
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION
END-EXEC.
01 CUSTOMNBR
   PIC X(5).
01 SRNBR
   PIC X(10).
01 CHECKVAR
   PIC S9.
EXEC SQL END DECLARE SECTION
END-EXEC.

01 WINMSG.
   03 MSG1
      PIC X(30)
      VALUE "YOU CANNOT DEAL WITH CUSTOMER".
   03 MSG2
      PIC X(30)
      VALUE "TRIGGER ERROR - SEE JOBLOG".
   03 MSGDSP
      PIC X(30).
*=================================================================* *
* This is the area to receive the record image                      *
*=================================================================* *
01 ORDER-HEADER.
   03 ORDHnbr
      PIC X(5).
   03 CUSNBR
      PIC X(5).
   03 ORHDTE
      PIC X(10).
   03 ORHDLY
      PIC X(10).
   03 ORDTOT
      PIC S9(9)V9(2) COMP-3.
   03 SRNNBR
      PIC X(10) VALUE " ".

01 JOBA-AREA.
   03 BYTES-RTN
      PIC 9(8) BINARY VALUE 0.
   03 BYTES-AVAIL
      PIC 9(8) BINARY VALUE 0.
   03 JOBNUMBER
      PIC X(10).
   03 USERNAME
      PIC X(10).
   03 INTERNALJID
      PIC X(6).
   03 INTERNALJID
      PIC X(16).
   03 JOBSTATUS
      PIC X(10).
   03 JOBTYPE
      PIC X(1).
EXEC SQL
INCLUDE SQLCA
END-EXEC.

*===============================================================================*
* These are the parameters for the API QUSRJOBI  *
* to get job attributes - we need the USERID  *
*===============================================================================*
  01 RTV-JOBA.
    03 RTV-JOB-VAR      PIC X(61).
    03 RTV-JOB-LEN      PIC 9(8) BINARY VALUE 61.
    03 RTV-JOB-FMT      PIC X(8) VALUE "JOBI400".
    03 RTV-JOB-NAME     PIC X(26) VALUE "".
    03 RTV-JOB-ID       PIC X(16) VALUE "".

*===============================================================================*
* These are the parameters needed to signal an   *
* exception inside trigger programs             *
*===============================================================================*
  01 SNDPGMMSG.
    03 SND-MSG-ID       PIC X(7)  VALUE "TRG0005".
    03 SND-MSG-FILE     PIC X(20) VALUE "ORDMSGF ORDENTLIB".
    03 SND-MSG-DATA     PIC X(30) VALUE "TRIGGER ERROR 66."
    03 SND-MSG-LEN      PIC 9(8) BINARY VALUE 0.
    03 SND-MSG-QUEUE    PIC X(10) VALUE "".
    03 SND-PGM-STACK    PIC 9(8) BINARY 0.
    03 SND-MSG-KEY      PIC X(4) VALUE "  ".
    03 SND-ERROR-CODE   PIC 9(8) BINARY 66.
    05 PROVIDED        PIC 9(8) BINARY 0.
    05 AVAILABLE       PIC 9(8) BINARY 0.
    05 EXCEPTION-ID    PIC X(7) VALUE "  ".
    05 FILLER          PIC X(1) VALUE "  ".
    05 EXCEPTION-DATA  PIC X(50) VALUE "  ".

*===============================================================================*
* PARM 1 = Trigger Buffer             *
* PARM 2 = Trigger Length            *
*===============================================================================*
LINKAGE SECTION.
  01 PARM-1.
    03 FILE-NAME        PIC X(10).
    03 LIB-NAME         PIC X(10).
    03 MEM-NAME         PIC X(10).
    03 TRG-EVENT        PIC X.
    03 TRG-TIME         PIC X.
    03 CMT-LCK-LVL      PIC X.
    03 DATA-AREA-CCSID  PIC 9(8) BINARY.
    03 FILLER           PIC X(8).
    03 DATA-OFFSET.
      05 OLD-REC-OFF    PIC 9(8) BINARY.
      05 OLD-REC-LEN    PIC 9(8) BINARY.
      05 OLD-REC-NULL-MAP PIC 9(8) BINARY.
      05 OLD-REC-NULL-LEN PIC 9(8) BINARY.
      05 NEW-REC-OFF    PIC 9(8) BINARY.
      05 NEW-REC-LEN    PIC 9(8) BINARY.
      05 NEW-REC-NULL-MAP PIC 9(8) BINARY.
      05 NEW-REC-NULL-LEN PIC 9(8) BINARY.
      05 FILLER         PIC X(16).
PROCEDURE DIVISION USING PARM-1, PARM-2.

MAIN-PROGRAM SECTION.

START-SECTION.

* New record image received from the Database manager*
* and moving it to the working storage          *

MOVE NEW-RECORD TO ORDER-HEADER.

* Call API to get the job attributes - USERID *
* and move the receiving contents to working storage *

CALL "QUSRJOBI" USING RTV-JOB-VAR,
RTV-JOB-LEN,
RTV-JOB-FMT,
RTV-JOB-NAME,
RTV-JOB-ID.

MOVE RTV-JOB-VAR TO JOBA-AREA.

IF SRNNBR OF ORDER-HEADER EQUAL USERNAME THEN
  MOVE SRNNBR OF ORDER-HEADER TO SRNBR
  MOVE CUSNBR OF ORDER-HEADER TO CUSTOMNBR
  EXEC SQL SELECT 1 INTO &colon.CHECKVAR
  FROM ORDENTL/SALESCUS
  WHERE SALESREP_NUMBER = &colon.SRNBR AND
        CUSTOMER_NUMBER = &colon.CUSTOMNBR
END-EXEC

  IF SQLCODE = 100 THEN
    PERFORM ERROR-MSG
  ELSE
    NEXT SENTENCE
  ELSE
    PERFORM ERROR-MSG.

* If salesperson can deal with the customer just    *
* send a message that the salesperson was found.   *
* Otherwise, we will audit trail and signal exception *
* so the DB change operation will fail. We are     *
* testing the USERID to ensure that whatever interface*
* invokes the trigger, the salesperson will be checked*

CLOSE DSPMSGD.

GOBACK.

ERROR-MSG.
MOVE CUSNBR OF ORDER-HEADER TO CUSTOMNBR.
MOVE SRNNBR OF ORDER-HEADER TO SRNBR.

EXEC SQL
 INSERT INTO ORDENTLIB/AUDTFIL (SALESREP_NUMBER,
 CUSTOMER_NUMBER) VALUES(&colon.SRNBR, &colon.CUSTOMNBR)
END-EXEC.

******************************************************
* If the insert fails inside trigger program you      *
* should provide the appropriate escape message      *
* signalling an exception to the application         *
******************************************************

IF SQLCODE NOT EQUAL 0 THEN
 MOVE "TRG0003" TO SND-MSG-ID
 MOVE MSG2 TO MSGDSP
ELSE

******************************************************
* If salesperson cannot deal with the customer       *
* send an escape message - DB change operation will  *
* not happen                                         *
******************************************************

MOVE MSG1 TO MSGDSP
MOVE "TRG0002" TO SND-MSG-ID.
MOVE "*ESCAPE" TO SND-MSG-TYPE.
PERFORM SND-MSG.

SND-MSG.

******************************************************
* If the job is interactive, we can send a message   *
* to the screen.                                     *
******************************************************

IF JOBTYPE EQUAL "I" THEN
 OPEN I-O DSPMSGD
 MOVE MSGDSP TO MSGFLD OF DSPMSGD-O
 WRITE DSP01 FORMAT IS "DSPMSGD"
 READ DSPMSGD
 CLOSE DSPMSGD
END-IF.

******************************************************
* Using the API to signal back to the application    *
* an escape message in order to make the insert fail. *
******************************************************

CALL "QMHSNPDMP" USING SND-MSG-ID,
 SND-MSG-FILE,
 SND-MSG-DATA,
 SND-MSG-LEN,
 SND-MSG-TYPE,
 SND-MSG-QUEUE,
 SND-PGM-STACK,
 SND-MSG-KEY,
 SND-ERROR-CODE.

IF AVAILABLE IS NOT EQUAL 0 THEN
 DISPLAY "QMHSNPDMP API ERROR" SND-MSG-ID.
 GOBACK.
Audit trail trigger example in ILE RPG

The following example of the audit trail trigger was implemented using the ILE RPG language. As mentioned in the general discussion (11.4.2, “Audit trail trigger example programs” on page 381), we want a separate commitment definition for this program. For this reason, follow these steps:

1. Write an ILE CL program. Enter the command:
   
   STRCMTCTL CMTSCOPE(*ACTGRP)

2. Create the corresponding CL module.

3. Call the CL program from within RPG before any file has been opened. Use the User Open (USROPN) option in F specs, and open the file explicitly in your program.

4. Create the corresponding RPG module.

5. Create a program specifying a named activation group.

The commands needed to create this program are:

   CRTRPGMOD   MODULE(ORDENTLIB/T4249IADT)
   SRCFILE(ORDENTLIB/QRPGILE)
   SRCMBR(T4249IADT)

   CRTCLMOD   MODULE(ORDENTLIB/T4249CTL)
   SRCFILE(ORDENTLIB/QCLSRC)

   CRTPGM   PGM(ORDENTLIB/T4249IADT)
   MODULE(ORDENTLIB/T4249IADT  ORDENTLIB/T4249CTL)
   TEXT(*ENTMODTXT)
   ACTGRP(AUDIT)

Figure 11-27 shows the CL code needed to start commitment control in a separate activation group.

   PGM
   MONMSG MSGID(CPF0000)
   STRCMTCTL LCKLVL(*CHG) CMTSCOPE(*ACTGRP)
   ENDPGM

   Figure 11-27   Starting a separate commitment definition

The RPG program is shown in the following section.

Audit trail trigger in ILE RPG - T4249IADT

   FSALESCUS IF E   K DISK INFDS(FILDS1)
   F
   F
   FAUDTFIL O E   K DISK COMMIT
   F
   F
   FDSPMSGD CF E   WORKSTN INFDS(FILDS2)
   F

   * Program status subroutines
   D FILDS1 DS
   D FIL1 *FILE
   D REC1 *RECORD
   D OP1 *OPCODE
   D STS1 *STATUS
   D RTN1 *ROUTINE

   *====================================================================*
Chapter 11. External triggers

D FILDS2 DS
D FIL2 *FILE
D REC2 *RECORD
D OP2 *OPCODE
D STS2 *STATUS
D RTN2 *ROUTINE

*====================================================================*
* Trigger Buffer passed by the system to this PGM                *
*====================================================================*
* FNAME = PHYSICAL FILE NAME                                     *
* LNAME = PHYSICAL FILE LIBRARY                                   *
* MNAME = MEMBER NAME                                             *
* TEVEN = TRIGGER EVENT                                           *
* TTIME = TRIGGER TIME                                            *
* CMTLCK= COMMIT LOCK LEVEL                                      *
* FILL1 = RESERVED                                               *
* CCSID = CCSID                                                  *
* FILL2 = RESERVED                                               *
* OLD0FF= OFFSET TO THE ORIGINAL RECORD                          *
* OLDLEN= LENGTH OF THE ORIGINAL RECORD                          *
* ONOFF = OFFSET TO THE ORIGINAL RECORD NULL BYTE MAP            *
* ONLEN = LENGTH OF THE NULL BYTE MAP                            *
* NOFF = OFFSET TO THE NEW RECORD                                 *
* NEWLEN= LENGTH OF THE NEW RECORD                               *
* NNOFF= OFFSET TO THE NEW RECORD NULL BYTE MAP                  *
* NNLEN = LENGTH OF THE NULL BYTE MAP                            *
* RESV3 = RESERVED                                               *
* OREC = OLD RECORD                                               *
* OOMAP = NULL BYTE MAP OF OLD RECORD                            *
* RECORD= NEW RECORD                                             *
* NMAP = NULL BYTE MAP OF NEW RECORD                             *
*====================================================================*

D PARM1 DS
D FNAME 1 10
D LNAME 11 20
D MNAME 21 30
D TEVEN 31 31
D TTIME 32 32
D CMTLCK 33 33
D FILL1 34 36
D CCSID 37 40B 0
D FILL2 41 48
D OLD0FF 49 52B 0
D OLDLEN 53 56B 0
D ONOFF 57 60B 0
D ONLEN 61 64B 0
D NOFF 65 68B 0
D NEWLEN 69 72B 0
D NNOFF 73 76B 0
D NNLEN 77 80B 0
D RESV3 81 96
D OREC 97 142
D OOMAP 143 148
D RECORD 149 194
D NMAP 195 200

*====================================================================*
* Definition of the length buffer received by trigger             *
*====================================================================*

D PARM2 DS
D LENG 1 4B 0
* This is the area to receive the new record image *
*---------------------------------------------------------------------*
D NRECORD     DS
D ORHNBR      1  5
D CUSNBR      6 10
D ORHDTE      11 20
D ORHDLY      21 30
D OTOTAL      31 36P 0
D SRNBR       37 46
*---------------------------------------------------------------------*
* This is the receiving parms from the API QUSRJOBI*
* Retrieve the job attributes - USERID  *
*---------------------------------------------------------------------*
D RTVAPI     DS
D BTRN       1  4B 0
D BAVAIL     5  8B 0
D JOBNAM     9 18
D USERID     19 28
D JOBNBR     29 34
D JOBID      35 50
D JOBSTS     51 60
D JOBTPY     61 61
*---------------------------------------------------------------------*
* Output parameters for QMHSNDPM                                 *
*---------------------------------------------------------------------*
D MSGERR     DS
D PROVID     1  4B 0
D AVAIL      5  8B 0
D RTNMSG     9 15
D RSVR       16 16
D RTNDTA     17 56
*
D FLDS     DS
D MSGLEN     1  4B 0
D PGMSTK     5  8B 0
D RTVLEN     9 12B 0
D MSGQLEN    13 16B 0
D PGMWTT     17 20B 0
*
D MSG1     C     CONST('S/C NOT ALLOWED  ')
D MSG2     C     CONST('TRIGGER ERROR    ')
*---------------------------------------------------------------------*
* RPG ILE - Call stack entry - signal exception parameters *
*---------------------------------------------------------------------*
D LIBNAM     C     CONST('ORDENTLIB')
D MSGQNAM    C     CONST('_QRNP_PEP_T4249IADT')
D MODNAME    C     CONST('*NONE     *NONE     ')
*
C     *ENTRY      PLIST
C     PARM1     PARM     PARM1
C     PARM2     PARM     PARM2
*---------------------------------------------------------------------*
* Parameter needed to signal an exception inside triggers*
*---------------------------------------------------------------------*
C     PLIST1     PLIST
C     PARM     MSGID    7
C     PARM     MSGF    20
C     PARM     MSGDTA  25
Chapter 11. External triggers

C PARM MSGLEN
C PARM MSGTYP 10
C PARM MSGQUE 19
C PARM PGMSKX
C PARM MSGKEY 4
C PARM MSGERR
C PARM MSGQLEN
C PARM CSEQUAL 20
C PARM PGMNTT

*===================================================*
* Retrieve job attributes - USERID *
*===================================================*
C PLIST2 PLIST
C PARM RTVVAR 61
C PARM RTVLN
C PARM RTVFMT 8
C PARM RTVNAM 26
C PARM RTVID 16

* KEYFLD KLIST
C KFLD SRNBR
C KFLD CUSNBR

*===================================================*
* Initialization for *PSSR routine if some       *
* unmonitored errors occur                       *
*===================================================*
C MOVEL MSG2 MSGFLD
C MOVEL 'TRG0005' MSGID

*===================================================*
* Start a different commitment control definition *
*===================================================*
C CALLB 'T4249CTL'
C OPEN AUDTFIL 88

*===================================================*
* Get job attributes - USERID                      *
*===================================================*
C Z-ADD 61 RTVLN
C MOVEL 'JOBI0400' RTVFMT
C MOVEL(P) '*' RTVNAM
C MOVE ':' RTVID
C CALL 'QUSRJOBI' PLIST2

*===================================================*
* Move THE NEW RECORD RECIEVED BY TRIGGER TO WORK.FLDS*
*===================================================*
C MOVEL RECORD NRECORD

*=================================================================
* This program will check if the salesperson can deal *
* with the customer - testing first for the USERID *
* and with the salesperson because another interface *
* may call the same trigger, and you have to think *
* about it. Besides that we are checking if they exist*
* in the SALESCUS file, if they don't trigger program *
* will signal an exception to the application and DB *
* change operation will not happen and will audit     *
* trail, otherwise the change will run successfully  *
*=====================================================*
C     SRNBR      IFNE      USERID
C     GOTO       NTFND
C     END

C  KEYFLD      CHAIN      SALECS                             71
C  *IN71       IFEQ      '0'
C  GOTO       EOFIN
C  END
C  NTFND      TAG

C  WRITE       AUDIT                                99
C  *IN99       IFEQ      *ON
C  MOVEL       MSG2      MSGFLD
C  MOVEL       'TRG0005'      MSGID
C  ROLBK
C  ELSE
C  MOVEL       MSG1      MSGFLD
C  MOVEL       'TRG0002'      MSGID
C  COMMIT
C  END
C  EXSR       *PSSR
C  EOFIN      TAG
C  CLOSE      SALESCUS
C  RETURN

*==========================================================*
* Signalling an exception inside trigger program           *
* We check if the job is interactive or not and if it is   *
* we send a message to user before signalling the exception*
*==========================================================*
C  *PSSR       BEGSR
C  JOBTYP      IFEQ      'I'
C  EXFMT      DSPM
C  END
C  MOVEL(P)    LIBNAM    LIB             10
C  MOVEL(P)    'ORDMSGF'    ID             10
C  ID          CAT(P)    LIB           MSGF
C  MOVE       ' '       MSGDTA
C  Z-ADD      25       MSGLEN
C  MOVEL(P)    '**ESCAPE'    MSGTP
C  MOVEL(P)    MSGQNM    MSGQUE
C  MOVEL(P)    MODNAME    CSEQUAL
C  MOVE       ' '       MSGDTA
C  Z-ADD      1       PGMSMTK
C  Z-ADD      19       MSGQLEN
C  MOVE       ' '       MSGKEYY
C  Z-ADD      66      PROVID
C  Z-ADD      0       AVAIL
C  MOVE       ' '       RTNMSG
C  MOVE       ' '       RSVR
C  MOVE       ' '       RTNDTA
C  MSGQUE      DSPLY
C  CALL      'QMHSNDPM'    PLIST1
C  AVAIL      IFNE      0
C  RTNMSG      DSPLY
C  RTNDTA      DSPLY
C  END
C  ENDSR
Audit trail trigger example in ILE C
ILE C allows you to start commitment control at the activation group scope by using the system ANSI C function. Any CL command executed by this function runs in the same activation group as the program issuing the system function.

For the audit trail example in ILE C, refer to “Soft coding the trigger buffer in ILE C” on page 421. In addition to the features covered by the COBOL and RPG programs, the ILE C example shows how to soft code the trigger buffer.

11.4.3 Updating a trigger on the Order Header file program examples
Whenever an update takes place on the ORDERHDR file, we want to be sure that the following conditions are satisfied:

- The record can be updated either by the originator of the order or by QSECOFR.
- QSECOFR can update any field in the Order Header program.
- The originator cannot update the customer field, because we want to prevent an order issued for a customer from being rerouted to another one. See the discussion in 11.4.2, “Audit trail trigger example programs” on page 381, about the audit trail trigger.
- If the originator updates the Grand total field, this means that the order is complete. We need to generate the invoice in this case.

Enforcing all these rules in a traditional environment is difficult, and the enforcement is restricted to the applications that implement this logic.

In our scenario, we provide an update trigger on the Order Header (ORDERHDR) file to perform all these functions. The trigger complements the Order Entry application because, when the Finalize Order module is called, the grand total is updated and the invoice is automatically generated. In addition, this trigger ensures that our sales department organization policy is never violated. The program prevents a sales representative from placing a “dummy” order for a customer to which they are authorized and rerouting it to a different customer at a later time.

In our application scenario, this trigger plays a significant role when the order is finally submitted. At that time, the procedure responsible for finalizing the order (FNLOAD) is invoked and updates the Order Header file with the order grand total. In the following section, you can follow the logic of this function and look at the operation that fires this trigger, which is highlighted in bold.

**Finalize order program - T4249FNLO**

```
F*****************************************************************
F*     This program performs the final processing of the order
F*     information; updates the grand total in the Order Header
F*     and updates the Customer total ordered amount.
F*****************************************************************
FFNLORDD CF   E                                WORKSTN
C                  *ENTRY      PLIST
C                                  PARM                  PCUSN     5
C                                  PARM                  PORDN     5
C                                  PARM                  PORDT   112
C                                  PARM                  RTNCD     1
C*
C                                  MOVE  *BLANK   WSR   10
C*
C*     If the total order amount is greater than the customer's
C*     credit limit, the program displays an error message and
```
returns to the calling program with the return code '1'.

C/EXEC SQL
C+ UPDATE ORDENTL/CUSTOMER
C+    SET CUSTOT = CUSTOT + :PORDT
C+    WHERE CUSNBR = :PCUSN AND
C+          CUSCRD >= :PORDT
C/END-EXEC
C*
C                  SQLCOD      IFEQ 100
C                  SQLSTT      ANDEQ'02000'
C                 SETON
C                 MOVE  '1'           RTNCD
C                 EXFMTFNORDER
C                 RETRN
C                 END
C*
C/EXEC SQL
C+ SELECT SRNBR INTO :WSR
C+ FROM ORDENTL/ORDERHDR
C+ WHERE ORHNBR = :PORDN
C/END-EXEC
C*
The total order amount is added to the sales_rep's amount.

C/EXEC SQL
C+ UPDATE ORDENTL/SALESCUS
C+     SET SRAMT = SRAMT + :PORDT
C+     WHERE SRNBR = :WSR
C/END-EXEC
C*
C*     If the sales_rep not found, an error message is displayed, and set return_code to '1'.
C*
C                  SQLCOD      IFEQ 100
C                  SQLSTT      ANDEQ'02000'
C                 SETON
C                 MOVE  '1'           RTNCD
C                 EXFMTFNORDER
C                 RETRN
C                 END
C*
The total order amount is added to the sales_rep's amount.

C/EXEC SQL
C+ UPDATE ORDENTL/ORDERHDR
C+     SET ORHTOT = :PORDT
C+     WHERE ORHNBR = :PORDN
C/END-EXEC
C*
C                  SQLCOD      IFLT 0
C*
C/EXEC SQL
C+ ROLLBACK
C/END-EXEC
Invoice trigger example in native COBOL

This version of the trigger program generates a printed invoice. We use the printer file shown in Figure 11-28 to generate the printout.

** This is the Printer file for Order Entry application  
** This covers the trigger invoice programs  

```
*   
* REF(ORDENTREF)
*   
R HEADER

3      5 'ORDER NUMBER: '
ORHNBR  R  3 19

3      30 'CUST. NUMBER: '
CUSNBR  R  3 45

4      5 'ORDER DATE: '
ORHDTE  10  4 19

4      30 'ORDER TOTAL: '
ORHTOT  R  4 43EDTCDE(6)

5      5 'SALES REP.: '
SRNBR   R  5 19

R DETAIL

9      5 'PRODUCT NBR'
PRDNBR  R  10  5

9      20 'ORDER QTY'
ORDQTY  R  10  20

9      40 'ORDER TOTAL'
ORDTOT  R  10  40EDTCDE(6)
```

*Figure 11-28  Printer file for printing invoices: T4249INV*

Commitment control is not a concern in this case, because the invoice is printed and the trigger performs read-only database access.
To prevent an uncontrolled trigger failure due to possible MCHxxxx errors, such as a Decimal Data Error, we defined a COBOL exception handler through the QLRSETCE API. The program is invoked when an exception occurs, handles the exception, and sends a meaningful escape message to the interface that generated the originating database change.

To create this trigger program, enter the following command:

```
CRTCBLPGM  PGM(ORDENTLIB/T4249CINV)
   SRCFILE(*LIBL/QLBLSRC)
   SRCMBR(*PGM)
CRTCBLPGM  PGM(ORDENTLIB/T4249CHDL)
   SRCFILE(*LIBL/QLBLSRC)
   SRCMBR(*PGM)
```

**Update trigger on Order Header - T4249CINV**

```
PROCESS   OPTIONS.
IDENTIFICATION  DIVISION.
PROGRAM-ID.  T4249CINV.
   AUTHOR. PROGRAMMER NAME.
   INSTALLATION. ITSC LABORATORY.
   DATE-COMPILED.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
   SOURCE-COMPUTER. IBM-AS400.
   OBJECT-COMPUTER. IBM-AS400.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

   SELECT T4249INV ASSIGN TO FORMATFILE-T4249INV
      ORGANIZATION IS SEQUENTIAL
      ACCESS IS SEQUENTIAL.

   SELECT ORDERDTL ASSIGN TO DATABASE-ORDERDTL
      ORGANIZATION IS INDEXED
      ACCESS IS SEQUENTIAL
      RECORD KEY IS EXTERNALLY-DESCRIBED-KEY
      FILE STATUS IS STATUS-ERR.

**********************************************************
DATA DIVISION.
FILE SECTION.
FD   ORDERDTL
   LABEL RECORD ARE STANDARD.
   01   ORDEDTL01.
      COPY DDS-ALL-FORMATS OF ORDERDTL.

FD   T4249INV
   LABEL RECORDS ARE STANDARD.
   01   PRT-REC.
      COPY DDS-ALL-FORMATS-O OF T4249INV.

**********************************************************
```
WORKING-STORAGE SECTION.

77 END-OF-FILE                  PIC X(1) VALUE "0".
   88 NOT-EOF                                VALUE "0".
   88 EOF                                    VALUE "1".

01 QTY                              PIC S9(5).
01 TOTAL                           PIC S9(7)V9(2) VALUE ZEROS.
01 TOTAL-ZON                     PIC S9(9)V9(2) VALUE ZEROS.

01 STATUS-ERR                   PIC XX.
01 ORDERNBR                      PIC X(5).

*==================================================================*
* This is the area to receive the New record image                  *
*==================================================================*
01 NEW-ORDER.
   03 NORDHNBR                  PIC X(5).
   03 NCUSNBR                   PIC X(5).
   03 NORHDTE                   PIC X(10).
   03 NORHDLY                   PIC X(10).
   03 NORDTOT                   PIC S9(9)V9(2) COMP-3.
   03 NORHSR                      PIC X(10).

*==================================================================*
* This is the area to receive the Old record image                  *
*==================================================================*
01 OLD-ORDER.
   03 ORDHNBR                   PIC X(5).
   03 CUSNBR                     PIC X(5).
   03 ORHDTE                     PIC X(10).
   03 ORHDLY                     PIC X(10).
   03 ORDTOT                     PIC S9(9)V9(2) COMP-3.
   03 ORHSR                      PIC X(10).

01 JOBA-AREA.
   03 BYTES-RTN                  PIC 9(8) BINARY VALUE 0.
   03 BYTES-AVAIL              PIC 9(8) BINARY VALUE 0.
   03 JOBNUMBER                  PIC X(6).
   03 USERNAME                   PIC X(10).
   03 RTV-JOBA.
      03 RTV-JOB-VAR             PIC X(50).
      03 RTV-JOB-LEN             PIC 9(8) BINARY VALUE 50.
      03 RTV-JOB-FMT             PIC X(8) VALUE "JOB1O400".
      03 RTV-JOB-NAME            PIC X(26) VALUE "**".
      03 RTV-JOB-ID                     PIC X(16) VALUE "".

* Parameter passed to the API QUSRJOBI to retrieve*   
* the job attributes                                    *
*==================================================================*
*----------------------------------------------------------------------*
* COBOL ERROR HANDLER routine to treat severe errors as MCHXXXX          *
*----------------------------------------------------------------------*
01 ERROR-HDL.
  03 ERR-HDL-EXIT           PIC X(20) VALUE "T4249CHDL ORDENTLIB".
  03 ERR-HDL-SCOPE          PIC X(1) VALUE "C".
  03 ERR-HDL-PGML           PIC X(10) VALUE " ".
  03 ERR-HDL-PGMN           PIC X(20) VALUE "T4249CINV ORDENTLIB".
  03 ERR-HDL-CODE.
    05 PROV                   PIC 9(8) BINARY VALUE 66.
    05 AVAIL                  PIC 9(8) BINARY VALUE 0.
    05 EXCEP-ID              PIC X(7) VALUE "           ".
    05 FILLER                 PIC X(1) VALUE " ".
    05 EXCEP-DATA            PIC X(50) VALUE " ".

*----------------------------------------------------------------------*
* Signalling the exception inside trigger                              *
*----------------------------------------------------------------------*
01 SNDPGMMSG.
  03 SND-MSG-ID                PIC X(7) VALUE "TRG0005".
  03 SND-MSG-FILE             PIC X(20) VALUE "ORDMSGF     ORDENTLIB".
  03 SND-MSG-DATA             PIC X(30) VALUE "TRIGGER ERROR        ".
  03 SND-MSG-LEN              PIC 9(8) BINARY VALUE 0.
  03 SND-MSG-TYPE             PIC X(10) VALUE "*ESCAPE     ".
  03 SND-MSG-QUEUE           PIC X(10) VALUE "".
  03 SND-PGM-STACK           PIC 9(8) BINARY VALUE 1.
  03 SND-MSG-KEY              PIC X(4) VALUE " ".
  03 SND-ERROR-CODE.
    05 PROVIDED              PIC 9(8) BINARY VALUE 66.
    05 AVAILABLE             PIC 9(8) BINARY VALUE 0.
    05 EXCEPTION-ID        PIC X(7) VALUE "           ".
    05 FILLER                  PIC X(1) VALUE " ".
    05 EXCEPTION-DATA     PIC X(50) VALUE " ".

*----------------------------------------------------------------------*
* PARM 1 = TRIGGER BUFFER                                              *
* PARM 2 = TRIGGER LENGTH                                              *
*----------------------------------------------------------------------*
LINKAGE SECTION.
01 PARM-1.
  03 FILE-NAME                  PIC X(10).
  03 LIB-NAME                   PIC X(10).
  03 MEM-NAME                   PIC X(10).
  03 TRG-EVENT                  PIC X.
  03 TRG-TIME                   PIC X.
  03 CMT-LCK-LVL              PIC X.
  03 FILLER                      PIC X(3).
  03 DATA-AREA-CCSID        PIC 9(8) BINARY.
  03 FILLER                      PIC X(8).
  03 DATA-OFFSET.
    05 OLD-REC-OFF          PIC 9(8) BINARY.
    05 OLD-REC-LEN          PIC 9(8) BINARY.
    05 OLD-REC-NULL-MAP PIC 9(8) BINARY.
    05 OLD-REC-NULL-LEN PIC 9(8) BINARY.
    05 NEW-REC-OFF          PIC 9(8) BINARY.
    05 NEW-REC-LEN          PIC 9(8) BINARY.
    05 NEW-REC-NULL-MAP PIC 9(8) BINARY.
PROCEDURE DIVISION USING PARM-1, PARM-2.

DECLARATIVES.
TRANSACTION-ERROR SECTION.
USE AFTER STANDARD ERROR PROCEDURE ON T4249INV ORDERDTL.
ERROR-HANDLER.
CLOSE T4249INV ORDERDTL.
CALL "QMHSNDPM" USING SND-MSG-ID,
    SND-MSG-FILE,
    SND-MSG-DATA,
    SND-MSG-LEN,
    SND-MSG-TYPE,
    SND-MSG-QUEUE,
    SND-PGM-STACK,
    SND-MSG-KEY,
    SND-ERROR-CODE.

IF AVAILABLE IS NOT EQUAL 0 THEN
    DISPLAY "ERROR - QMHSNDPM API" SND-MSG-ID.
GOBACK.
END DECLARATIVES.
MAIN-PROGRAM SECTION.
START-SECTION.

OPEN OUTPUT T4249INV INPUT ORDERDTL.
MOVE ZEROS TO NORDTOT OF NEW-ORDER
ORDTOT OF OLD-ORDER TOTAL-ZON TOTAL.

*=================================================================
* This is the new record image.
*=================================================================
MOVE NEW-RECORD TO NEW-ORDER.

*=================================================================
* This is the old record image.
*=================================================================
MOVE OLD-RECORD TO OLD-ORDER.

* Call API COBOL ERROR HANDLER - If MCHXXXX occurs, *
* the routine associated with this API will be called *
*=================================================================

CALL "QLRSETCE" USING ERR-HDL-EXIT,
ERR-HDL-SCOPE,
ERR-HDL-PGML,
ERR-HDL-PGMN,
ERR-HDL-CODE.

*====================================================================*
* Call API to get job attributes - USERID *
*====================================================================*

CALL "QUSRJOBI" USING RTV-JOB-VAR,
RTV-JOB-LEN,
RTV-JOB-FMT,
RTV-JOB-NAME,
RTV-JOB-ID.

MOVE RTV-JOB-VAR TO JOBA-AREA.

*====================================================================*
* This is a BEFORE UPDATE trigger program associated to ORDERHDR file. This program will check: *
* - The update is being made by the correct salesperson *
* or by QSECOFR, otherwise trigger will signal an exception and will stop the update operation *
* *
* - QSECOFR can update all values but will not print the invoice. The salesperson can update all the fields but CUSTOMER_NUMBER, and in this case will print the invoice *
* *
* - This provides an example of how to handle both record images inside trigger program. *
*====================================================================*

IF USERNAME NOT EQUAL "QSECOFR" THEN
  IF NORHSR OF NEW-ORDER EQUAL USERNAME AND NCUSNBR OF NEW-ORDER EQUAL CUSNBR OF OLD-ORDER THEN
    PERFORM HEADER-LINE
    PERFORM DETAIL-LINE UNTIL EOF
  ELSE
    MOVE "TRG0002" TO SND-MSG-ID
    PERFORM ERROR-HANDLER.
  END-IF
END-IF

CLOSE T4249INV ORDERDTL.
GOBACK.

HEADER-LINE.
  MOVE NORDHNBR OF NEW-ORDER TO ORHNBR OF HEADER-O.
  MOVE NCUSNBR OF NEW-ORDER TO CUSNBR OF HEADER-O.
  MOVE NORDHDE OF NEW-ORDER TO ORHDTE OF HEADER-O.
  MOVE NORDTOT OF NEW-ORDER TO TOTAL-ZON.
  MOVE TOTAL-ZON TO ORHTOT OF HEADER-O.
  MOVE NORHSR OF NEW-ORDER TO SRNBR OF HEADER-O.
  WRITE PRT-REC FORMAT IS "HEADER".
  MOVE NORDHNBR OF NEW-ORDER TO ORHNBR OF ORDERDTL.
  MOVE NORDHNBR OF NEW-ORDER TO ORHNBR OF ORDERDTL.

DETAIL-LINE.
  READ ORDERDTL NEXT RECORD AT END SET EOF TO TRUE.
  IF ORHNBR OF ORDERDTL EQUAL ORDERNBR AND NOT EOF THEN
    MOVE PRDNBR OF ORDERDTL TO PRDNBR OF DETAIL-O
    MOVE ORDTOT OF ORDERDTL TO TOTAL-
MOVE TOTAL TO ORDTOT OF DETAIL-O
MOVE ORDQTY OF ORDERDTL TO ORDQTY OF DETAIL-O
WRITE PRT-REC FORMAT IS "DETAIL".

Exception Handler for T4249CINV - T4249CHDL
PROCESS OPTIONS.
IDENTIFICATION DIVISION.
PROGRAM-ID. T4249CHDL.
AUTHOR. PROGRAMMER NAME.
INSTALLATION. ITSC LABORATORY.
DATE-COMPILED.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-AS400.
OBJECT-COMPUTER. IBM-AS400.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

**********************************************************************
DATA DIVISION.
**********************************************************************

WORKING-STORAGE SECTION.
*********************************************************************************
* Message for signalling trigger error                      *
* This is an escape message send to the calling program.   *
* The objective is signalling the database manager the       *
* change operation must not happen                           *
*********************************************************************************

01 SNDPGMMSG.
   03 SND-MSG-ID          PIC X(7) VALUE "TRG0005".
   03 SND-MSG-FILE        PIC X(20) VALUE "ORDMSGF     ORDENTLIB".
   03 SND-MSG-DATA        PIC X(30) VALUE "TRIGGER ERROR        ".
   03 SND-MSG-LEN         PIC 9(8) BINARY VALUE 0.
   03 SND-MSG-TYPE        PIC X(10) VALUE "*ESCAPE     ".
   03 SND-MSG-QUEUE       PIC X(10).
   03 SND-MSG-LEN         PIC 9(8) BINARY VALUE 1.
   03 SND-MSG-KEY         PIC X(4) VALUE "      ".
   03 SND-ERROR-CODE.
      05 PROVIDED        PIC 9(8) BINARY VALUE 66.
      05 AVAILABLE       PIC 9(8) BINARY VALUE 0.
      05 EXCEPTION-ID    PIC X(7) VALUE "           ".
      05 FILLER          PIC X(1) VALUE " ".
      05 EXCEPTION-DATA  PIC X(50) VALUE " ".

LINKAGE SECTION.
*********************************************************************************
*This is the parameter list expected by the program that        *
*activated the QLRSETCE API.                                     *
*********************************************************************************

01 MSG-RCV-ID          PIC X(7).
01 MSG-RCV-RSP         PIC X(6).
01 MSG-RCV-PGMN        PIC X(20).
01 MSG-RCV-SMSG        PIC X(7).
01 MSG-RCV-TMSG        PIC X(50).
01 MSG-RCV-LENG   PIC X(2).
01 MSG-RCV-CODE   PIC X(1).

**********************************************************
PROCEDURE DIVISION USING MSG-RCV-ID,
   MSG-RCV-RSP
   MSG-RCV-PGMN,
   MSG-RCV-SMSG,
   MSG-RCV-TMSG,
   MSG-RCV-LENG,
   MSG-RCV-CODE.

***********************************************************
* Set the program message queue, received by the calling *
* program, telling to which message queue we should signal*
* the escape message                                      *
***********************************************************
MOVE MSG-RCV-PGMN TO SND-MSG-QUEUE.

***********************************************************
* Signalling the escape message - DB change operation     *
* will be rejected                                        *
***********************************************************
CALL "QMHSNDPM" USING SND-MSG-ID,
   SND-MSG-FILE,
   SND-MSG-DATA,
   SND-MSG-LEN,
   SND-MSG-TYPE,
   SND-MSG-QUEUE,
   SND-PGM-STACK,
   SND-MSG-KEY,
   SND-ERROR-CODE.

IF AVAILABLE IS NOT EQUAL 0   THEN
   DISPLAY "API ERROR CBHDL" SND-MSG-ID.

STOP RUN.

Invoice trigger example in ILE RPG
In this version of the trigger program, we generate the invoice information in a database file
rather than printing the invoice directly. The layout of the INVOICE file is the same as the
DETAIL record format of the printer file in the previous paragraph (Figure 11-28 on page 399).
We simply added one field—the order number (ORDNBR).

The example shows how you can dynamically put a database file under commitment control
in ILE RPG. If the originating application runs under commitment control, the invoice trigger
has to become part of the application transaction. The application has to be able to commit or
roll back all the records that the trigger inserts in the invoice file.

To accomplish this, we can use the dynamic commitment definition in the F specifications
provided by ILE RPG. The keyword we need to specify is COMMIT(variable-name). We also
need to specify the USROPN keyword, because the COMMIT keyword takes effect only when
the file is opened. We test the CMTLCK field in the trigger buffer data structure and set the
RPG variable to the proper value based on the commitment control lock level of the
application. The file is opened manually with the correct commitment definition.
Because we want to let the application take care of committing or rolling back the entire transaction, the trigger must share the same commitment definition as the application. The trigger runs in the same activation group and does not issue any commit or rollback statements.

To create this trigger program, follow these two steps:

```sql
CRTRPGMOD  MODULE(ORDENTLIB/T4249IINV)
SRCFILE(ORDENTLIB/QRPGILE)
SRCMBR(T4249IINV)

CRTPGM      PGM(ORDENTLIB/T4249IINV)
ACTGRP(*CALLER)

Invoice trigger in ILE RPG - T4249IINV
```

```
*====================================================================*
* This is a BEFORE UPDATE trigger program associated to*            *
* ORDERHDR file. This program will check that:                    *
* - The update is being made by the correct salesperson*          *
*   or by QSECOFR, otherwise trigger will signal an             *
*   exception and will stop the update operation                *
* - QSECOFR can update all values but will not print            *
*   the invoice. The salesperson can update all the             *
*   fields but CUSTOMER_NUMBER, and in this case will           *
*   print the invoice                                           *
* - This provides an example of how to handle both               *
*   record images inside trigger program.                       *
*====================================================================*

FORDERDTL  IF   E           K DISK    INFDS(FILDS1)
F                                     INFSR(*PSSR)
F                                     RENAME(ORDERDTL:ORDDET)

*====================================================================*
* The RPG variable VAR will determine whether this file will      *
* be opened under commitment control or not. We use the          *
* explicit open option (USROPN) to set the correct value of the* *
* variable VAR before the file is opened.                        *
*====================================================================*

FINVOICE   O  A E             DISK    INFDS(FILDS2)
F                                     INFSR(*PSSR)
F                                     USROPN
F                                     COMMIT(VAR)
F                                     RENAME(INVOICE:DETAIL)

*====================================================================*
* Exception handling in RPG trigger                              *
*====================================================================*

DVAR S 1A
D FILDS1 DS
D FIL1  *FILE
D REC1  *RECORD
D OP1   *OPCODE
D STS1  *STATUS
D RTN1  *ROUTINE
D FILDS2 DS
D FIL2  *FILE
D REC2  *RECORD
D OP2   *OPCODE
D STS2  *STATUS
```
D  RTN2             *ROUTINE
*====================================================================*
* Definition of the structure to be received by the trigger program - Buffer *
*====================================================================*
* THE FIELDS DESCRIPTION:*
* FNAME = PHYSICAL FILE NAME *
* LNAME = PHYSICAL FILE LIBRARY *
* MNAME = MEMBER NAME *
* TEVEN = TRIGGER EVENT *
* TTIME = TRIGGER TIME *
* CMTLCK = COMMIT LOCK LEVEL *
* FILL1 = RESERVED *
* CCSID = CCSID *
* FILL2 = RESERVED *
* OLDOFF = OFFSET TO THE ORIGINAL RECORD *
* OLDLEN = LENGTH OF THE ORIGINAL RECORD *
* ONOFF = OFFSET TO THE ORIGINAL RECORD NULL BYTE MAP *
* ONLEN = LENGTH OF THE NULL BYTE MAP *
* NOFF = OFFSET TO THE NEW RECORD *
* NEWLEN = LENGTH OF THE NEW RECORD *
* NNOFF = OFFSET TO THE NEW RECORD NULL BYTE MAP *
* NNLEN = LENGTH OF THE NULL BYTE MAP *
* RESV3 = RESERVED *
* OREC = OLD RECORD *
* OOMAP = NULL BYTE MAP OF OLD RECORD *
* RECORD = NEW RECORD *
* NMAP = NULL BYTE MAP OF NEW RECORD *
*====================================================================*

D PARM1           DS
D  FNAME                  1     10
D  LNAME                 11     20
D  MNAME                 21     30
D  TEVEN                 31     31
D  TTIME                 32     32
D  CMTLCK                33     33
D  FILL1                 34     36
D  CCSID                 37     40B 0
D  FILL2                 41     48
D  OLDOFF                49     52B 0
D  OLDLEN                53     56B 0
D  ONOFF                 57     60B 0
D  ONLEN                 61     64B 0
D  NOFF                  65     68B 0
D  NEWLEN                69     72B 0
D  NNOFF                 73     76B 0
D  NNLEN                 77     80B 0
D  RESV3                 81     96
D  OREC                  97    142
D  OOMAP                143    148
D  RECORD               149    194
D  NMAP                  195    200
*====================================================================*
* Definition of the structure to be received by the trigger program - BUFFER LENGTH *
*====================================================================*

D PARM2           DS
D  LENG                   1      4B 0
*
Chapter 11. External triggers

* These are the work fields to receive the trigger *
* new record image *
*==================================================*
D NORDER          DS
D ORHNBR                 1      5
D CUSNBR                 6     10
D ORHDTE                11     20
D ORHDL                 21     30
D OTOTAL                31     36P 0
D SRNBR                 37     46
*==================================================*

* These are the work fields to receive the trigger *
* old record image *
*==================================================*
D OORDER          DS
D OORHNB                 1      5
D OCUSNB                 6     10
D OORHDT                11     20
D OORHDL                21     30
D OOTOTA                31     36P 0
D OSRNBR                37     46
*==================================================*

* This is the work area that will receive the *
* job attributes retrieved *
*==================================================*
D RTVAPI          DS
D BTRN                   1      4B 0
D BAVAIL                 5      8B 0
D JOBNAM                 9     18
D USERID                19     28
D JOBNBR                29     34
*==================================================*

* Output parameters used in QMHSNDPM API *
*==================================================*
D MSGERR          DS
D PROVID                 1      4B 0
D AVAIL                  5      8B 0
D RTNMSG                9     15
D RSVR                  16     16
D RTNDTA                17     26
*

D FLDS            DS
D MSGLEN                 1      4B 0
D PGMSMTK                5     8B 0
D RTVLEN                  9    12B 0
D MSGQLEN               13    16B 0
D PGMWTT                17    20B 0
*=============================================================*

* RPG ILE - Call stack entry - signal exceptions *
*=============================================================*
D LIBNAM          C                   CONST('ORDENTLIB')
D MSGQNAM         C                   CONST('_QRNP_PEP_T4249IINV')
D MODNAME         C                   CONST('*NONE     *NONE     '*)
*
C     *ENTRY        PLIST
C     PARM1         PARM                    PARM1
C     PARM2         PARM                    PARM2
*

* Signalling exception inside trigger program *

* *** End of external trigger ***
*=====================================================*
C     PLIST1        PLIST
C                   PARM                    MSGID             7
C                   PARM                    MSGF               20
C                   PARM                    MSGDTA             25
C                   PARM                    MSGLEN
C                   PARM                    MSGTP              10
C                   PARM                    MSGQUE              19
C                   PARM                    MSGERR
C                   PARM                    MSGQLEN
C                   PARM                    CSEQUAL            20
C                   PARM                    PGMWTT
*C     KEYS          KLIST
C                   KFLD                    ORHNBR
*=====================================================
C Retrieve job attributes - QUSRJOBI API - USERID *
*=====================================================
C     PLIST2        PLIST
C                   PARM                    RTVVAR            50
C                   PARM                    RTVLEN
C                   PARM                    RTVFMT             8
C                   PARM                    RTVFNM            26
C                   PARM                    RTVFID            16
C CMTLCK           IFNE      '0'
C                     MOVEL     '1'          VAR
C                     ELSE
C                     MOVEL     *BLANKS      VAR
C                     '*NO'         dsply
C                     END
C*=====================================================
C Initialize MSGID in case the PSSR is called. *
C*=====================================================
C                   MOVE      'TRG0005'     MSGID
C*=====================================================
C Retrieve job attributes - USERID *
C*=====================================================
C                   Z-ADD     50             RTVLEN
C                   MOVEL     ' '          RTVFNM
C                   MOVE      'JOBI0400'  RTVFMT
C                   MOVEL(P)  '*'          RTVFNM
C                   MOVE      ' '          RTVFID

*=====================================================
C Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries
Chapter 11. External Triggers

C CALL 'QUSRJOBI' PLIST2
C MOVEL RTVVAR RTVAPI
C SRNBR IFEQ 'QSECOFR'
C GOTO EOFIN
C END
C SRNBR IFNE USERID
C MOVE 'TRG0002' MSGID
C EXSR *PSSR
C GOTO EOFIN
C END

*
C CUSNBR IFEQ OCUSNB
C OPEN INVOICE
C MOVEL ORHNBR ORDNBR
C KEYS SETLL ORDDET
C *IN99 DOUEQ *ON
C KEYS READE ORDDET *IN99
C *IN81 IFEQ *ON
C EXSR *PSSR
C GOTO EOFIN
C END
C *IN99 IFEQ *OFF
C WRITE DETAIL *IN99
C *IN81 IFEQ *OFF
C EXSR *PSSR
C GOTO EOFIN
C END
C END
C ENDDO
C EOFIN TAG
C close invoice
C RETURN

*=================================================================
* Trigger signalling exception
*=================================================================

C *PSSR BEGSR
C MOVEL(P) LIBNAM LIB 10
C MOVEL(P) 'ORDMSGF' ID 10
C ID CAT(P) LIB MSGF
C MOVE ' ' MSGDTA
C Z-ADD 25 MSGLEN
C MOVEL(P) '*ESCAPE' MSGTP
C MOVEL(P) MSGQNAME MSGQUE
C MOVE MODNAME CSEQUAL
C Z-ADD 19 MSGQLEN
C MOVE ' ' MSGDTA
C Z-ADD 1 PGMSTK
C MOVE ' ' MSGKEY
C Z-ADD 66 PROVID
C Z-ADD 0 AVAIL
C MOVE ' ' RTNMSG
C MOVE ' ' RSVR
C MOVE ' ' RTNDTA
C CALL 'QMHSNDPM' PLIST1
C AVAIL IFEQ 0
C RTNMSG DSPLY
C RTNDTA DSPLY
C END
C ENDSR
Invoice trigger example in ILE C

We coded the same trigger program in ILE C to show how you can manage the dynamic commitment control using this language. See “Invoice trigger example in ILE RPG” on page 406 for a discussion on commitment control in this trigger. The _Ropen function allows you to specify whether the file has to be opened under commitment control at run time.

Create this program with the following commands:

CRTCMOD MODULE(ORDENTLIB/T4249CCIV)
   SRCFILE(ORDENTLIB/QCSRC)
CRTPGM PGM(ORDENTLIB/T4249CCIV)
   ACTGRP(*CALLER)

Invoice trigger in ILE C - T4249CCIV

/*................................................................*/
/*. This is a BEFORE UPDATE trigger program associated with the */
/*. ORDERHDR file. The program checks that:                      */
/*.   - The update is being made by the originator of the order  */
/*.      or by QSECOFR; in any other case, the trigger stops the */
/*.      update.                                                 */
/*.   - If the originator of the order updates the grand total, */
/*.      the invoice is also generated. The originator will not */
/*.      be able to update the CUSTOMER_NUMBER field. QSECOFR */
/*.      has the ability to update any field, but no invoice is */
/*.      generated.                                              */
/*................................................................*/
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#include  <decimal.h>
#include  <string.h>
#include  <signal.h>

/*..... Externally described files .....*/
#pragma mapinc("InvoiceFile", "ORDENTL/INVOICE(*ALL)",
              "both", "d z _P", " ", "inv")
#include "InvoiceFile"

#pragma mapinc("OrderDetail", "ORDENTL/ORDERDTL(*ALL)",
              "input key", "d z _P", " ", "dtl")
#include "OrderDetail"

/*..... APIs linkage .....*/
#pragma linkage(QUSRJOBI, OS)
#pragma linkage(QMHSNDPM, OS)
/*..... We defined our own trigger buffer; you can also include */
/*..... the system provided definition QSYSINC/H/TRGBUF ..........*/
typedef _Packed struct {
   char FileName[10];
   char LibName[10];
   char MbrName[10];
   char TrgEvent[1];
   char TrgTime[1];
   char CmtCtlLvl[1];
   char reserved1[3];
   int   CCSID;
   char reserved2[8];
}
int OldRecordOffset;
int OldRecordLength;
int OldRecordNullByteMapOffset;
int OldRecordNullByteMapLength;
int NewRecordOffset;
int NewRecordLength;
int NewRecordNullByteMapOffset;
int NewRecordNullByteMapLength;
} TrgBuf;

typedef _Packed struct { /*.... Record layout ....*/
    char OrderNumber[5];
    char CustomerNumber[5];
    char OrderDate[10];
    char OrderDelivery[10];
    decimal(11,2) OrderTotal;
    char SalesRep[10];
} DBFileRec;

void QUSRJOBI(char *, int, char *, char *, char *);
void QMHSNDPM(char *, char *, char *, int, char *, char *, int, char *, char *);
void sendMessage(char *);

int checkUpdateValidity(DBFileRec *, DBFileRec *);
/*.... This procedure returns:
....   -1 if the originator is trying to update CUSTOMER_NUMBER
....   0 if the originator is not updating the grand total
....   1 if the originator is updating the grand total
..............................................................*/
DBFileRec *NewOrder, *OldOrder; /*.... Old and New Image ....*/
TrgBuf *TrgBuffer;

inv_INVOICE_both_t Invoice;
dtl_ORDERDTL_i_t OrderDetail;
dtl_ORDERDTL_key_t OrderDetailKey;
static _RFILE *Inv, *OrderDtl;

void main(int argc, char **argv)
{
    _RIOFB_T *InvFB, *OrderDtlFB;
    char JobInfo[86]; /*.... Parameters for QUSRJOBI ....*/
    int JobInfoLen = 86;
    char JobFmt[8] = "JOB10100";
    char JobName[26];
    char JobId[16];
    char UserId[10], MsgId[7];
    double OrderTotalD, OrderQtyD, PartialTotalD;
    int UpdateType;
    void ExcptHandler(int);
    signal(SIGALL, ExcptHandler);
    memset(JobName, ' ', 26);
Field: 414

JobName[0] = '*';
memset(JobId, ' ', 16);

TrgBuffer = (TrgBuf *) argv[1];

/* .... Setting the pointers to the storage areas where the
 system keeps the record images ......................*/
OldOrder = (DBFileRec *) (char *) TrgBuffer +
TrgBuffer->OldRecordOffset;
NewOrder = (DBFileRec *) (char *) TrgBuffer +
TrgBuffer->NewRecordOffset;

/* .... Retrieving the current USERID and checking if this is the
 same who actually issued the order .................*/
QUSRJOBI(JobInfo, JobInfoLen, JobFmt, JobName, JobId);
memcpy(UserId, JobInfo+18, 10);
if(!strncmp(UserId, "QSECOFR   ", 10))
  return; /* .... User is QSECOFR, no further checking or action ...*/
if(strncmp(UserId, OldOrder->SalesRep, 10) ||
  (UpdateType = checkUpdateValidity(OldOrder, NewOrder)) == -1)
  {
    memcpy(MsgId, "TRG0002", 7);
    sendMessage(MsgId);
    return;
  }
if (UpdateType == 0)
  return;
if (*TrgBuffer->CmtCtlLvl == '0')
  /* .... If the application runs without commitment control ....*/
  /* .... open the invoice file with the commit option set to NO ....*/
  Inv = _Ropen("ORDENTL/INVOICE", "ar commit=N");
else /* .... otherwise, open the file with commit YES ....*/
  Inv = _Ropen("ORDENTL/INVOICE", "ar commit=Y");

/* .... Scanning the Order to produce the Invoice ....*/
memcpy(OrderDetailKey.ORDER_NUMBER, NewOrder->OrderNumber, 5);

/* .... File OrderDtl is opened only if the trigger is being invoked
 .... for the first time. .................................................................*/
if (OrderDtl == NULL)
  OrderDtl = _Ropen("ORDENTL/ORDERDTL", "rr arrseq=N");
OrderDtlFB = _Rlocate(OrderDtl, (void *) OrderDetailKey.ORDER_NUMBER,
  5, __KEY_EQ|__DATA_ONLY);
if (OrderDtlFB->num_bytes == 0)
  exit(1);
OrderDtlFB =
  _Rreads(OrderDtl, (void *) &OrderDetail, sizeof(OrderDetail),
  __NO_LOCK);
strcpy(Invoice.ORDNBR, NewOrder->OrderNumber, 5);
while(OrderDtlFB->num_bytes != EOF &&
      !strncmp(OrderDetail.ORDER_NUMBER, NewOrder->OrderNumber, 5))
{
    memcpy(Invoice.PRDNBR, OrderDetail.PRODUCT_NUMBER, 5);
    Invoice.ORDQTY = OrderDetail.ORDERDTL_QUANTITY;
    Invoice.ORDTOT = OrderDetail.ORDERDTL_TOTAL;
    InvFB = _Rwrite(Inv, (void *) &Invoice, sizeof(Invoice));
    OrderDtlFB =
                _Rreadn(OrderDtl, (void *) &OrderDetail, sizeof(OrderDetail),
                __NO_LOCK);
    _Rclose(Inv);
}

int checkUpdateValidity(DBFileRec *OldOrder, DBFileRec *NewOrder)
{
    if (strncmp(NewOrder->CustomerNumber, OldOrder->CustomerNumber, 5))
        return -1; /*... Violation ....*/
    if (NewOrder->OrderTotal != OldOrder->OrderTotal)
        return 1;   /*... Print Invoice ....*/
    return 0;        /*... No action ........*/
}

void sendMessage(char *MsgId)
{
    char MsgFile[20] = "ORDMSGF       ORDENTLIB ",
                     MsgData[30] = "Trigger Error                      ",
                     MsgType[10] = "*ESCAPE          ",
                     MsgQueue[10]= "_C_pep            ",
                     MsgKey[4]     = "      ";

    struct {
        int provided;
        int available;
        char Excp[7];
        char filler;
    } ErrorCode;

    int MsgLen = 0, PgmStack = 1;

    ErrorCode.provided = 8;
    QMHSNDPM(MsgId, MsgFile, MsgData, MsgLen, MsgType, MsgQueue,
             PgmStack, MsgKey, (char *) &ErrorCode);
    if(ErrorCode.available)
    {
        printf("Error: %7.7s\n", ErrorCode.Excp);
        exit(1);
    }
}
void ExcptHandler(int sig)
{
    char MsgId[7];
    memcpy(MsgId, "TRG0005", 7);
    sendMessage(MsgId); /*... Send generic trigger error message ....*/
    exit(0);
}

11.4.4 Soft coding the trigger buffer example

The example of trigger programs shown in this section illustrates two important concepts:

- Soft coding the trigger buffer
- Changing the trigger buffer

In all of the preceding examples, you saw that the structure of the table for which the trigger program is being written is hardcoded inside the trigger program. This means that, if for some reason you change the structure of the physical file, you also need to change the structure of the physical file inside the trigger program. This must be repeated for all the trigger programs where the structure of the related table has changed. The alternative is that, at the time of writing the trigger program, you do not hard code the structure of the table, but soft code it. That way, if and when you change the structure of a table, you only need to recompile the trigger program and not make any changes to the trigger program itself.

Soft coding the trigger buffer is a good programming technique because a change in the physical file's record can be incorporated by simply recompiling the trigger program.

Note: The sample code for the soft-coded trigger buffer is based on code that originally was published in Chapter 17 of Database Design and Programming for DB2/400 by Paul Conte. You can find an improved technique of this approach on pages 320 and 323 in SQL/400 Developer's Guide by Paul Conte and Mike Cravitz.

Soft coding the trigger buffer in ILE RPG

Each customer in our database was assigned a credit limit. This limit is the maximum dollar amount that the customer can order in one month. We want to notify the customers that they are nearing their monthly credit limit to let them have control over their orders or to apply to increase their credit limit. The notification should be sent through a fax when the monthly total for a customer amounts to more than 90 percent of the credit limit. If the customer is a special customer, which is recognized because its customer number starts with a 9, the trigger program automatically increases the credit limit by 30 percent. This involves changing the record that activated the trigger.

If you are dealing with an existing application, integrating this new function without using triggers involves some modifications to the application code. In a composite environment, where multiple applications work on the same data, this process might be complex and costly. Moreover, if you plan to move from a host-based to a client-server environment, also move the logic for the advanced technology integration on the client platform.

By using triggers, this accomplishment can be carried out with almost no change to existing applications. Make sure that your programs are monitoring the return code after the database access, which is common practice among application developers to avoid a failure of the trigger leading to an abnormal end of the application.
The example program we provide monitors the update operations on the CUSTOMER file. The fax number of the customer is checked whenever the total amount of the purchases is increased. A blank fax number means that the customer is not provided with this facility. In this case, an informational message is sent to the job log, but the update occurs anyway.

In our application, this trigger comes up again during the final order information processing performed by the program FNLOD shown in “Finalize order program - T4249FNLO” on page 397. When the customer information is updated to keep track of the monthly balance of this specific customer, the trigger performs the appropriate checking and possibly sends the fax to the customer.

Note: We do not show, for the sake of simplicity, the program that actually formats the document and sends the fax to the customer. The program SENDFAX invoked in this example is not shown in this book.

To create this trigger program, enter the following commands:

```
CRTRPGMOD  MODULE(ORDENTLIB/T4249CTA)
         SRCFILE(ORDENTLIB/QRPGILE)
         SCRMBR(T4249CTA)

CRTPGM      PGM(ORDENTLIB/T4249CTA)
         ACTGRP(*CALLER)
```

**Data structure with trigger buffer - T4249BUF**

```
********************************************************
*   TRIGGER BUFFER data structure
********************************************************
D TgBufDs              DS
D   TgFile                                             Like( TypeSysNam )
D   TgLib                                              Like( TypeSysNam )
D   TgMbr                                              Like( TypeSysNam )
D   TgTrgEvt                                           Like( TypeChr )
D   TgTrgTime                                          Like( TypeChr )
D   TgCmtLvl                                           Like( TypeChr )
D   TgReserve1                                3A
D   TgCcsId                                            Like( TypeBin4 )
D   TgReserve2                                8A
D   TgBufChr                        1   32767A
D   TgBufAry                              1A     Overlay( TgBufChr )
D                                                  DIM ( %Size( TgBufChr ) )
D
D*   End of TbBufDs
```
Notes: The following notes refer to the previous example.

1 The TgBufDs data structure defines the trigger buffer. It is coded in a separate member so it can be referenced by all of the ILE RPG trigger programs.

2 The TgBufChr field is the variable part of the trigger buffer. It is declared as a single character field with the maximum size allowed, which is 32767. At the same time, it is redefined as an array of bytes using another subfield called TbBufAry.

ILE RPG trigger program to send a fax - T4249CTA

********************************************************************
* This is the trigger program using the technique of
* Softcoding the trigger buffer.
********************************************************************

********************************************************************
* Some standard data type definitions
********************************************************************
D NulTypePtr     S               *
D TypeBin4       S              9B 0 Based( NulTypePtr )
D TypeChr        S              1A   Based( NulTypePtr )
D TypeSysNam     S             10A   Based( NulTypePtr )
D TypePtr        S               *   Based( NulTypePtr )
********************************************************************

* TRIGGER BUFFER
* This is the declaration of the trigger buffer
* It copies the structure from the TRIGBUF member
********************************************************************
/COPY T4249TBUF
********************************************************************
* Declarations of the Buffer Length and Pointers to the
* After and Before Images
********************************************************************
D TgBufLen       S                   Like( TypeBin4 )
D TgBfrPtr       S                   Like( TypePtr )
D TgAftPtr       S                   Like( TypePtr )
D TgBufSiz       C                   Const( %Size( TgBufChr ) )
********************************************************************
* Data Structures for the Before and After images
********************************************************************
D BfCustomer    E DS                  ExtName( Customer )
D                                   Prefix( Bf )
D                                   Based( TgBfrPtr )
D AfCustomer    E DS                  ExtName( Customer )
D                                   Prefix( Af )
D                                   Based( TgAftPtr )
********************************************************************
* OUTPUT PARAMETERS FOR QMHSENDPM
********************************************************************
D ERROR           DS
D PROVID                 1      4B 0
D AVAIL                  5      8B 0
D RTNMSG                 9     15
D MSGD            DS
D MSGLEN                 1      4B 0
Chapter 11. External triggers

D PGMSTK 5 88 0
*
D LIBNAM C "CONST('ORDAPPLIB')"
*
******************************************************************************
C* TWO PARAMETERS GO INTO THE TRIGGER PROGRAM
******************************************************************************
C | ENTRY         PLIST
C | TgBufDs       PARM TgBufDs
C | TgBufLen      PARM TgBufLen
******************************************************************************
C | PARAMETERS NEEDED TO SIGNAL AN EXCEPTION INSIDE
C | TRIGGERS
******************************************************************************
C PLIST1 PLIST
C | PARM MSGID 7
C | PARM MSGF 20
C | PARM MSGDTA 30
C | PARM MSGLEN
C | PARM MSGTPY 10
C | PARM MSGQUE 10
C | PARM PGMSTK
C | PARM MSGKEY 4
C | PARM ERROR
******************************************************************************
C | PARAMETER CUSTOMER NUMBER TO SEND A FAX
C |
******************************************************************************
C PLIST2 PLIST
C | PARM CUSNBR 5
******************************************************************************
C | THIS TRIGGER PROGRAM WILL:
C | - RETURN IMMEDIATELY IF THE TOTAL AMOUNT IS NOT
C |   BEING UPDATED
C | - IF THE TOTAL AMOUNT IS BEING INCREASED AND REACHED
C |   90% OF THE CREDIT LIMIT
C |   * SEND A FAX TO THE CUSTOMER
C |   * IF THE CUSTOMER NUMBER STARTS WITH A 9 IS BECAUSE
C |   IT IS A VERY IMPORTANT CUSTOMER SO ITS CREDIT LIMIT
C |   IS INCREASED BY 30%. THIS REQUESTS TO CHANGE THE
C |   AFTER IMAGE BUFFER OF THE RECORD THAT FIRED THIS
C |   TRIGGER.
C | - IF THE FAX NUMBER IS *BLANKS, AN INFO MESSAGE
C |   IS SENT AND WILL BE FOUND IN THE JOB LOG.
******************************************************************************
C | LETS EVALUATE THE VALUE OF THE POINTERS FOR THE BEFORE
C | AND AFTER IMAGES
******************************************************************************
C | Eval TgBfrPtr = %Addr(TgBufAry(TgBfrOfs + 1))
C | Eval TgAftPtr = %Addr(TgBufAry(TgAftOfs + 1))
******************************************************************************
C | IF THE NEW TOTAL IS EQUAL OR LESS THAN THE OLD ONE,
C | GO BACK IMMEDIATELY
******************************************************************************
C AfCusTot IFLF BfCusTot
C RETURN
C ENDIF
******************************************************************************
C | IF THE NEW TOTAL IS EQUAL OR LESS THAN 90% OF THE
* CREDIT LIMIT, GO BACK IMMEDIATELY
****************************************************
C AfCusCrd MULT 0.90 TmCusCrd 11 2
C AfCusTot IFLE TmCusCrd
C RETURN
C ENDIF
****************************************************
* CHECK IF THE CUSTOMER IS A SPECIAL CUSTOMER,
* WHOSE CUSNBR STARTS WITH 9
****************************************************
C 1 SUBST BfCusNbr:1 TypCus 1
C TypCus IFEQ '9' AfCusCrd MULT 1.3 AfCusCrd
C ENDIF
***********************************************************
* CHECK THAT THE CUSTOMER HAS A FAX NUMBER TO CALL TO
***********************************************************
C AfCusFax IFNE *Blanks
C MOVE AfCusNbr CUSNBR
C CALL 'SENDFA' PLIST2
C ELSE
C MOVEL(P) 'TRG0004' MSGID
C MOVEL(P) LIBNAM LIB 10
C MOVEL(P) 'ORDMSGF' ID 10
C ID CAT(P) LIB MSGF
C Z-ADD 0 MSGLEN
C MOVEL(P) '*INFO' MSGTYP
C MOVEL(P) '*' MSGQUE
C MOVE ' ' MSGOTA
C Z-ADD 1 PGMSTK
C MOVE ' ' MSGKEY
C Z-ADD 0 PROVID
C Z-ADD 0 AVAIL
C CALL 'QMHSNDPM' PLIST1
C AVAIL IFNE 0
C 'APIER' DSPLY
C ENDIF
C ENDIF
C RETURN
Chapter 11. External triggers

Soft coding the trigger buffer in ILE C

This section shows how to code trigger programs in ILE C that use the trigger buffer soft-coding technique. Consider the following scenario.

Whenever a sales person tries to change a record in the ORDERHDR table or to insert a record into the ORDERHDR table, we need to check whether that sales person is authorized to deal with the customer specified in the record being inserted or updated. The sales person ID is assumed to be the user ID of the user who is inserting or updating the record. The information about the authority of the sales person dealing with the customer is specified in the SALESCUS table.

Therefore, we retrieve the user ID for the current job and check whether that user ID is authorized to deal with the given customer by querying the SALESCUS table.

To create the ILE C trigger program, submit the following commands:

```
crtcmop module(ordentlib/t4249ccat) srcfile(ordentlib/qcsrc)
crtpgm pgm(ordentlib/t4249ccat) actgrp(audit)
```

Audit trail trigger in ILE C - T4249CCAT

/*................................................................*/
/*. This is a BEFORE INSERT trigger program associated with the    */
/*. ORDERHDR file. The program checks that:                      */
/*.   - The orginator of the order (UserId) is allowed to place    */
/*.      an order for the customer (CUSTOMER_NUMBER).            */
/*.   - If this rule is not satisfied, the trigger logs the      */
/*.      violation attempt and causes the insert to fail.        */
/*................................................................*/
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

Notes: The following notes refer to the previous example.

1. The /COPY statement incorporates the trigger buffer data structure into the source member. This is a good programming technique because all of the trigger programs can use it.

2. This is the before image structure of the CUSTOMER record. It is an externally defined record structure with the same format as the CUSTOMER file. A prefix is used for every field name (Bf for the before image fields).

3. This is the after image structure of the CUSTOMER record. It is an externally defined record structure with the same format as the CUSTOMER file. A prefix is used for every field name (Af for the after image fields). This technique must be used for every file that has a trigger program.

4. The TgBfrPtr pointer variable is set to the address of the first byte in the before image part of the trigger buffer parameter. This is done by getting the address (using the %Addr function) of the corresponding byte in the array that was declared to contain the trigger buffer. This is done similarly for the TgAftPtr pointer variable. After these two pointers are set, subsequent statements in the trigger program can refer to the subfields of the BfCustomer and AfCustomer data structures that are referencing the appropriate fields in the trigger buffer.

5. In this statement, the program is updating one of the fields of the after image record in the trigger buffer.
```c
#include <decimal.h>
#include <string.h>
#include <trgbuf.h> /*..... Header file for trigger buffer .....*/
/*..... Externally described files .....*/

#pragma mapinc("SalesCus", "ORDAPPLIB/SALESCUS(*ALL)",
    "input key", "d z P", " ", "sc")
#pragma mapinc("Orderhdr", "ORDAPPLIB/ORDERHDR(*ALL)",
    "input key", "d z P", " ", "ord" )

#include "SalesCus"
#include "Orderhdr"

/*..... APIs linkage .....*/
#pragma linkage(QUSRJOBI, OS) 3
#pragma linkage(QMHSNDPM, OS) 4

void QUSRJOBI(char *, int, char *, char *, char *);
void QMHSNDPM(char *, char *, char *, int, char *, char *, int,
    char *, char *);

void sendMessage(char *);

void main(int argc, char **argv)
{

    char JobInfo[86]; /*.... Parameters for QUSRJOBI ....*/
    int JobInfoLen = 86;
    char JobFmt[8] = "JOBI0100";
    char JobName[26];
    char JobId[16];
    char UserId[10], MsgId[7];
    double OrderTotalD, OrderQtyD, PartialTotalD;
    int UpdateType;
```

memset(JobName, ' ', 26);
JobName[0] = '*';
memset(JobId, ' ', 16);

TrgBuffer = (Qdb_Trigger_Buffer_t *) argv[1];

/*.... Setting the pointer to the storage area where the
   system keeps the record image .................*/
Order = (ord_ORDERHDR_i_t *) ((char *) TrgBuffer + 7)

TryBuffer->New_Record_Offset;

/*.... Retrieving the current USERID ....*/
QUSRJOB1(JobInfo, JobInfoLen, JobFmt, JobName, JobId);
memcpy(UserId, JobInfo+18, 10);
if(SalesCus == NULL) /*.... First time ....*/
{
    system("STRCMTCTL LCKLVL(*CHG) CMTSCOPE(*ACTGRP)");
    SalesCus = _Ropen("ORDAPPLIB/SALESCUS", "rr arseq=N");
}
if(Audit == NULL)
    Audit = _Ropen("ORDAPPLIB/AUDITFIL", "ar commit=Y");

strncpy(SalesCustomerKey.SRNBR,
    UserId, 10);
strncpy(SalesCustomerKey.CUSNBR,
    Order->CUSNBR, 5);

SalesCusFB = _Rlocate(SalesCus, (void*) &SalesCustomerKey,
    sizeof(SalesCustomerKey), __KEY_EQ);
if (SalesCusFB->num_bytes == 0)
{
    _Rwrite(Audit, (void*) &SalesCustomerKey,
        sizeof(SalesCustomerKey));
    _Rcommit("Audit Written");
    sendMessage("ORH0001");
    return;
}

void sendMessage(char *MsgId)
{
    char MsgFile[20] = "ORDMSGF	ORDAPPLIB ",
        MsgData[30] = "Trigger Error ",
        MsgType[10] = "*ESCAPE	",
        MsgQueue[10] = "_C_pep ",
        MsgKey[4] = "	";
    struct {
        int provided;
        int available;
        char Excpt[7];
        char filler;
    } ErrorCode;

    /* Other code... */
int   MsgLen          = 0, PgmStack = 1;

ErrorCode.provided = 8;
QMHSNDPM(MsgId, MsgFile, MsgData, MsgLen, MsgType, MsgQueue, 12
       PgmStack, MsgKey, (char *) &ErrorCode);
if(ErrorCode.available)
   {
      printf("Error: %7.7s\n", ErrorCode.Excpt);
      exit(1);
   }
}

Notes: The following notes refer to the previous example.

1 Include the header file trgbuf.h in your ILE C program. You can then use the trigger buffer that is defined in this header file in the trigger program. Therefore, the need to code the trigger buffer in the program is eliminated.

2 This statement allows you to declare the externally described files in your program. Therefore, you do not have to code the record structure in your program.

3 This is an API that can retrieve the user ID for the current job into the program.

4 This is an API that can be used to send a message to another program. In the current example, this is used to send a message to the database writer to make the write operation fail if the current sales person is not authorized to deal with the customer.

5 Declare a pointer to the record image. This pointer is used to access the contents of the trigger buffer.

6 Declare a pointer to the trigger buffer. This pointer can be used to move the offsets to the locations at which the data passed to the program is present.

7 Set the pointer to the record structure to the address at which the data passed to the program is present. In this example, this is the after record image for the ORDERHR table.

8 Call the QUSRJOBI() API to get the user ID for the current job.

9 Start commitment control from within the trigger program by using the system() ILE C instruction.

10 Query the SALESCUS table to determine whether the current user ID is authorized to the customer specified in the record being written to the database.

11 If the user ID is not authorized, record the violation in the AUDIT file and commit the changes to the AUDIT file.

12 If the user ID is not authorized to deal with the specified customer, call the QMHSENDPM() API to send a message to the database manager to make the current operation fail.

11.4.5 Changing the record that fired a trigger

There are situations where it may be useful to let the trigger program update the record that fired the trigger program. This option can be helpful in trigger programs that are designed for data validation and data correction. This section shows how to write trigger programs that change the record that fired the trigger.

To add a trigger record to a physical file, use the ADDPFTRG CL command. If you need to change the record that fired the trigger program, you must specify the Allow repeated change (ALLWREPCHG) parameter of the ADDPFTRG command as *YES.
There are two methods for changing the record that fired that trigger from within the trigger program:

- Change the after image of the trigger record in the trigger buffer. In this case, the trigger has to be a BEFORE trigger.

- Update the trigger record by performing another I/O operation using embedded SQL statements or by using the normal I/O operations of a high-level language. What really happens when you use this approach is that the trigger program is invoked recursively again by the trigger program. This is a good alternative when the first method cannot be used. Note that this method cannot be implemented in languages that do not support recursion.

The Allow repeated change parameter is used only for those programs that actually change the trigger buffer. In this case, the trigger program must be a BEFORE trigger for the update and insert operations. The modified after image is used for the actual insert or update operation in the associated physical file.

It makes no sense to change the trigger buffer on an AFTER INSERT or UPDATE trigger program because it does not actually update the database record.

If you are using a trigger program that calls itself recursively, consider the following important points:

- The trigger must be written in a language that supports recursion.

- The trigger must be an AFTER trigger. In this method, you actually update the trigger record using an IO statement. You cannot update a record before it is written. Therefore, the trigger should be an AFTER trigger.

- When inserting or updating records into tables that have recursive triggers attached, set the isolation level of your program to *NONE. This is because, with any other isolation level, the record that was inserted or updated is locked unless a commit is issued. Therefore, when the trigger program tries to update the record, it finds the record locked, and you receive a file in use error.

- You must create the trigger program with activation group *CALLER. If the activation group of the trigger program is different from that of the program that is inserting the records, the trigger program finds the record locked, and you receive a file in use error.

The scenario for the following examples is described in “Soft coding the trigger buffer in ILE RPG” on page 416. Refer to that section for an ILE RPG code example, which illustrates how to change an after image buffer. In this section, the ILE C programs are shown.

**Changing the trigger buffer example**

```c
/*  *******************************************************************
   . This is a BEFORE UPDATE trigger on the CUSTOMER table. If the  
   . total sales amount for a customer exceeds 90% of the credit.  
   . limit on update then this trigger will invoke the fax program.  
   . If the customer is a special customer which is denoted by a    
   . customer number beginning with a '9' then the credit limit is  
   . automatically increased by 30%. This program changes the      
   . after image of the trigger record before it is written to the .
   . database.                                                  
   *******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
`#include <decimal.h>
#include <recio.h>
#include <trgbuf.h>

/* ...... Include externally described files ...... */
#pragma mapinc( "Customer", "ORDAPPLIB/CUSTOMER(*ALL)",
               "input key", "d_P", " ", "cst" )

#include "Customer"

Qdb_Trigger_Buffer_t *TrgBuffer;
cst_CUSTOMER_i_t  *NewRec;
cst_CUSTOMER_i_t  *OldRec;
decimal( 11,2 )     CheckCrd;
char                CustomerNumber[ 5 ];

void main( int argc, char **argv )
{
    TrgBuffer = (Qdb_Trigger_Buffer_t *) argv[1];

    /* ........ Get the after image of the trigger record ........ */
    NewRec = (cst_CUSTOMER_i_t * ) ((char *) TrgBuffer +
                                    TrgBuffer->New_Record_Offset );

    /* ........ Get the before image of the trigger record ........ */
    OldRec = (cst_CUSTOMER_i_t * ) ((char *) TrgBuffer +
                                     TrgBuffer->Old_Record_Offset );

    /* ........ Get 90% of the credit limit ........ */
    CheckCrd = 0.9 * NewRec->CUSCRD;

    /* ........ Check if the total sales amount exceeds 90% of credit ........ */
    if ( CheckCrd <= NewRec->CUSTOT )
    {
        strncpy( CustomerNumber, NewRec->CUSNBR, 5 );

        /* ........ Check if the customer number begins with a '9' ........ */
        if ( NewRec->CUSNBR[0] == '9' )
        {
            /* ........ Change the trigger buffer and increase the credit ........ */
            NewRec->CUSCRD = NewRec->CUSCRD * 1.3 ;
            printf( "90 percent of credit limit exceeded:\n" );
            printf( "Customer - %s\n", CustomerNumber );
            printf( "The credit limit has been increased:\n" );
            printf( "Old Credit Limit - %D(11,2)\n", OldRec->CUSCRD );
            printf( "New Credit Limit - %D(11,2)\n", NewRec->CUSCRD );
        }
        else
        {
            printf( "Customer - %s\n", CustomerNumber );
            printf( "The credit limit has not been increased:\n" );
            printf( "Old Credit Limit - %D(11,2)\n", OldRec->CUSCRD );
            printf( "New Credit Limit - %D(11,2)\n", NewRec->CUSCRD );
        }
    }
else
    {
        printf( "Customer - %s\n", CustomerNumber );
        printf( "The credit limit is below 90%:\n" );
        printf( "Old Credit Limit - %D(11,2)\n", OldRec->CUSCRD );
        printf( "New Credit Limit - %D(11,2)\n", NewRec->CUSCRD );
    }
}
printf( "90 percent of credit limit exceeded:\n" );
printf( "Customer - %s\n", CustomerNumber );
printf( "Please Wait... Now Sending Fax...\n" );
printf( "Call SENDFAX( NewRec->CUSFAX )\n" );
exit( 0 );
}

Notes: The following notes refer to the previous example.

1. Obtain the address of the after image of the trigger record. This is the image that needs to be changed.

2. Determine 90 percent of the credit limit.

3. Determine whether the total sales amount for the customer has exceeded 90 percent of the credit limit.

4. If 90 percent of the credit limit has been exceeded, see if the customer in question is a special customer by checking whether the customer number begins with a 9.

5. If the customer number begins with a 9, increase the credit limit by 30 percent.

Calling the trigger program recursively

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <decimal.h>
#include <recio.h>
#include <trgbuf.h>

/* ...... Include the externally described files ...... */
#include "Customer"

#pragma mapinc( "Customer", "ORDAPPLIB/CUSTOMER(*ALL)",
        "input key", "d _P", " ", "cst" )

#include "Customer"

Qdb_Trigger_Buffer_t  *TrgBuffer;
cst_CUSTOMER_i_t      *NewRec;
cst_CUSTOMER_i_t      *OldRec;
decimal( 11,2 )        CheckCrd;
static decimal(11,2)   NewCredit;
static decimal(11,2)   OldCrd;
static int NumTimes = 0;
char CustomerNumber[ 5 ];
char dummy[ 5 ];

EXEC SQL BEGIN DECLARE SECTION;
decimal( 11,2 ) NewCrd;
char CustomerNumber[ 5 ];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

void main( int argc, char **argv )
{
    TrgBuffer = (Qdb_Trigger_Buffer_t *) argv[1];

    /* ....... Get the after image of the trigger record ...... */
    NewRec = (cst_CUSTOMER_i_t * ) ((char *) TrgBuffer +
        TrgBuffer->New_Record_Offset );

    /* ....... Get the before image of the trigger record ...... */
    OldRec = (cst_CUSTOMER_i_t * ) ((char *) TrgBuffer +
        TrgBuffer->Old_Record_Offset );

    /* ...... Get 90% of the credit limit ...... */
    CheckCrd = 0.9 * NewRec->CUSCRD;

    /* ...... Check if total sales amount exceeds 90% of the credit ...... */
    /* ...... limit ...... */
    if ( CheckCrd <= NewRec->CUSTOT )
    {
        strncpy( CustomerNumber, NewRec->CUSNBR, 5 );
    /* ...... Check if the customer number begins with a '9' ...... */
    if ( NewRec->CUSNBR[0] == '9' )
    {
        /* ...... Is it the first time the trigger program is called ...... */
        if ( NumTimes == 0 )
        {
            NewCrd = NewRec->CUSCRD * 1.3;
            OldCrd = OldRec->CUSCRD;
            NewCredit = NewCrd;
        /* ...... Update the trigger record ...... */
        EXEC SQL update
            ordapplib/customer
            set
            CUSCRD = :NewCrd
            where
            CUSNBR = :CustomerNumber;
            NumTimes++;  
    }
    else
    {
        printf( "In Non - recursive\n" );
        printf( "90 percent of credit limit exceeded:\n" );
    }
}
printf( "Customer - %s\n", CustomerNumber );
printf( "Please Wait... Now Sending Fax...\n" );
printf( "Call SENDFAX( NewRec->CUSFAX )\n" );
gets( dummy );
}
}

if ( NumTimes != 0 )
{
    printf( "90 percent of credit limit exceeded:\n" );
    printf( "Customer - %s\n", CustomerNumber );
    printf( "The credit limit has been increased:\n" );
    printf( "Old Credit limit - %D(11,2)\n", OldCrd );
    printf( "New Credit Limit - %D(11,2)\n", NewCredit );
    gets( dummy );
}

exit( 0 );

Notes: The following notes refer to the previous example.

1. Get the after image of the trigger record being updated.
2. Determine 90 percent of the credit limit.
3. Determine whether the total sales amount for the customer has exceeded 90 percent of the credit limit.
4. Determine whether this is the first time the trigger program is being executed. This condition is not true if the trigger program is being recursively called a second time. Check it to make sure that the trigger program does not go into an infinite loop. If the trigger program is called recursively, it is terminated without updating the trigger record for the second time. To check the number of times the trigger program was called, we use a static variable in the program. If the value of this variable is not equal to zero, we assume that this is not the first invocation of the trigger program. In the first invocation of the trigger program, we explicitly change the value of the variable from zero to 1.
5. Update the trigger record. This step leads to the trigger being called recursively.
6. If this is the first time the trigger is being called, incrementally increase the variable that keeps a record of the number of times the trigger program has been called.

11.5 Applications and triggers: Design considerations

Be aware of the following list of considerations when you decide to incorporate triggers in your applications and database design:

- Opening database files SHARE(*YES)

  If your trigger is going to call other programs, you may want to take advantage of the SHARE(*YES) option for opening common files. However, if your trigger tries to open the same file that caused the trigger activation with the SHARE(*YES) option, no I/O operations are allowed on that file. If you want to access and modify data in the same file that fired the trigger, you have to use a separate Open Data Path and a full open is required. See Figure 11-29.
You cannot perform DRDA access in a trigger program. In particular, you are not allowed to use the SQL CONNECT statement and the CL CRTSQLPKG statement in a trigger program.

However, if you need to access and modify data located at a remote site from within a trigger, you can open DDM files or start an APPC session with a remote partner program. When accessing remote data in those ways, you may take advantage of the two-phase commit support offered by DB2 Universal Database for iSeries. If the trigger fails after a remote access has been done and an exception is sent back to the originating application, the entire transaction is put into a rollback required state.

In these cases, it is absolutely necessary that an escape message is sent back to the calling interface, either by the system or by the trigger to ensure that all the changes are rolled back consistently.

When triggers are activated remotely by a DRDA data change, they are not allowed to change the current DRDA connection. Consider the scenario in Figure 11-30.

In Figure 11-30, a trigger was fired when the client application issued an update on the database file TABLE A. Any attempt to access a different location by the trigger will fail at the points marked by an asterisk.
If you plan to access data stored on a remote iSeries server from within a trigger program, you may use DDM files or start an APPC conversation. In Figure 11-31, you can see how a trigger can be activated remotely by updating a DDM file. The bottom part of the figure shows how a local trigger can also successfully open a DDM file and perform remote I/O operations. One of these operations can, in turn, fire a remote trigger.
If you need to access multiple locations in the same logical unit of work, the client application has to control the connection switching shown in Figure 11-32.

![Diagram](image)

Figure 11-32  Triggers in a DRDA-2 application

- Destructive data change within triggers:

  The database record that caused the trigger activation is always protected against any attempt to change it. You are not allowed to modify the record that fired the trigger from within the trigger itself. This restriction has been introduced to avoid possible inconsistencies, such as a trigger working with a trigger buffer that no longer matches the real data in the database files.

  **Note:** There are cases where you may want to update the record that fired the trigger. Refer to 11.4.5, “Changing the record that fired a trigger” on page 424, for a discussion on this topic.

The restriction also applies in the case of a chain of triggers. The record that activated the first trigger cannot be modified even by the second one.

If both your triggers and the applications run under commitment control, all of the rows modified by your trigger programs are locked and cannot be changed, not even through the same ODP used to perform the first data change.

For example, in Figure 11-33, the records marked with 1 are protected in any case until the last trigger in the chain has completed its execution. This means that the database I/O operations marked with 2 will result in a failure. In the same example, if both the triggers and the application are running commitment control, the I/O operation marked with 4 will fail. Triggers do not allow you to change the same row more than once when running under the application’s commitment control definition. However, if the trigger is not running under the application’s commitment control definition, multiple changes to the same row are allowed, as indicated by 3.
Figure 11-33  Destructive data change

Table 11-4 summarizes the different levels of protection, depending on the various commitment control scenarios.

Table 11-4  Various commit scenarios

<table>
<thead>
<tr>
<th>Application program</th>
<th>Trigger program</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 COMIT(*YES)</td>
<td>COMIT(*YES)</td>
<td>All rows are protected.</td>
</tr>
<tr>
<td>2 COMIT(*YES)</td>
<td>COMIT(*NO)</td>
<td>Only change operation is protected.</td>
</tr>
<tr>
<td>3 COMIT(*NO)</td>
<td>COMIT(*NO)</td>
<td>Only change operation is protected.</td>
</tr>
<tr>
<td>4 COMIT(*NO)</td>
<td>COMIT(*YES)</td>
<td>Only the change operation is protected. Triggers can change the same record more than once by using the same ODP.</td>
</tr>
</tbody>
</table>

Applications cannot be defined as triggers. Only objects of type *PGM can be added to a physical file. Consequently, service programs cannot be used to define trigger programs.

Triggers can fire other triggers:

A trigger can perform the same type of I/O operation on the same triggering file and fire a copy of itself (recursive triggers). There is a limit of 200 nested triggers (to avoid potentially infinite loops). This limitation does not apply to recursive DELETE triggers, since they end up deleting all of the records in the file and there is no risk of an infinite loop.

A chain of triggers is a rather common scenario in complex scenarios with multiple applications, where you might want to implement the common logic independently from the various applications. Recursive triggers and circular triggers should be coded with much attention to the possibility of generating meaningless loops.

While a file is open, triggers cannot be added, removed, enabled, or disabled.

You cannot add a DELETE trigger program to a dependent file in a referential constraint relationship with a *CASCADE delete rule. Similarly, you cannot add an UPDATE trigger to a dependent file in a referential constraint relationship with the *SETNULL or *SETDFT delete rule.

No trigger is fired if the file is overridden to INHWRT(*YES) (Inhibit Write), even if the program is defined as a *BEFORE trigger.

System changes SEQONLY(*YES) to SEQONLY(*NO):

If the physical file or the dependent logical file is opened for SEQONLY(*YES) and there is a trigger associated with it, the system takes care of changing the open to SEQONLY(*NO) so that the trigger can be invoked for each record changed.
Triggers and object management:

Since the trigger library is resolved when the trigger is added to a physical file, be aware of the following implications:

- Renaming, moving, and recompiling a trigger:
  These operations can be done since there is no "hard" link between the trigger and the database file. If you change the trigger name, delete it, or move it to another library, the data change operation on the associated file will always fail because the system cannot locate the trigger program.

- Saving and restoring:
  When you save a database file, the trigger information is saved in the object description unless you save it with a target release parameter that indicates an OS/400 release prior to V3R1. The trigger program, however, has to be saved separately. You might find it convenient to create the trigger programs in the same library as the associated file so that a SAVLIB command can save all the objects.

- Creating duplicate objects and copying the file:
  When you use the CRTDUPOBJ command or the CPYF command to create a copy of a database file in a different library, the trigger information is not changed. If you need to create a copy of both trigger programs and database files in a different library, consider using the CPYLIB command or CRTDUPOBJ OBJ("ALL) command. When you use CPYLIB, the system updates the trigger library information in the file description if the triggers and the database file are in the same library.

11.6 Recommendations

This section summarizes some of the recommendations presented in the chapter and includes more considerations on trigger development:

- Create the program with ACTGRP("CALLER) if the program is running in an ILE environment. This ensures that the trigger runs under the same commitment definition as the application.

- In an SQL application, use the SET TRANSACTION SQL statement to set the same isolation level of the SQL trigger program as the application. In native, use the appropriate file definition to open the files with or without commitment control at run time, based on the application commitment definition.

- Remember that the trigger inherits the library list of the job that has activated it. Do not forget to explicitly qualify the objects referenced inside the trigger with their libraries or to add those libraries to the library list, since you may get a failure depending on the application that is activating the triggers.

- You may need to have your trigger programs run asynchronously, for example, when you want to trigger a long running process which should not keep the application from proceeding. For this purpose, you might use the Submit Job (SBMJOB) command. In this case, applications cannot expect any kind of feedback from the trigger execution.
Security and triggers:

Triggers run as part of the job that activated them. Since they might access objects to which the current user is not authorized, you may create them with the USRPRF(*OWNER) parameter. Alternatively, since triggers are generally used to enforce business rules, avoid granting the *OBJMGMT authorities or the ALTER and REFERENCE SQL privileges to users that do not strictly need them. Use this precaution to avoid having users easily circumvent the rules by removing the triggers from the database files.

Also remove all the authorities on the trigger program from the public, since they are not necessary for the triggering mechanism. The system can always invoke the trigger, regardless of which user is performing the data change.

Performance considerations:

It is important to take performance into account when you decide to implement triggers in your database design. Remember that triggers are activated by means of an external call. Try to evaluate carefully the trade-off of the performance impact over the benefit of the trigger functions.

Here are some suggestions that you might find useful when developing trigger programs:

- Avoid compiling an ILE trigger with ACTGRP(*NEW).
- Creating a new activation group is expensive; try to prevent it as much as you can.
- If for some special reason your trigger runs in a separate activation group, remember to handle all the exceptions. An unhandled exception will terminate the activation group, close all the files, and cause an implicit rollback for the changes made by your trigger.
- Minimize the number of file opens and closes.
- Try to exit a trigger program the “soft” way. Avoid, if you can, SETON LR in RPG, STOP RUN in COBOL, and exit() in C. This allows you to leave some files open and avoid the overhead of opening them again when you get back into the trigger. This technique is broadly used in our trigger examples. Use a static variable to determine whether the file needs to be opened. In the C language, define the file pointer as static and check for the NULL value. In terms of application logic, if you open a file to append record at the end or for reading with random positioning, you can avoid closing it.
- For SQL triggers, try to write the statements so that the optimizer chooses a reusable ODP.
- Use share open in triggers.

If your triggers call other programs and they access the same files, try to share the Open Data Path by using the share open option. A share open is much faster than a full open, which will create a new ODP.
Triggers, referential integrity, and constraints

DB2 Universal Database for iSeries allows you to define both referential integrity constraints and triggers on the same database table. This chapter explains the coexistence of triggers and referential integrity, with particular regard to the role played by commitment control in this scenario. The chapter starts with a discussion on transaction isolation and recovery before going into the details of the coexistence of triggers and referential integrity.

This chapter discusses:
- Transaction isolation and recovery
- Trigger journal entries
- Trigger and referential integrity
12.1 Transaction isolation and recovery

All triggers, when they are activated, perform a SET TRANSACTION statement so that all of the operations by the trigger are performed with the same isolation level as the application program that caused the trigger to be run. The user may put her own SET TRANSACTION statements in an SQL-control-statement in the SQL-trigger-body of the trigger. If the user places a SET TRANSACTION statement within the SQL-trigger-body of the trigger, then the trigger will run with the isolation level specified in the SET TRANSACTION statement, instead of the isolation level of the application program that caused the trigger to be run.

If the application program that caused a trigger to be activated is running with an isolation level other than No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger will run under commitment control and not be committed or rolled back until the application commits its current unit of work.

If ATOMIC is specified in the SQL-trigger-body of the trigger, and the application program that caused the ATOMIC trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger will not be run under commitment control.

If the application that caused the trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), then the operations of a trigger are written to the database immediately and cannot be rolled back.

If both system triggers defined by the Add Physical File Trigger (ADDPFTRG) CL command and SQL triggers defined by the CREATE TRIGGER statement are defined for a table, we recommend that the system (external) triggers perform a SET TRANSACTION statement so that they are run with the same isolation level as the original application that caused the triggers to be activated. We also recommend that the system (external) triggers run in the Activation Group of the calling application. If the system triggers run in a separate Activation Group (ACTGRP(*NEW)), then those system triggers will not participate in the unit of the work for the calling application, nor in the unit of work for any SQL triggers. System triggers that run in a separate Activation Group are responsible for committing or rolling back any database operations they perform under commitment control. Note that SQL triggers defined by the CREATE TRIGGER statement always run in the caller’s Activation Group.

If the triggering application is running with commitment control, the operations of an SQL trigger and any cascaded SQL triggers will be captured into a sub-unit of work. If the operations of the trigger and any cascaded triggers are successful, the operations captured in the sub-unit of work will be committed or rolled back when the triggering application commits or rolls back its current unit of work. Any system triggers that run in the same Activation Group as the caller, and perform a SET TRANSACTION to the isolation level of the caller, will also participate in the sub-unit of work. If the triggering application is running without commit control, then the operations of the SQL triggers will also run without commitment control.

If an application that causes a trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), and it issues an INSERT, UPDATE, or DELETE statement that encounters an error during the execution of the statement, no other system (external) and SQL triggers will still be activated following the error for that operation. However, a number of changes will already be performed. If the triggering application is running with commitment control, the operations of any triggers that are captured in a sub-unit of work will be rolled back when the first error is encountered, and no additional triggers will be activated for the current INSERT, UPDATE, or DELETE statement.
12.2 Trigger journal entries

Applying and removing journal changes on physical files does not cause triggers to be activated. When a record is changed by a trigger, its corresponding journal entry has a special marker that identifies that particular change as a result of a trigger action. If your triggers are using journals other than the application journals or if they perform some non-database activity (data queues, messages, data areas, and so on), you cannot find any evidence of the trigger activity in the application journal, and applying the journal entries on a restored version of a database file may not lead to the same result as the application execution would produce.

In these cases, when you develop a trigger program, we recommend that you code the appropriate actions to make a possible recovery process easier. For example, you can send user-defined journal entries to the application journals using the QJOSJRNE API or the SNDJRNE CL command. These entries might not be applied or removed, but they at least provide a trail of what happened on the system due to triggers activity. To fully recover the trigger actions, develop a specific procedure to process these user-defined journal entries. For more information about journaling, applying, and removing journal entries, and for a full description of QJOSJRNE, see Backup and Recovery, SC41-5304.

12.3 Triggers and referential integrity

This section discusses:

- Triggers and referential constraints
- Commitment control considerations
- Journal changes with triggers and referential integrity
- Combining triggers and stored procedures

Combining the various DB2 Universal Database for iSeries functions gives you new advantages in writing your applications in terms of flexibility, performance, and ease of development and maintenance. This chapter presents several scenarios where different functions coexist. It also points out interesting technical considerations to be aware of when you start implementing your applications.

12.4 Comparing referential integrity and triggers

DB2 Universal Database for iSeries allows you to define both referential constraints and triggers on the same database file. This chapter explains the coexistence of triggers and referential integrity with particular regard to the role played by commitment control in this scenario. It also discusses how the journal entries reflect the effects of triggers and referential integrity enforcement and the implications on the recovery process.

12.4.1 Using triggers to implement referential integrity rules

Many different relational database platforms have implemented referential integrity by means of system-provided triggers. Generally, this design choice carries a strong limitation. Referential constraints can be enforced only when data is changed in the database. After a restore process, for example, the logical consistency of the data in the database may not be guaranteed. DB2 Universal Database for iSeries implements declarative referential integrity. This means that data consistency in a referential integrity network is verified also after a restore operation or an apply of the journal entries or when a new referential constraint is
created in an existing database. Using triggers to enforce referential constraint ensures data validation only at a single I/O operation, rather than at the global contents of the database. This basic difference should be kept in mind when business rules are enforced by means of trigger programs.

A practical situation where you may want to use triggers to enforce a referential integrity type of rule is represented by the UPDATE CASCADE rule, which is not yet provided by DB2 Universal Database for iSeries in the referential constraints definition. For example, in a hotel management application, you may have the database layout shown in Figure 12-1.

![Figure 12-1 An UPDATE CASCADE example](image)

In Figure 12-1, the Customers file contains all the customers currently present at the hotel and reports their room number. The Item Detail file contains all the items charged to the various rooms. If a customer moves from one room to another, we want all of the items to be charged to the new room numbers. We need to propagate the update operation on the Customer file down to the Item Detail file, but there is no referential integrity rule that operates this way in DB2 Universal Database for iSeries. This constraint can be implemented by a trigger program that is activated by update operations on the Customer file. The trigger can check if the room number has been changed and update all the corresponding records in the detail file.

In general, we recommend that you use declarative referential integrity as much as you can, as opposed to implementing the same rules by means of a trigger program. Use referential constraints to ensure that your data validity is constantly enforced, even after a restore operation, and to provide better performance in verifying data relationships since this type of checking is done by the system at a low level.

### 12.5 Constraints and triggers: Ordering the actions

When constraints and triggers coexist in the same file, it is important that you remember how DB2 Universal Database for iSeries orders the various actions. The following discussion should help you determine whether to use a RESTRICT or a NOACTION rule and whether to implement a BEFORE or an AFTER trigger to satisfy your application requirements.

Since DB2 Universal Database for iSeries allows the coexistence of multiple constraints and triggers in the same file, we have to analyze several possible combinations.

#### 12.5.1 Insert operations

Insert operations can lead to unique key violations on physical files and to referential integrity constraint violations on dependent files. As far as referential integrity constraints are concerned, insert operations will always be successful on parent files.
The following sequence of actions occurs after an insert request in a DB2 Universal Database for iSeries database file:

1. The *BEFORE trigger is activated.
2. If commitment control is not started, a check constraint is processed at this point.
3. The record is inserted and any non-constraint checking is performed (for example, member full).
4. The *AFTER trigger is activated.
5. If the file is a dependent file, any referential constraint is enforced at this time.
6. If commitment control is started, a check constraint is processed immediately after referential constraint enforcement.

This method of sequencing the actions allows you to write an INSERT trigger that, for example, removes or updates a previously existing key value to avoid a duplicate key exception.

12.5.2 Update operations

Updates are more complex to analyze, since they influence data integrity either when they affect a parent or a dependent file of a referential integrity network. This is the detailed sequence:

1. The *BEFORE trigger is activated.
2. If the file is a parent of a *RESTRICT referential constraint, this constraint is enforced.
3. If commitment control is not started, a check constraint is processed at this point.
4. The record is actually updated, and any non-constraint checking is performed (for example, invalid data in the new record image, such as an invalid date format).
5. The *AFTER trigger is activated.
6. If the file is a parent file of a *NOACTION referential constraint, this constraint is enforced.
7. If the file is a dependent file, the constraint is enforced.
8. If commitment control is started, a check constraint is processed immediately after referential constraint enforcement.

12.5.3 Delete operations

Data integrity can be affected by a delete operation only if it is executed against a parent file. Delete operations on dependent files are always successful. In addition, delete operations cannot possibly lead to a violation of unique constraints. This is the sequence of actions DB2 Universal Database for iSeries takes when a delete operation is being performed:

1. The *BEFORE trigger is activated.
2. If the file is a parent file of a *RESTRICT delete rule, this constraint is enforced at this time.
3. If commitment control is not started, a check constraint is processed at this point.
4. The record is actually deleted from the database file, and any non-constraint checking is performed at this time.
5. The *AFTER trigger is activated.
6. If the file is a parent file of a *CASCADE delete rule, this constraint is enforced at this time.
   If multiple *CASCADE delete constraints have been defined, all of them are enforced now. Even if some of the files affected by the cascade process are parent files with a *RESTRICT delete rule, all the matching records are deleted first. Then, the *RESTRICT
delete rule is enforced. In Figure 12-2, you can see how this sequence influences the result of a delete operation in practice.

![Rule ordering in a cascade network](image)

Figure 12-2  Rule ordering in a cascade network

The scenario shown in Figure 12-2 implements the following business rule: If a customer is deleted, all the related orders and invoices must also be deleted. Alternatively, deleting an invoice related to an existing order must be prevented. If an application deleted a customer record, DB2 Universal Database for iSeries deletes all the corresponding invoices and orders. When this process has completed, the *RESTRICT rule in the invoice file is enforced, but by that time, all the matching orders have already been removed and the operation terminates successfully. If the *RESTRICT rule is enforced before the *CASCADE rule on the orders file, the whole operation fails.

7. If the file is a parent file of a *SETNULL or *SETDFT delete rule, these constraints are enforced at this time.

8. If the file is a parent file of a *NOACTION delete rule, this constraint is enforced at this time.

9. If commitment control is started, a check constraint is processed immediately after referential constraint enforcement.

Remember:
- No delete triggers can be defined on a file that depends on a referential constraint with the *CASCADE delete rule.
- No update triggers can be defined on a file that depends on a referential constraint with the *SETNULL or *SETDFT delete rule.

It is important that you keep in mind the following consideration when you decide to combine triggers and referential integrity constraints in your database. As you may have noticed, DB2 Universal Database for iSeries activates the *AFTER triggers before enforcing any referential integrity constraint with update or delete rules different from *RESTRICT. This implies that, when you develop *AFTER triggers for your database, you may not always assume that the I/O operation that activated them has already completed successfully when the trigger programs are in execution. The reason is because the operation may fail later as a result of some referential constraint violation. An example of this situation is shown in Figure 12-3.
The example shown in Figure 12-3 is a slight variation of our application scenario. If we try to translate the constraints in terms of business rules, we can say that this implementation allows an application to delete a customer if there are no outstanding orders or if all of the orders are “empty” (that is, with no detail rows). Empty orders may be the result of failures or an incomplete order cancellation due to application problems. If an application deletes a record from the customer file, the sequence of actions is:

1. The record is deleted from the Customer file.
2. The AFTER DELETE trigger is activated.
3. DB2 Universal Database for iSeries will start removing the related records in the Order Header file.
4. The *RESTRICT constraint is enforced at this time. If there are any related records in the Order Detail file, the entire transaction is rolled back, and the original delete on the Customer file will fail.

You may notice that the AFTER DELETE trigger should not assume that the original delete terminated successfully. This means that you should design this trigger carefully. Avoid, for example, any non-database activity, since this kind of operation cannot be rolled back if the subsequent delete cascade fails.
12.6 Triggers, referential integrity, and commitment control

We already discussed the implications of triggers and referential integrity on journaling and commitment control in 11.3.1, “Commitment control and triggers” on page 378. This section describes how the integrity of transactions can be preserved when triggers and referential integrity constraints are both operating on the database.

Triggers, referential integrity, and commitment control interact in different ways, depending on whether the application changing the data in the database is running commitment control. We address these two cases separately.

12.6.1 When the application is not running commitment control

Even if the application runs without commitment control, the system starts a transparent commitment control cycle whenever rules, other than *RESTRICT, are defined for the referential constraints of the file being accessed. Consequently, if a trigger program is activated in this environment, the commitment control parameter passed to the program by DB2 Universal Database for iSeries in the trigger buffer (see 11.2, “Trigger program structure” on page 364) is set to a value different from 0, indicating that commitment control has been started.

The trigger has no direct way to determine whether commitment control has been started by the system or by the application. If the trigger opens a file with commitment control, according to the parameter received into the trigger buffer, either of the following two situations occurs:

- The file opened by the trigger program has no referential integrity constraints, or it only has *RESTRICT rules.
  In this case, the open fails because no user commitment definition has actually been started. The system issues an exception (CPF4326). The trigger should monitor the exception and re-open the file without commitment control. All the changes made by the trigger are uncommitted (that is, they become immediately permanent).
- The file opened by the trigger program has referential integrity constraints with rules other than *RESTRICT.
  The open will succeed, and the changes fall into the system commitment definition. The changes made by the trigger are automatically rolled back if a failure occurs before the originating I/O has completed.

If the trigger ignores the commitment control parameter and opens the file without commitment control, the open will always succeed.

If the trigger uses commitment control, it runs in its own commitment definition. This implies that a failure due to the enforcement of a referential integrity constraint does not roll back any changes previously made by the trigger program.

12.6.2 When the application runs under commitment control

In this scenario, the trigger program should use the SET TRANSACTION statement (SQL triggers) or open any database file with the commitment control option (see 11.2, “Trigger program structure” on page 364 for a full discussion on this topic). This way, you are guaranteed the atomicity of the whole transaction. All the changes that result from the original database I/O are treated as a single transaction. Triggers, of course, must share the same commitment definition as the application. Otherwise, their database changes are not considered a part of the atomic transaction.
Consider the example in Figure 12-3 on page 443. Suppose the AFTER DELETE trigger defined on the customer file is performing some database changes. In Figure 12-4, you can see the flow of database operations caused by delete operation 1.

The application in Figure 12-4 runs under commitment control. The trigger program shares the same commitment definition. This can be ensured by compiling the trigger with ACTGRP(*CALLER) if it is an ILE program. The delete operation will fail after the trigger has terminated (2) and all the changes included in box 1 are rolled back by the system. The application still has the ability to commit or roll back any other previous change operation (3) that is not affected by the implicit rollback.

Summarizing the previous discussion, we strongly recommend that you use commitment control in your applications if the database design includes the coexistence of referential integrity constraints with rules other than *RESTRICT and triggers at the same time. If this condition cannot be guaranteed by your application environment, code triggers carefully to determine whether they are running under the user commitment definition or under the system commitment definition. Never start commitment control in triggers if the triggers are created with ACTGRP(*CALLER). It is better to monitor for the appropriate error message when opening files under commitment control.

### 12.7 Referential integrity, triggers, and journal entries

In 12.2, “Trigger journal entries” on page 439, we explain how DB2 Universal Database for iSeries logs additional information to identify that a journal entry has been generated by a database change resulting from a trigger action.

This additional information is relevant to those applications implementing some sort of online system duplication such as MIMIX, Multiple Systems Software, and Dual System Backup. Generally, these applications scan the journal receivers and send the record images across the network to a different system. At the remote site, a partner program receives the record images and performs the action required, such as inserting, updating, or deleting the records from the duplicated database.
These kinds of applications or products can take significant advantage of triggers and referential integrity from a performance standpoint. The backup and the production database are exact copies of each other in terms of definitions, referential constraints, and triggers. Therefore, the journal entries caused by triggers and referential constraints do not need to be sent across the network. The duplicated application needs to send only the “real” changes, and the partner program only needs to perform the same operations as the original application.

The amount of information flowing across the network can be greatly reduced by using this technique. Consider the example where deleting a record from the header causes a cascade delete on two dependent files. To reproduce the same situation on the hot backup database, we need to send across the line, only the first journal entry related to the delete operation of the record in the header file. In a traditional environment, where referential integrity is implemented at the application level, all the journal entries related to the dependent files are also sent.
User-defined functions (UDFs) are used to enrich the capabilities of the database manager by providing new functions to the already rich set of predefined functions. They can be written in two ways on DB2 Universal Database for iSeries. One approach is described as SQL UDFs. This is based on procedural extensions to the SQL language, highly used by other DBMS providers, as described in Chapter 14, “SQL user-defined functions” on page 467. The other approach is based on high-level languages that you are familiar with, such as C, CL, RPG, COBOL, Java and others. They are described as External UDFs and are explained in Chapter 15, “External user-defined functions” on page 493.

UDFs may or may not receive some values as parameters and return a result. Depending on the result they provide, they are known as scalar UDF or Table UDF; also known as user-defined table functions (UDTF). Scalar UDF are those UDFs returning a single value while UDTFs return a temporary table.

This part is dedicated to the UDFs and UDTFs. It has the following chapters:

- Chapter 13, “User-defined functions” on page 449, where the general concepts of UDFs and UDTFs are exposed
- Chapter 14, “SQL user-defined functions” on page 467, where particularities of UDFs and UDTFs written in SQL PSL are explained
- Chapter 15, “External user-defined functions” on page 493, where high-level language written UDFs and UDTFs, known as external UDFs, are exposed
User-defined functions

This chapter discusses the advantages of developing user-defined functions (UDFs) on DB2 Universal Database for iSeries as the facility to create scalar or table functions just like another system-supplied function that can be used in SQL statements.

This chapter describes:
- Concepts and benefits of UDFs
- Types of UDFs
- The CREATE FUNCTION
- Resolving UDFs
- System catalog tables
- Debugging UDFs
13.1 Introduction

UDFs are host-language functions for performing customized, often used tasks in applications. UDFs allows the programmers to modularize a database application, creating a function that can be used in SQL.

DB2 Universal Database for iSeries comes with a rich set of built-in functions, but users and programmers may have different particular requirements not covered by them. UDFs comes to play a very important role by allowing users and programmers to enrich the database manager by providing their own functions.

Some of the advantages of UDFs are:

- **Customization**
  Functions specifically required by your application not existing in the set of DB2 built-in functions can be created. Whether the function is a simple transformation, a trivial calculation, or a complex multivariate analysis, you may choose a UDF to do the job.

- **Flexibility**
  You can use functions with the same name in the same library that accepts different sets of parameters.

- **Standardization**
  Many of the programs that you implement use the same basic set of functions, but there are minor differences in all the implementations. If you correctly implement your business logic as an UDF, you can reuse those UDFs in your other applications using SQL.

- **Object-relational support**
  UDF also provides additional functions for User-defined Distinct Type (UDT) created in the database. UDFs act as methods for UDTs. For more information about UDTs and how UDFs are used to encapsulate methods for them, see *DB2 UDB for AS/400 Object Relational Support*, SG24-5409.

- **Performance**
  A UDF can run in the Database engine and is very useful for performing calculations in the database manager server. Another area where performance may be increased is in dealing with Large Objects (LOBs). UDFs may be used for extracting or modifying portions of the information contained in a LOB directly in the database manager server instead of sending the complete LOB to the client side.

- **Migration**
  When migrating from other database managers, there can be built-in functions that are not defined in DB2 Universal Database for iSeries. UDFs allow us to create those functions in order to make the migration process easier.

UDFs are useful for the following reasons:

- **Supplement built-in functions**
  A UDF is a mechanism with which you can write your own extensions to SQL. The built-in functions supplied with DB2 are a useful set of functions, but they may not satisfy all of your requirements. Therefore, you may need to extend SQL. For example, porting applications from other database platforms may require coding of some platform-specific functions.

- **Handle user-defined data types**
  You can implement the behavior of a user-defined distinct type (UDT) using UDFs. When you create a distinct type, the database provides only cast functions and comparison
operators for the new type. You are responsible for providing any additional behavior. It is best to keep the behavior of a distinct type in the database where all of the users of the distinct type can easily access it. Therefore, UDFs are the best implementation mechanism for UDTs.

- Provide function overloading
  Function overloading means that you can have two or more functions with the same name in the same library. For example, you can have several instances of the SUBSTR function that accept different data types as input parameters. Function overloading is one of the key features required by the object-oriented paradigm.

- Allow code re-use and sharing
  A business logic implemented as a UDF becomes part of the database, and it can be accessed by any interface or application using SQL.

### 13.2 Nature of user-defined functions

A function is a relationship between a set of input values and a set of result values. When invoked, a function performs some operation (for example, concatenate) based on the input and returns a single or multiple results to the invoker. Depending on the nature of the return value or values, UDFs can be classified into:

- User-defined scalar functions
- User-defined table functions

#### 13.2.1 User-defined scalar functions

User-defined scalar functions are UDFs that return a single scalar value. A function that returns the temperature in Celsius for a given temperature in Fahrenheit is a scalar function. The statement in Example 13-1 uses two different scalar functions.

**Example 13-1  Scalar UDTs and UDTFs combined in the same Select statement**

```sql
SELECT
DEC2DATE(ORDERDATE),
DEC2DATE(SHIPDATE),
WORKINGTIME(ORDERDATE, SHIPDATE),
FROM ORDERS
```

The scalar functions in the example are DEC2DATE and WORKINGTIME.

DEC2DATE will be executed two times for each row processed by the SELECT statement.

#### 13.2.2 User-defined table functions

User-defined table functions (UDTFs) are UDFs that are capable of returning a set of output values. This set of output values is known as a table or result set. UDTFs return a table instead of a scalar value. Examples of this type of functions are:

- A function that returns the names of sales representatives in a specified region.
- A function that returns all employees whose annual compensation is above the average of the organizational unit to which they belong.
- A function returning the $k$ most profitable customers is a table UDF.
13.3 Type of user-defined functions

There are three categories into which UDFs can be divided:

- Sourced UDFs
- SQL UDFs
- External UDFs

13.3.1 Sourced UDFs

Sourced UDFs are functions registered to the database that simply make a reference to another function. In fact, they map to the sourced function, which means that there is no coding involved. As such, nothing more is required in implementing a sourced UDF than registering it to the database using the CREATE FUNCTION statement. Sourced UDFs are often used to implement the required behavior of UDTs.

You can define a Sourced UDF over an arithmetical operator like +, -, *, /, ||. This is useful if you want to enable the use of binary operators such as arithmetic operations for UDTs. For example, if you want to add two columns defined as UDT MONEY, the function “+” may be defined as a function “+(MONEY, MONEY) returning MONEY, based on the standard “+”(DECIMAL, DECIMAL) returning DECIMAL. See Example 13-2.

Example 13-2  Sourced function + for MONEY UDT

```
CREATE FUNCTION Library/"+"(MONEY, MONEY)
returns MONEY
specific plus00001
source QSYS2/"+"(decimal, decimal);
```

13.3.2 SQL UDFs

SQL UDFs are functions written entirely using procedural SQL language. Their “code” is actually SQL statements embedded within the CREATE FUNCTION statement. SQL UDFs provide several advantages:

- They are written in SQL, making them portable to other database platforms.
- Defining the interface between the database and the function is by use of SQL declares, with no need to worry about details of actual parameter passing.
- They allow the passing of large objects, datalinks, and UDTs as parameters, and subsequent manipulation of them in the function itself.

As an example, consider the situation where there are many tables with columns DECIMAL(8) representing dates in format YYYYMMDD, and those dates have to be converted to DATE. There is not a built-in function that performs the requested operation, but we can use an SQL statement such as the one in Example 13-3.

Note: One useful and important use of a table function is the ability to access data in non-relational objects with an SQL. A table function can be written to extract data out of a stream file in IFS, and then the invoking SQL statement is able to process that data like data from an SQL-created table.
Example 13-3  Decimal YYYYMMDD date conversion to DATE format in a SELECT statement

```
SELECT 
  DATE(
    SUBSTRING(DIGITS(ORDER_DATE), 1, 4) || '-'
    SUBSTRING(DIGITS(ORDER_DATE), 5, 2) || '-'
    SUBSTRING(DIGITS(ORDER_DATE), 6, 8)
  ) AS CONVERTED_DATE
FROM 
SOURCE_TABLE
```

Alternatively, we can define a UDF that makes the desired conversion and changes the previous select statement to the one in Example 13-4.

Example 13-4  Select statement using a UDF for DECIMAL to DATE conversion

```
SELECT DEC2DATE(ORDER_DATE) AS CONVERTED_DATE FROM SOURCE_TABLE
```

The SQL UDF for this conversion may be as shown in Example 13-5.

Example 13-5  SQL DEC2DATE UDF

```
CREATE FUNCTION DEC2DATE (
  DATEDEC DECIMAL(8, 0) )
RETURNS DATE
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
RETURNS NULL ON NULL INPUT
NO EXTERNAL ACTION
BEGIN
  DECLARE RESULT DATE ;
  DECLARE InvalidDate CONDITION FOR '22007';
  DECLARE EXIT HANDLER FOR InvalidDate
  BEGIN
    RETURN CAST(NULL AS DATE);
    SIGNAL SQLSTATE '01HDI' SET MESSAGE_TEXT='Invalid date';
  END;
  SET RESULT = DATE(
    SUBSTRING(DIGITS(DATEDEC),1,4) || '-' ||
    SUBSTRING(DIGITS(DATEDEC),5,2) || '-' ||
    SUBSTRING(DIGITS(DATEDEC),7,2));
  RETURN RESULT;
END ;
```

SQL UDFs are also useful when you want to see the result of a query as a table, for example, a table that has a group of employees in a particular project. The implementation of these SQL UDFs are discussed in Chapter 14, “SQL user-defined functions” on page 467.

13.3.3  External UDFs

External UDFs are references to programs and service programs written in high-level languages such as C, C++, ILE CL, COBOL, ILE COBOL, FORTRAN, PLI, RPG, ILE RPG or Java. After the function is registered to the database, the database invokes the program or service program whenever the function is referenced in a DML statement. As in SQL UDFs, external UDFs can return a scalar value or a table.
Some of the reasons to work with external UDFs are:

- To perform non-database functions
- To access non-relational data
- To reuse existing code
- To leverage existing skills

An example can be to write an external function that checks whether the a passed Binary Large Object (BLOB) contains a picture in GIF format. See Example 13-6.

Example 13-6 Creation of an external function

```sql
CREATE FUNCTION Library/ISGIF(BLOB)
returns INTEGER
language C
specific ISGIF0001
no sql
no external action
external name 'Library/PICTCHECK(fun_CheckPictureType)'
parameter style SQL;
```

13.4 Creating user-defined functions

Before a UDF can be recognized and used by the database manager, it should be created using the CREATE FUNCTION statement. This statement allows you to specify the name and the language of the function, as well as certain behavioral characteristics such as though the function is or is not deterministic, if it may be used in parallel or not, if it reads or modifies SQL data, and so on.

You can use the DROP FUNCTION statement to delete the function in the catalog information entry. You can find detailed information about the CREATE FUNCTION in the SQL Reference, which is available on the Web at:


13.4.1 CREATE FUNCTION

The CREATE FUNCTION statement can be used to define any of the three kinds of UDFs. It can be embedded in an application program, or issued interactively. It is an executable statement that can be dynamically prepared.

During UDF creation, you define characteristics that affect the way in which it is identified in DB2 Universal Database for iSeries. This section explains some of these ways. For a complete description of the CREATE FUNCTION command, refer to SQL Reference for Cross-Platform Development, which you can find on the Web at:


Function name

The function name names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a UDF that exists at the current server.

For SQL naming, the function is created in the schema specified by the implicit or explicit qualifier. For system naming, the function is created in the schema that is specified by the
qualifier. If no qualifier is specified, the function is created in the current library (*CURLIB). If there is no current library, the function is created in QGPL.

**Parameter-Declaration**

Parameter-Declaration specifies the number of input parameters of the function and the data type of each parameter. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in CREATE FUNCTION is 90. For external functions created with PARAMETER STYLE SQL, the input and result parameters specified and the implicit parameters for indicators, SQLSTATE, function name, specific name, and message text, as well as any optional parameters, are included. The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program that is used to compile the external program.

<table>
<thead>
<tr>
<th>Tip: For portability of functions across other DB2 Universal Database, do not use the following data types, which might have different representations on different platforms:</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ FLOAT: Use DOUBLE or REAL instead.</td>
</tr>
<tr>
<td>➤ NUMERIC: Use DECIMAL instead.</td>
</tr>
</tbody>
</table>

**RETURNS**

RETURNS specifies the output of the function.

**SPECIFIC specific-name**

SPECIFIC specific-name specifies a unique name for the function.

When defining multiple functions with the same name and schema but different parameters (see 13.5, “Resolving a UDF” on page 460), we recommend that a specific name also be specified. The specific name can be used to uniquely identify the function, such as when sourcing on this function, dropping the function, or commenting on the function. However, the function cannot be invoked by its specific name.

The specific name is implicitly or explicitly qualified with a schema name. If a schema name is not specified on CREATE FUNCTION, it is the same as the explicit or implicit schema name of the function name (function-name). If a schema name is specified, it must be the same as the explicit or implicit schema name of the function name. The name, including the schema name, must not identify the specific name of another function or procedure that exists at the current server.

If the SPECIFIC clause is not specified, a specific name is generated.

| Tip: One important use of the SPECIFIC keyword is that it allows you to control the name of the underlying C program object when an SQL Function name is longer than 10 characters. |

**LANGUAGE**

LANGUAGE specifies the language interface convention to which the function body is written. All programs must be designed to run in the server’s environment.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program at the time the function is created. The language of the program is assumed to be C if:
The program attribute information associated with the program does not identify a recognizable language.

The program cannot be found.

**DETERMINISTIC or NOT DETERMINISTIC**
The DETERMINISTIC or NOT DETERMINISTIC clause specifies whether the function is deterministic.

- NOT DETERMINISTIC: Specifies that the function will not always return the same result from successive function invocations with identical input arguments. NOT DETERMINISTIC should be specified if the function contains a reference to a special register or a non-deterministic function.
- DETERMINISTIC: Specifies that the function will always return the same result from successive invocations with identical input arguments.

A UDF that returns the temperature in Celsius provided that a Fahrenheit temperature is deterministic, because no matter under which circumstances it is called, it always will return the same result when parameter values are equal.

A UDF that accesses a thermometer and returns the temperature is non-deterministic because it may provide different results even if the received parameters are equal.

**CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA or NO SQL**
This clause specifies whether the function can execute any SQL statements and, if so, what type. The database manager verifies that the SQL issued by the function is consistent with this specification.

- CONTAINS SQL: The function does not execute SQL statements that read or modify data.
- NO SQL: The function does not execute SQL statements.
- READS SQL DATA: The function does not execute SQL statements that modify data.
- MODIFIES SQL DATA: The function can execute any SQL statement except those statements that are not supported in any function.

**FENCED or NOT FENCED**
The FENCED or NOT FENCED clause specifies whether the function will run in the same thread as the invoking SQL statement or in a separate thread.

- FENCED: The function will run in a separate thread.
- NOT FENCED: The function may run in the same thread as the invoking SQL statement. NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function.

A UDF, defined as FENCED, runs in the same job as the SQL statement that invoked it. However, the UDF runs in a system thread, separate from the thread that is running the SQL statement. By default, UDFs are created as FENCED. For complex UDFs, this separation is meaningful, as it will avoid potential problems such as generating unique SQL cursor names. A UDF created with the NOT FENCED option indicates to the database that the user is requesting that the UDF can run within the same thread that initiated the UDF. Unfenced is a suggestion to the database, which can still decide to run the UDF in the same manner as a fenced UDF.
RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
This clause specifies whether the function is called if any of the input arguments are null at execution time.

- RETURNS NULL ON INPUT: Specifies that the function is not invoked if any of the input arguments is null. The result is the null value.
- CALLED ON NULL INPUT: Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or non-null value.

EXTERNAL ACTION or NO EXTERNAL ACTION
This clause specifies whether the function contains an external action.

- EXTERNAL ACTION: The function performs some external action (outside the scope of the function program). Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action. An example of an external action can be to insert a row to a table or putting an entry on a data queue.

- NO EXTERNAL ACTION: The function does not perform an external action. It need not be called with each successive function invocation.

SCRATCHPAD
SCRATCHPAD specifies whether the function requires a static memory area.

SCRATCHPAD integer
SCRATCHPAD integer specifies that the function requires a persistent memory area of length integer. The integer can range from 1 to 16,000,000. If the memory area is not specified, the size of the area is 100 bytes. If parameter style DB2SQL is specified, a pointer is passed following the required parameters that points to a static storage area. If PARALLEL is specified, a memory area is allocated for each user-defined function reference in the statement. If DISALLOW PARALLEL is specified, only one memory area will be allocated for the function.

The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

```
SELECT A, UDFX(A)
FROM TABLEB
WHERE UDFX(A) > 103 OR UDFX(A) < 19
```

If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the SQL statement. Specify the DISALLOW PARALLEL clause for functions that will not work correctly with parallelism.

SCRATCHPAD is only allowed with PARAMETER STYLE DB2SQL or PARAMETER STYLE DB2GENERAL.

Tip: The use of UNFENCED versus FENCED UDFs provides better performance since the original query and the UDF can run within the same thread.
NO SCRATCHPAD

NO SCRATCHPAD specifies that the function does not require a persistent memory area.

FINAL CALL

FINAL CALL specifies whether the function requires special call indication. If PARAMETER STYLE DB2SQL is specified and FINAL CALL is specified, an additional parameter is passed to the function indicating first call, normal call, or final call.

- NO FINAL CALL: Specifies that a final call is not made to the function.
- FINAL CALL: Specifies that a final call is made to the function. To differentiate between final calls and other calls, the function receives an additional argument that specifies the type of call. FINAL CALL is only allowed with PARAMETER STYLE DB2SQL or PARAMETER STYLE DB2GENERAL. The types of calls are:
  - First call: Specifies the first call to the function for this reference to the function in this SQL statement. A first call is a normal call. SQL arguments are passed and the function is expected to return a result.
  - Normal call: Specifies that SQL arguments are passed and the function is expected to return a result.
  - Final call: Specifies the last call to the function to enable the function to free resources. A final call is not a normal call. If an error occurs, the database manager attempts to make the final call. A final call occurs at these times:
    - End of statement: When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
    - End of a parallel task: When the function is executed by parallel tasks.
    - End of transaction: When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

Some functions that use a final call can receive incorrect results if parallel tasks execute the function. For example, if a function sends a note for each final call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that have inappropriate actions when executed in parallel.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made when the cursor is closed or the application ends. If a commit occurs at the end of a parallel task, a final call is made regardless of whether a cursor defined as WITH HOLD is open.

Commitable operations should not be performed during a FINAL CALL, because the FINAL CALL may occur during a close invoked as part of a COMMIT operation.

PARALLEL

The PARALLEL parameter indicates if the function can be run in a parallel implementation of the query (if the optimizer chooses to do so); thus, it only applies when DB2 Symmetric Multiprocessing (SMP) is installed and activated. This means that the same UDF program can be running in multiple threads at the same time. Therefore, if ALLOW PARALLEL is specified for the UDF, ensure that it is thread safe.
The default is DISALLOW PARALLEL, if you specify one or more of the following clauses:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- FINAL CALL
- MODIFIES SQL DATA
- SCRATCHPAD

Otherwise, ALLOW PARALLEL is the default.

User-defined table functions cannot run in parallel; therefore, DISALLOW PARALLEL must be specified when creating the function.

**DBINFO**

DBINFO specifies whether or not the function requires the database information be passed.

- **DBINFO**
  - This specifies that the database manager should pass a structure containing status information to the function. Detailed information about the DBINFO structure can be found in include file SQLUDF in QSYSINC.H.
  - DBINFO is only allowed with PARAMETER STYLE DB2SQL or PARAMETER STYLE DB2GENERAL.

- **NO DBINFO**
  - This specifies that the function does not require the database information to be passed.

**PARAMETER STYLE**

PARAMETER STYLE specifies the conventions used for passing parameters to and returning the values from functions.

### 13.4.2 Modifying an UDF

There is no ALTER statement for altering or modifying an UDF. When a change has to be done to an existing UDF, it will be necessary to drop and recreate the function.

### 13.4.3 Dropping a UDF

To drop an UDF using the SQL interface, use the DROP FUNCTION statement. The DROP FUNCTION statement references the function by:

- **Name**: For example, DROP FUNCTION myUDF. This is only valid if exactly one function of that name exists in that schema. Otherwise, SQLSTATE 42854 (‘More than one found’) or SQLSTATE42704 (‘Function not found’) is signalled.

- **Signature** (name and parameters): For example, DROP FUNCTION myUDF(int). The data type of the parameter(s) must match exactly those of the function found. Also, if length, precision, or scale are specified, they must match exactly the function to be dropped. SQLSTATE 42883 is signalled if a match to an existing function is not found.

- **Specific name**: For example, DROP SPECIFIC FUNCTION myFun0001. Since the SPECIFIC name must be unique per schema, this will find, at most, one function. If the function is not found, SQLSTATE 42704 (“Function not found”) is signalled.
To drop a UDF using iSeries Navigator, you open the required library, right-click the UDF that you want to delete, and select **Delete**, as shown in Figure 13-1.

![Figure 13-1 Dropping functions with iSeries Navigator](image)

If there are no dependent functions, the right panel refreshes, and you should see that the UDF object has been removed from the library.

### 13.5 Resolving a UDF

Resolving to the correct function to use for an operation is more complicated than other resolution operations since DB2 Universal Database supports function overloading. This means that a user may define a function with the same name as a built-in function or another UDF on the system. For example, `SUBSTR` is a built-in function, but the user may define his own `SUBSTR` function that takes slightly different parameters. Therefore, even resolving to a supposedly built-in function still requires that function resolution be performed. The following sections explain how DB2 Universal Database for iSeries resolves references to functions.

#### 13.5.1 UDF overloading and function signature

As mentioned earlier, DB2 UBD for iSeries supports the concept of function overloading. This means that you can have two or more functions with the same name in the same schema, library, or collection, provided that they have a different signature. The signature of a function can be defined as the combination of the qualified function name and the basic data types of the input parameters of the function.

No two functions on the system can have the same signature. The length and precision of the input parameters are not considered to be part of the signature. Only the data type of the input parameters are considered to be part of the signature. Therefore, if you have a function called `DNAME` in library `SAMPLEDB01` that accepts an input parameter of type `CHAR(10)`, you cannot have another function called `DNAME` in the same `SAMPLEDB01` that accepts `CHAR(12)`. However, it is possible to have another function `DNAME` in library `SAMPLEDB01` that accepts an `INTEGER` value as input parameter and another one that accepts `SMALLINT`. The following examples illustrate the concept of the function signature. These two functions can exist in the same schema:

```
SAMPLEDB01.DNAME(int)
SAMPLEDB01.DNAME(smallint)
```

These two functions cannot exist in the same schema:

```
DNAME(char(10))
DNAME(char(5))
```
Notice that certain data types are considered equivalent when it comes to function signatures. For example, CHAR and GRAPHIC are treated as the same type from the signature point of view.

The data type of the value returned by the function is *not* considered to be part of the function signature. This means that you cannot have two functions called DNAME in library SAMPLEDB01 that accept input parameters of the same data type, even if they return values of different data types.

### 13.5.2 Parameter matching and promotion

When an SQL DML statement references a UDF, the system, at first, tries to find an exact match for the function by searching for functions that have the same signature. If the system finds a function having input parameters that exactly match those specified in the DML statement, that function is chosen for execution. In case the system cannot find any function in the path that exactly matches those specified on the DML statement, the parameters on the function call in the DML statement are *promoted* to their next higher type. Then another search is made for a function that accepts the promoted parameters as input. During parameter promotion, a parameter is cast to its next higher data type. For example, a parameter of type CHAR is promoted to VARCHAR, and then to CLOB. There are restrictions on the data type to which a particular parameter can be promoted. We explain this concept with an example.

Let us assume that you have created a table CUSTOMER in library LIB1. This table has, among its other columns, a column named CUSTOMER_NUMBER, which is a CHAR(5). Let us also assume that you have written a function GetRegion that will perform some processing and return the region to which your customer belongs. The data type of the parameter that this function accepts as input is defined to be of type CLOB(50K). Let us assume that there are no other functions called GetRegion in the path. Now, if you execute the following query, you will see that the function GetRegion(CLOB(50K)) is actually executed:

```sql
SELECT GetRegion( customer_number ) FROM customer
```

How is this possible? The field CUSTOMER_NUMBER from the CUSTOMER table has the data type CHAR(5). The function GetRegion actually accepts a CLOB as a parameter, and there are no other functions called GetRegion in the path. In its attempt to resolve the function call, the system first searched the library path for a UDF called GetRegion, which accepts an input parameter of type CHAR. However, no such UDF was found. The system then *promoted* the input parameter, in our case the CUSTOMER_NUMBER, up in the hierarchy list of promotable types to a VARCHAR. Then a search was made for an UDF called GetRegion, which accepted an input parameter of type VARCHAR. Again, no such UDF was found. Then, the system *promoted* the input parameter up the hierarchy list to a CLOB. A search was made for an UDF called GetRegion, which accepted an input parameter of type CLOB. This time the search was successful. The system invoked the UDF GetRegion( CLOB(50K) ) to satisfy the user request.
The concept of parameter promotion is clearly demonstrated in the previous example. Table 13-1 indicates the data types and the data types to which they can be promoted.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Data type precedence list (in best to worst order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR or GRAPHIC</td>
<td>CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB, or DBCLOB</td>
</tr>
<tr>
<td>VARCHAR or VARGRAPHIC</td>
<td>VARCHAR or VARGRAPHIC, CLOB, or DBCLOB</td>
</tr>
<tr>
<td>CLOB or DBCLOB</td>
<td>CLOB or DBCLOB</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT, INTEGER, DECIMAL or NUMERIC, REAL, DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER, DECIMAL or NUMERIC, REAL, DOUBLE</td>
</tr>
<tr>
<td>DECIMAL or NUMERIC</td>
<td>DECIMAL or NUMERIC, REAL, DOUBLE</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL, DOUBLE</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATALINK</td>
<td>DATALINK</td>
</tr>
<tr>
<td>A User Defined Type</td>
<td>The same User Defined Type</td>
</tr>
</tbody>
</table>

As you see from the previous table, data types can be promoted up the hierarchy only to particular data types. Distinct types cannot be promoted. Even though distinct types are based on one of the built-in data types, it is not possible to promote distinct types to anything other than the same type. Parameters cannot be demoted down the hierarchy list as shown in Table 13-1. This means that, if the CUSTOMER_NUMBER column of the CUSTOMER table is a CLOB, and the GetRegion UDF was defined to accept a CHAR as an input parameter, a call, such as the following example, fails because function resolution does not find the UDF:

```sql
SELECT GetRegion( CUSTOMER_NUMBER ) from customer
```

**Note:** CHAR parameters cannot be passed to the function as character literals (‘ABC’), because all character literals are treated as VARCHAR data types and that is not compatible with a fixed-length character data type.

### 13.5.3 Function path and the function selection algorithm

On the iSeries system, there are two types of naming conventions when using SQL. One of them is called the *system naming convention*, and the other one is called the *SQL naming convention*. The system naming convention is native to the iSeries system, and the SQL naming convention is specified by the ANSI SQL standard.

The function resolution process depends on which naming convention you are using at the time you execute the SQL statement, which refers to a UDF.
Functionpath

When unqualified references are made to a UDF inside an SQL statement, DB2 Universal Database for iSeries uses the concept of PATH to resolve references to the UDF. The path is an ordered list of library names. It provides a set of libraries for resolving unqualified references to UDFs as well as UDTs. In cases where a reference to a UDF matches more than one UDF in different libraries, the order of libraries in the path is used to resolve to the correct UDF.

The path can be set to any desired set of libraries using the SQL SET PATH statement. The current setting of the path is stored in the CURRENT PATH special register.

For the SQL naming convention, the path is set initially to the following default value:
"QSYS", "QSYS2", "<USER ID>"

For the system naming convention, the path is set initially to the following default value:
*LIBL

When you are using the system naming convention, the system uses the library list of the current job as the path, and uses this list to resolve the reference to the unqualified references to the UDFs.

The current path can be changed with the SET PATH statement. Note that this statement overrides the initial setting for both naming conventions. For example, you can use the following statement:
SET PATH = MYUDFS, COMMONUDFS

To set the path to the following list of libraries:
QSYS, QSYS2, MYUDFS, COMMONUDFS

Notice that the libraries QSYS and QSYS2 are automatically added to the front of the list. This is the case unless you explicitly change the position of these libraries in the SET PATH statement. For example, the following statement sets the CURRENT PATH registry to myfunc, QSYS, QSYS2:
SET PATH myfunc, SYSTEM PATH

For portability reasons, we recommend that you use SYSTEM PATH registry rather than QSYS and QSYS2 library names on the SET PATH statement.

The function selection algorithm

The function selection algorithm searches the library path for a UDF using the steps outlined here:

1. Finds all functions from the catalog (SYSFUNCS) and built-in functions that match the name of the function. If a library was specified, it only gets those functions from that library. Otherwise, it gets all functions whose library is in the function path.

2. Eliminates those functions whose number of defined parameters does not match the invocation.

3. Eliminates functions whose parameters are not compatible or “promotable” to the invocation.
For the remaining functions, the algorithm follows these steps:

1. It considers each argument of the function invocation, from left to right. For each argument, it eliminates all functions that are not the best match for that argument. The best match for a given argument is the first data type that you see in the precedence list. Lengths, precessions, scales, and the "FOR BIT DATA" attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

2. If more than one candidate function remains after the above steps, it has to be the case (the way the algorithm works) that all the remaining candidate functions have identical signatures but are in different schemas. It chooses the function whose schema is earliest in the user’s function path.

3. If there are no candidate functions, it signals the error SQLSTATE 42884.

13.6 Systems catalog tables

The database manager provides a number of data dictionary facilities that can be used to keep track of UDFs. In this section we see how to view UDF information using the SYSROUTINES catalog, the SYSPARAMS catalog, and the SYSFUNCS view.

13.6.1 SYSROUTINES catalog

UDF reference are stored in the SYSROUTINES catalog. For detailed descriptions of the DB2 Universal Database catalogs, see SQL Reference for Cross-Platform Development at the following Web address:


**Note:** The SYSROUTINES catalog contains details for both UDFs and stored procedures. When you want to see only UDFs you can use a view called SYSFUNCS, or you can select rows in the SYSROUTINES catalog where ROUTINE_TYPE is FUNCTION.

The SQL statement in Example 13-7 displays SYSROUTINES information about UDFs in our test SAMPLERDB01 library.

**Example 13-7 Sample query on SYSROUTINES showing UDFs defined on a specific schema**

```sql
SELECT * 
FROM QSYS2.SYSROUTINES 
WHERE ROUTINE_SCHEMA = 'SAMPLEDB01' 
AND ROUTINE_TYPE = 'FUNCTION';
```

Figure 13-2 shows that in this schema there are two UDFs, DNAME and EMPBYPROY. The first of them is a scalar function, while the second is a table function. Neither of them allows parallelism. EMPBYPROJ is fenced, while DNAME is not.

```
<table>
<thead>
<tr>
<th>SPECIFIC_SCHEMA</th>
<th>SPECIFIC_NAME</th>
<th>ROUTINE_NAME</th>
<th>ROUTINE_BODY</th>
<th>FUNCTION_TYPE</th>
<th>PARALLELIZABLE</th>
<th>FENCED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLEDB01</td>
<td>DNAME</td>
<td>DNAME</td>
<td>SQL</td>
<td>S</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SAMPLEDB01</td>
<td>EMPBYPROJ</td>
<td>EMPBYPROJ</td>
<td>SQL</td>
<td>T</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
```

**Figure 13-2 Content of SYSROUTINES catalog**
13.6.2 SYSPARMS catalog

The SYSPARMS catalog contains one row for each parameter of an UDF created by the CREATE FUNCTION statement.

Suppose that you want to retrieve the parameter details for all instances of the DNAME function located in the SAMPLEDB01 library. You can run the SQL statement in Example 13-8 to display this information.

Example 13-8  Sample query on SYSPARMS showing parameters for DNAME UDF

```
SELECT *
FROM QSYS2.SYSPARMS
WHERE SPECIFIC_SCHEMA = 'SAMPLEDB01'
AND SPECIFIC_NAME IN (
    SELECT SPECIFIC_NAME
    FROM QSYS2.SYSFUNCS
    WHERE SPECIFIC_SCHEMA = 'SAMPLEDB01'
    AND ROUTINE_NAME = 'DNAME');
```

Note that, due to function overloading, the SAMPLEDB01 schema may contain functions with the same routine name. Running this query produced the results shown in Figure 13-3.

There are two instances of the DNAME function in the SAMPLEDB01 library. Their signatures differ since they accept an input parameter of type SMALLINT or INTEGER, respectively. Note, also, that the result of a function is stored in the SYSPARMS catalog as an OUTPUT parameter.

```
```

Figure 13-3  UDF parameter details in SYSPARMS catalog

13.7 Authorization and adopted authority

When a UDF is executed as part of an SQL statement by a client program, the statements in the UDF are executed with the authorities of the user. Or they are executed with the authorities of the user, plus the authorities of the owner of the program object corresponding to that UDF. It depends on how it was defined in the USRPRF attribute at program or service program object creation time.

The authorization and adopted authorities behave as in a stored procedure as explained in 4.7, “Authorization and adopted authority” on page 78.
13.8 Transaction management considerations

Because UDFs are called and executed in the middle of an SQL statement, and SQL statements must be atomic by principle, the UDF should not affect the transactional behavior of their callers. This means that an UDF should not perform COMMIT, ROLLBACK, SAVEPOINT or SET TRANSACTION operations.

13.9 Coding considerations

When coding UDFs, you should keep in mind some of the limitations and restrictions that apply to them. The following list contains important recommendations and hints for UDF developers:

- UDFs should not perform operations that take a long time (minutes or hours).
- UDFs are invoked from a low-level in DB2 that holds resources (locks and seizes) for the duration of the UDF execution.
- If UDF does not finish in an allocated time, the SQL statement fails. You can override the system time-out value with the UDF_TIME_OUT parameter in the query option file QAQQINI. Refer to DB2 UDB for iSeries SQL Programming, SC41-5611, for details.
- Avoid inserts, updates, and delete operations on the same tables as the one referred to in the invoking statement.
- A UDF runs in the same job as the invoking SQL statement, but runs in a separate system thread, so secondary thread considerations apply:
  - UDFs will conflict with thread-level resources held by the SQL statement. UDFs cannot perform any operation that is blocked from secondary threads.
  - Activation Group (*NEW) is not allowed for UDFs.
  - UDFs do not inherit program adopted authority that may have been active. Authority comes from the UDF program itself or the user running the SQL.
SQL user-defined functions

This chapter describes the user-defined functions (UDFs) based on SQL Persistent Stored Module (PSM).

This chapter contains the following topics:
- DB2 Universal Database for iSeries SQL UDF ANSI-SQL implementation
- Language elements of UDF
- Transaction management and UDF
- Calling from client applications
- Debugging SQL UDFs
- Considerations regarding SQL UDF
14.1 Introduction

SQL functions are UDFs that you have defined, written, and registered using the CREATE FUNCTION statement. As such, they are written using only the SQL language and their definition is completely contained within one CREATE FUNCTION statement. The creation of an SQL function causes the registration of the UDF, generates the executable code for the function, and defines the details of how parameters are actually passed to the UDF or UDTF in the database catalog. Therefore, writing these functions is quite clean and provides less chance of introducing errors into the function.

14.2 System requirements and planning

When you execute the CREATE FUNCTION statement for the SQL UDF, DB2 Universal Database for iSeries walks through a multiphase process to create an ILE C service program object (*SRVPGM). During this process, DB2 Universal Database for iSeries generates an intermediary ILE C code with embedded SQL statements. This ILE C code is then precompiled, compiled, and linked automatically.

The same requirements needed for SQL stored procedures and SQL triggers are required for SQL UDFs. Refer to 3.2, “System requirements and planning” on page 20, for complete details of the system requirements.

14.3 Structure of an SQL UDF

An SQL UDF consists of:

- A function name.
- A sequence of parameters declarations.
- The option list or function properties.
- The set option statement, which specifies the parameters that will be used to create the function; for example, to create a debugable function. This is an optional component.
- The SQL routine body that specifies a single SQL statement, including a compound statement.

Example 14-1 shows the general structure of a UDF.

Example 14-1  CREATE FUNCTION prototype statement

```
CREATE FUNCTION name-of-udf
  (List of the input parameters of the UDF)
Returns
Function properties
Generation options
Routine body
```
Let us explore some examples that illustrate this structure.

### 14.3.1 Single SQL statement UDF

A single SQL statement UDF is an UDF whose body is composed of one and only one single SQL statement. A single SQL statement UDF can be either scalar or a table function.

#### Scalar single SQL statement UDF

A scalar single SQL statement UDF is a single SQL statement UDF whose return is a scalar value, as in Example 14-2, in which the total salary is calculated based on its base salary, bonus and commission.

**Example 14-2  Scalar single SQL statement UDF**

```
CREATE FUNCTION SAMPLEDB01.TOTSAL(
  SAL DECIMAL(9, 2),
  BON DECIMAL(9, 2),
  COM DECIMAL(9, 2)
) RETURNS DECIMAL(9, 2)
LANGUAGE SQL
SPECIFIC SAMPLEDB01.TOTSAL01
DETERMINISTIC
CONTAINS SQL
RETURN COALESCE(SAL, 0) + COALESCE(BON, 0) + COALESCE(COM, 0);
```

Notes: The following notes refer to Example 14-1.

1. Every UDF starts with CREATE FUNCTION and its name. The fully qualified name in combinations with the name and basic data type of parameters conforms the UDF signature. Two UDFs with the same signature cannot reside on the same schema.

2. Parameters of the UDF and the data type of each parameter. Although not required, you can give a name for each parameter. A maximum of 90 parameters is allowed in this list; also, a function can have no parameters.

3. Data type and attributes of the output return. You can specify any built-in data type or distinct type except LONG VARCHAR or LONG VARGRAPHIC.

4. You can define certain properties of the UDF. The most important is the language used to code the UDF. In this case this is an SQL UDF, so an SQL language has to be specified. Other characteristics that may be specified here are parallelism, external actions, and level of data access (whether the UDF contains SQL, reads SQL data, or modifies SQL data).

5. You can affect the way in which the function is generated by DB2 Universal Database for iSeries. One of the most relevant options is DBGVIEW, which forces the system to generate debugging information. You can find more details about the options in the SQL Reference manual.

6. The routine body of the function consists of a single SQL statement (SELECT, UPDATE, INSERT, DELETE) or an SQL compound statement (IF, WHEN, FOR, CASE...).
This function has just a single statement that performs an arithmetic operation between values received as parameters and return the result. This UDF may be used in a SELECT, SET or VALUES statement, or clauses such as that shown in Example 14-3.

Example 14-3  Scalar UDF usage

```
SELECT FIRSTNME, LASTNAME, TOTSAL(SALARY, BONUS, COMM) AS TOTAL FROM EMPLOYEE
```

```
UPDATE EMPLOYEE SET SALARY = SALARY * TOTSAL(SALARY, BONUS, COMM) * 0.5
WHERE WORKDEPT = A01;
```

```
INSERT INTO CTRLTBL (EMPNO, WORKDEPT, TOTAL_COMP) VALUES (:EMPNO, :WORKDEPT, TOTSAL(:SALARY, :BONUS, :COMM)
```

**Single SQL statement UDTF**

Example 14-4 shows is a single UDTF returning a table with the employees who are involved in a particular project.

Example 14-4  Single SQL statement UDTF

```
CREATE FUNCTION SAMPLEDB01.EMPBYPRJ ( PRJNBR VARCHAR(6) )
RETURNS TABLE ( EMPNO CHAR(6), FIRSTNME CHAR(20), LASTNAME CHAR(20), BIRTHDATE DATE )
LANGUAGE SQL
SPECIFIC SAMPLEDB01.EMPBYPRJ
RETURN
SELECT EMPNO, FIRSTNME, LASTNAME, BIRTHDATE
FROM SAMPLEDB01.EMPLOYEE WHERE EMPNO IN ( SELECT EMPNO FROM SAMPLEDB01.EMPPROJECT
WHERE PROJNO = PRJNBR );
```
Chapter 14. SQL user-defined functions

14.3.2 Compound SQL statement UDF

A compound SQL statement UDF is a UDF whose body contains a compound SQL statement, which is a BEGIN, followed by one or more SQL statements, followed by an END statement.

Scalar compound SQL statement UDF

In our first example in 14.3.1, “Single SQL statement UDF” on page 469, we just add three values; but in real life, things usually are more complex than that and cannot be solved with just a single SQL statement.

In Example 14-6, we add some complexity to that example by adding a business rule that will enforce the payment of the commission depending on the project assignments an employee has.

Example 14-6  Compound SQL statement UDF

```sql
CREATE FUNCTION SAMPLEDB01.TOTALSAL (P_EMPNO CHAR(6)) 
RETURNS DECIMAL(9, 2) 
LANGUAGE SQL 
SPECIFIC SAMPLEDB01.TOTALSAL2 
NOT DETERMINISTIC 
READS SQL DATA 
RETURNS NULL ON NULL INPUT
BEGIN
  DECLARE TOTAL DECIMAL (9, 2) ;
  DECLARE UTLZTN DECIMAL (5, 2);
  SELECT SUM ( EMPTIME ) INTO UTLZTN FROM SAMPLEDB01.EMPPROJACT
  WHERE EMPNO = P_EMPNO;
  SELECT COALESCE(SALARY, 0) + COALESCE(BONUS, 0) + COALESCE(UTLZTN * COMM, 0)
  INTO TOTAL
  FROM SAMPLEDB01.EMPLOYEE WHERE EMPNO = P_EMPNO;
  RETURN TOTAL ;
END;

COMMENT ON SPECIFIC FUNCTION SAMPLEDB01.TOTALSAL
IS 'Total Salary';
```
The use of this UDF is illustrated in the following statement.

**Example 14-7 Using TOTALSAL UDF**

```sql
SELECT FIRSTNAME, LASTNAME, TOTALSAL(EMPNO) AS TOTAL FROM EMPLOYEE;
```

```sql
SET V_SALARY = TOTALSAL(EMPNO);
```

When writing an UDF, consider that it will be invoked for each row that matches the selection criteria for the SELECT or SET statement where it is going to be used.

### Compound SQL statement UDTF

A compound SQL statement UDTF is a compound SQL statement UDF that returns a table. Let us suppose that we need to have a subset of sales persons containing the best performers, where the best are described as those whose sales are higher. If we want the best four sales people, but two sales people match in fourth place, so we want to have both.

**Example 14-8 Compound SQL statement UDTF**

```sql
CREATE FUNCTION RANK(N INTEGER)
RETURNS TABLE(
    POSITION INTEGER,
    EMPNO CHAR(6),
    FIRSTNAME CHAR(20),
    LASTNAME CHAR(20),
    SALARY DECIMAL(13,2)
)
LANGUAGE SQL
DISALLOW PARALLEL
MODIFIES SQL DATA
NOT FENCED
BEGIN
    DECLARE LAST_SALARY DEC(13,2) DEFAULT 0;
    DECLARE I INTEGER DEFAULT 1;
    DECLARE STMT VARCHAR(255);
    DECLARE TABLE_ALREADY_EXISTS CONDITION FOR '42710';
    DECLARE CONTINUE HANDLER FOR TABLE_ALREADY_EXISTS
        DELETE FROM SESSION.RETURN_TBL;
    DECLARE GLOBAL TEMPORARY TABLE SESSION.RETURN_TBL (POSITION INTEGER NOT NULL,
        EMPNO CHAR(6) NOT NULL,
        FIRSTNAME CHAR(20) NOT NULL,
        LASTNAME CHAR(20) NOT NULL,
        SALARY DECIMAL(13,2) NOT NULL);
```
FOR_LOOP: FOR EACH_ROW AS C1 CURSOR FOR
SELECT EMPNO, FIRSTNME, LASTNAME, SALARY
FROM SAMPLEDB01.EMPLOYEE ORDER BY SALARY DESC DO
  IF (I > N) AND (EACH_ROW.SALARY < LAST_SALARY) THEN
    LEAVE FOR_LOOP;
  ELSE
    SET LAST_SALARY = EACH_ROW.SALARY;
  END IF;
  SET I = I + 1;
  END FOR;
RETURN
SELECT POSITION, EMPNO, FIRSTNME, LASTNAME, SALARY
FROM SESSION.RETURN_TBL;
END;

Note: The following notes refer to Example 14-8.

1 This UDTF creates a Global Temporary Table in order to assemble the result table. If the
Global Temporary Table already exists, an SQL error is fired. An error condition is
defined for readability.

2 Then an error handler is declared. The UDTF tries to create a Global Temporary Table in
but if the table already exists (because the UDF was previously used in the same job),
the error handler catches the exception and cleans up the table.

3 A FOR loop walks through the employee table until the end of the table is reached or
until the requested number of rows is filled in. But, it happens that a match in the last row
should be checked.

5 We reached the end of our UDTF, in which it returns the requested table.

14.4 Creating an SQL UDF

Now that you know the general structure of an SQL UDF, you are ready to create one. This
section documents the steps required to edit and compile an SQL UDF. There are many ways
to build an SQL UDF:

- iSeries Navigator GUI
- iSeries Navigator SQL script utility
- Traditional 5250 programming using SEU and RUNSQLSTM utilities

14.4.1 Creating an SQL UDF with iSeries Navigator

iSeries Navigator provides an attractive graphical interface that allows you to perform typical
database administration tasks. It allows easy access to all server administration tools, gives a
clear overview of the entire database system, enables remote database management, and
provides assistance for complex tasks.

In this section, you learn how to efficiently use the GUI administration tools offered by iSeries
Access Express (5722-XE1) to work with SQL UDF on the iSeries. We expect that you
already know how to set up the iSeries Navigator connection to your iSeries server.
Creating scalar SQL UDF with iSeries Navigator

In this section we recreate the samples defined in 14.3, “Structure of an SQL UDF” on page 468. To create an SQL UDF using the iSeries Navigator.

1. Double-click the **iSeries Navigator** icon on your desktop. Under My Connections, double-click the iSeries server that you are working on, as shown in Figure 14-1.

![Figure 14-1  iSeries Navigator](image)

2. Double-click the **Database** icon and expand the database where the UDF will be located, under Libraries. In our case, the name of the database is MONACO.

3. Right-click the library that will contain your UDF, and select **New -> Function -> SQL**.

4. The New SQL Function window opens.
   a. In the **General** tab (Figure 14-2 on page 475), enter a representative name for the function in the Function input field; in our case the function is called TOTALSAL. In the description input field, type a description of the function. In the “Data returned to invoking statement” box, select the **Single value** radio button if you want to return an scalar or **Table** if you want to create a UDTF. In our TOTALSAL example we choose a single value.
Then select the type of the scalar that you want to return. In our case we select **DECIMAL** from the type drop-down list, and specify a length of 9 with scale 2. This is the same data type as the SALARY, BONUS and COMM column of the EMPLOYEE table that this UDF reads from, and returns it to the calling statement.

At the bottom part of the tab, you can select the following check boxes:

- **Can run in parallel**: Specify that the function can or cannot run in parallel. Table functions cannot run in parallel, so this option is not available when your data returned is **TABLE**.

- **Program does not call outside of itself (No External Action)**: Specify if the function performs some external action such as inserting, updating, or deleting rows in a table, calling a function or stored procedure that performs a external action, such as sending data to a data queue. When external actions are performed, the program must be invoked with each successive function invocation. Because our function does not perform external actions, we check this option for our function.

- **Same result returned from successive calls with identical input (Deterministic)**: You should check this option if the function will always return the same result from successive invocations when identical input arguments are provided. That is the case in our example.

- **Attempt to run in same thread as invoking statement (NOT FENCED)**: This means that the function may (NOT FENCED) or may not run (FENCED) in the same thread as the invoking SQL statement. If the function contains cursors, it is better to run as FENCED. In our example we choose to run as NOT FENCED, that is, it may run in the same thread.
b. In the **Parameters** tab you can insert the input parameters of the function, as shown in Figure 14-3.

![Figure 14-3 Creating a SQL UDF: Parameters tab](image)

C. In the **SQL Statement** tab, you write the SQL code for your UDF as we illustrate for our example in Figure 14-4.

![Figure 14-4 Creating SQL UDF: Statements tab](image)

d. Click **OK**.

You see the function in iSeries Navigator in the right panel of the object list.

**Creating an SQL UDTF with iSeries Navigator**

When you create a table SQL UDF with iSeries Navigator, some of the panels change, allowing to define the table structure to be returned, as well as specific characteristics related to table UDFs. In this section we recreate the table UDF sample defined in 14.3.2, “Compound SQL statement UDF” on page 471. The following steps show how to create an SQL UDF using the iSeries Navigator.

1. Double-click the **iSeries Navigator** icon on your desktop. Under My Connections, double-click the iSeries server that you are working on, as shown in Figure 14-1 on page 474.

2. Double-click the **Database** icon and expand the database where the UDF will be located, under Libraries. In our case, the name of the database is MONACO.

3. Right-click the library that will contain your UDF, and select **New -> Function -> SQL**.

4. The New SQL Function window opens.

   a. In the **General** tab (Figure 14-5), a representative name for the function is typed in the Function input field; in our case the function is called **SALESRANKING**. In the description input field, type a description of the function. In the data returned to invoking statement box, select the **Table** radio button.
Then a description for the returned table should be specified, using the Insert button to define each column.

At the bottom part of the tab, you can select the following check boxes:

- Program does not call outside of itself (No External Action): Specify if the function does or does not perform some external action. When external actions are performed, the program must be invoked with each successive function invocation. Because our function does not perform external actions, we check this option.

- Same result returned from successive calls with identical input (Deterministic): You should check this option if the function will always return the same result from successive invocations with identical input arguments. Our function is not deterministic because it may give us different returns when invoked with the same parameter.

- Attempt to run in same thread as invoking statement (Not Fenced): That means that the function may (unfenced) or may not (fenced) run in the same thread as the invoking SQL statement. If the function contains cursors, it is better to run as Fenced. In our example we choose to run as not fenced.
b. In the **Parameters** tab you can insert the input parameters of the function, as shown in Figure 14-6.

*Figure 14-6  Creating a Table SQL UDF: Parameters tab*
c. In the **SQL Statement** tab you write the SQL code for your UDF, as illustrated for our example in Figure 14-7.

![Figure 14-7 Creating Table SQL UDF: Statements tab](image)

d. Click **OK**.

You see the function in iSeries Navigator in the right panel of the object list.

### 14.4.2 Creating a user-defined function with the Run SQL Scripts utility

The Run SQL Scripts utility is another interface that you can use on the iSeries server to create a UDF. The script utility is available through the iSeries Navigator GUI. It allows you to create, edit, run, and troubleshoot scripts of SQL statements. You can also save the scripts with which you work on your PC.

To create an SQL UDF using the SQL script utility:

1. Double-click the **iSeries Navigator** icon on your desktop.
2. In the main panel, right-click the **Database** object and select the database for which the UDF will be created, and select **Run SQL Scripts**.
3. The Run SQL Scripts window opens. Type the function body as shown in Figure 14-8.

```
CREATE FUNCTION SAMPLEDB01.EMPBYPRJ(
    PRUNBR VARCHAR(8))
RETURNS TABLE (
    EMPNO CHAR(6),
    FIRSTNAME CHAR(20),
    LASTNAME CHAR(20),
    BIRTHDATE DATE)
LANGUAGE SQL
SPECIFIC SAMPLEDB01.EMPBYPRJ
NOT DETERMINISTIC
READS SQL DATA
CALLED ON NULL INPUT
NO EXTERNAL ACTION
DISALLOW PARALLEL
NOT FENCED
RETURN
SELECT EMPNO, FIRSTNAME, LASTNAME, BIRTHDATE FROM
SAMPLEDB01.EMPLOYEE WHERE
EMPNO IN (SELECT EMPNO FROM SAMPLEDB01.EMPLOYEE WHERE PRUNBR = PRUNBR )
```

To run the CREATE FUNCTION statement, select Run -> All from the Run pull-down menu. If there is an error message, you see it in the Message tab.

Syntax errors can be corrected one by one. When the function is free of errors, the last message in the Messages frame of the Run SQL Script window should read:

Statement ran successfully

If the run history panel does not supply sufficient information about the execution of the SQL statement, you can view the iSeries server job log to obtain additional, more specific messages. Select View-> Job Log. The Job Log window opens (see Figure 14-9).
Figure 14-9  Job log window

To save the script that contains the source code for the EMPBYPRJ UDF, select File -> Save As from the script utility menu bar. The Save As window is displayed. In the Save in list combo, open the directory you want to use as your SQL script repository. In our case, we used the d:\sg24_6503\SofyQ directory. Enter empbyprj.sql in the file name input field. Then click Save to return to the Run SQL Script window.

The Run SQL Scripts utility proved to be very useful when we ported the SQL UDF from other DB2 Universal Database platforms to the iSeries server system. We simply copied the scripts to our working directory and changed the file extension from .stp to .sql. Then we can double-click a UDF file from the Windows Explorer window to load the script into the Run SQL Scripts utility.

14.4.3 Creating a user-defined function with traditional 5250 tools

To create SQL UDF with traditional 5250 tools:

1. Create a library if you do not have one already.
2. Create a source physical file. This is the file where the SQL source members are going to be stored.
3. Start a Source Entry Utility (SEU) editing session.
4. Enter the SQL UDF source code.
5. Create the SQL UDF using the Run SQL Statement (RUNSQLSTM) command to issue a CREATE FUNCTION command. This creates a C program object that runs when the function is called. If there are problems generating the function, there is a listing that shows the syntax errors of the source.
6. Invoke the UDF through the SQL CALL statement passing the parameter list.
7. Check for the completion status of the SQL UDF.
To implement this scenario, first create a library and a source file, and then start an editing session:

1. To create a library called SAMPLELIB, type the following CL command at the 5250 emulation prompt:
   ```cl
crtlib lib(samplelib)
```
2. To create a source physical file called QSQLSRC, type the command:
   ```cl
crtsrcpf file(samplelib/qsqlsrc) rcdlen(112)
text('Source physical file for SQL user-defined function')
```
   The CRTSRCPF command creates a source physical file QSQLSRC in the library SAMPLELIB.
3. To start an editing session and create a source member, TOTALSAL, type the command:
   ```cl
   strseu srcfile(samplelib/qsqlsrc) srcmbr(totalsal)
   type(txt) option(2)
   ```
   Entering OPTION(2) indicates that you want to start a session for a new member. The STRSEU command creates a new member, TOTALSAL, in the QSQLSRC file in the SAMPLELIB library and starts an edit session.

   After typing the source, you see a display similar to the example in Figure 14-10.

   ```sql
   Columns . . . :  1  71 Edit         SAMPLELIB/QSQLSRC
   SEU==>
   FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
   ****************** Beginning of data ****************************************
   0001.00 CREATE FUNCTION SAMPLEDB.TOTALSAL (  
   0002.00   SAL DECIMAL(9, 2) ,  
   0003.00   BON DECIMAL(9, 2) ,  
   0004.00   COMM DECIMAL(9, 2) ,  
   0005.00   LASTNAME VARCHAR(15) )  
   0006.00   RETURNS DECIMAL(9, 2)  
   0007.00   LANGUAGE SQL  
   0008.00   SPECIFIC SAMPLEDB.TOTALSAL  
   0009.00   NOT DETERMINISTIC  
   0010.00   READS SQL DATA  
   0011.00   CALLED ON NULL INPUT  
   0012.00   BEGIN  
   0013.00   DECLARE TOTAL DECIMAL ( 9 , 2 ) ;  
   0014.00   IF ( SELECT COUNT ( * ) FROM SAMPLEDB.SALES  
   0015.00     WHERE SALES_DATE BETWEEN '1996-04-01'AND '1996-04-30'  
   0016.00     AND SALES_PERSON = LASTNAME AND SALES > 0 ) >= 16-04-30'  
   0017.00   THEN SET TOTAL = SAL + BON + COMM ;  
   0018.00   ELSE SET TOTAL = SAL + COMM ;  
   0019.00   END IF ;  
   0020.00   RETURN TOTAL ;  
   0021.00   END  ;  
   ******************** End of data ********************************************
   `

   Note: The line in bold in Figure 14-10 indicates a syntax error to produce the error listing.
4. Use the RUNSQLSTM command to create the UDF, as shown in Figure 14-11. We recommend that you use the Debugging view option *LIST and Listing output *PRINT. It is useful for debugging and testing purposes.

![Figure 14-11 Creating the SQL UDF by using the RUNSQLSTM command](image)

5. If there are syntax errors in your source, a message similar to the example shown in Figure 14-12 is issued.

![Figure 14-12 RUNSQLSTM command fails](image)
6. If the RUNSQLSTM command fails to create the SQL UDF (as in our example), go to the listing of the program by typing the WRKSPFL command at the command prompt (Figure 14-13).

7. Choose option number five to display the list of the items that generate the error we show in Figure 14-14.

The Preceding list found an error in record 16. If you return to Figure 14-10 on page 482 you will see that the error is in line 16.

Now you know how to find errors, correct them by editing the session and executing the RUNSQLSTM command again.
After the function is successfully created, two system catalog tables are updated: SYSRoutines and SYSPARMS. The SYSRoutines table contains one row for each function created by the CREATE FUNCTION statement. The SYSPARMS table contains one row for each parameter of a function created by the CREATE FUNCTION statement.

### 14.4.4 Verifying the user-defined function properties

After the UDF is successfully created, verify its properties by using the iSeries Navigator interface:

1. In iSeries Navigator, double-click the SAMPLEDB01 library icon. The right panel displays all DB2 Universal Database for iSeries objects in this library.
2. Right-click the TOTALSAL UDF icon, and select Properties.
3. In the TOTALSAL Properties window that opens, you see three tabs:
   - General page: Specifies the name by which the function is known and the type of UDF.
   - Parameters page: Specifies the parameters that the procedure uses.
   - SQL Statements page: Contains the code for the external SQL program that you are defining as a UDF. You can use the SQL statement examples and fill in the necessary information to make coding SQL easier. After an SQL function is created, the SQL statements cannot be changed.

### 14.5 SQL control statements

DB2 Universal Database for iSeries provides a set of programming constructs (syntactic structures used to write procedural code) to help those writing SQL procedures. In DB2 Universal Database for iSeries, these programming constructs are called control statements. A control statement is one of the statements that can be placed in the routine body of an SQL UDF. Among these constructs, there are:

- Assignment statements
- Conditional control statements
- Iterative control statements
- Calling external procedures
- Compound statements

These control statements are presented in 3.4, “SQL control statements” on page 21, or in the SQL Reference, which you can find on the Web at:


### 14.6 Error handling in SQL UDFs

Error handling can be seen from two perspectives: On one side of the coin, there is a UDF that has to deal with error and warning conditions that occur inside its scope, and that needs a mechanism to communicate error and warning conditions to its caller. On the other side of the coin, there are programs invoking UDFs and that need to catch the warning or error conditions that may be fired from the UDF.

The error handling in UDFs in general is very close related to the error handling in stored procedures. For a general discussion regarding error handling, see Chapter 8, “Stored procedure error handling” on page 221.
In this section, we highlight the particularities related to UDFs. Suppose we have plenty of tables representing dates as DEC(8), but our database manager does not have a built-in function that converts these dates to DATE format. We decided to expand the DBMS functionality by creating a DEC2DATE function that receives a DEC(8) and returning a DATE. In this particular case, the designer of the function wants to receive a NULL date and an SQLSTATE 01HDI when the function is provided with an invalid date.

Example 14-9  DEC2DATE UDF signaling warning message for invalid dates

```sql
CREATE FUNCTION DEC2DATE (
    DATEDEC DECIMAL(8, 0) )
RETURNS DATE
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
RETURNS NULL ON NULL INPUT
NO EXTERNAL ACTION
BEGIN
    DECLARE RESULT DATE ;
    DECLARE InvalidDate CONDITION FOR '22007';
    DECLARE EXIT HANDLER FOR InvalidDate
    BEGIN
        RETURN CAST(NULL AS DATE);
        SIGNAL SQLSTATE '01HDI' SET MESSAGE_TEXT='Invalid date';
    END;
    SET RESULT = DATE(
        SUBSTRING(DIGITS(DATEDEC),1,4) || '-' ||
        SUBSTRING(DIGITS(DATEDEC),5,2) || '-' ||
        SUBSTRING(DIGITS(DATEDEC),7,2));
    RETURN RESULT;
END ;
```

Note: The SIGNAL statement signals the error or warning condition explicitly. For a description of the SIGNAL statement, refer to 8.2.2, “SIGNAL and RESIGNAL” on page 227.

14.7 Example of an UDTF using Global Temporary Tables

Data Definition Language (DDL) statements may be used inside a SQL PSM, including the declaration of Global Temporary Tables, created with the DECLARE GLOBAL TEMPORARY TABLE statement. Global Temporary Tables are tables created temporarily that are not registered in the database catalog. These temporary tables are physically created in the QTEMP library and are automatically dropped when the session of the creating job ends.

Example 14-10 illustrates a Table UDF creating a Global Temporary Table.

Example 14-10  Table UDF creating a Global Temporary Table

```sql
CREATE FUNCTION RANK(N INTEGER)
RETURNS TABLE(
    POSITION INTEGER,
    EMPNO CHAR(6),
    FIRSTNAME CHAR(20),
    LASTNAME CHAR(20),
    SALARY DECIMAL(13,2)
)
LANGUAGE SQL
```
DISALLOW PARALLEL
MODIFIES SQL DATA
NOT FENCED
BEGIN
    DECLARE LAST_SALARY DEC(13,2) DEFAULT 0;
    DECLARE I INTEGER DEFAULT 1;
    DECLARE STMT VARCHAR(255);
    DECLARE TABLE_ALREADY_EXISTS CONDITION FOR '42710';
    DECLARE CONTINUE HANDLER FOR TABLE_ALREADY_EXISTS
        DELETE FROM SESSION.RETURN_TBL;

    DECLARE GLOBAL TEMPORARY TABLE SESSION.RETURN_TBL (  
        POSITION INTEGER NOT NULL,
        EMPNO CHAR(6) NOT NULL,
        FIRSTNME CHAR(20) NOT NULL,
        LASTNAME CHAR(20) NOT NULL,
        SALARY DECIMAL(13,2) NOT NULL);

    FOR_LOOP: FOR EACH_ROW AS C1 CURSOR FOR
        SELECT EMPNO, FIRSTNME, LASTNAME, SALARY
        FROM SAMPLEDB01.EMPLOYEE ORDER BY SALARY DESC DO
            IF (I > N) AND (EACH_ROW.SALARY < LAST_SALARY) THEN
                LEAVE FOR_LOOP;
            ELSE
                SET LAST_SALARY = EACH_ROW.SALARY;
            END IF;
            INSERT INTO SESSION.RETURN_TBL  
            VALUES ( I, EACH_ROW.EMPNO, EACH_ROW.FIRSTNME,  
                     EACH_ROW.LASTNAME, EACH_ROW.SALARY);
            SET I = I + 1;
        END FOR;
    RETURN
        SELECT POSITION, EMPNO, FIRSTNME, LASTNAME, SALARY
        FROM SESSION.RETURN_TBL;
END;

Notes: The following notes refer to Example 14-10:

1. A Global Temporary Table was created. When the DECLARE GLOBAL TEMPORARY TABLE statement does not explicitly qualify the table, it is qualified with a schema named SESSION. If it is qualified, the qualifier must be SESSION.

2. A DML SQL statement is performed against the RETURN_TBL temporary table. The temporary table has to be qualified with the SESSION schema. If you prefer an implicit qualification, you should assure current schema to be SESSION. This can be done using the statement SET SCHEMA = SESSION.

3. Another reference to a Global Temporary Table is used.

Global Temporary Tables must be implicitly or explicitly qualified with the SESSION schema, regardless of whether a library exists with that name. In a case where a library exists that contains a table with the same name of the Global Temporary Table that is being created, it does not interfere with the operation. The Global Temporary Table is stored in the QTEMP library.
After this UDF finishes and returns its result, the Global Temporary Table remains alive. Then, if subsequent uses of the same Global Temporary Table are performed in the same session scope, the DECLARE GLOBAL TEMPORARY TABLE at line 1 fails with SQLSTATE 42710 because the table already exists. For this reason, you must be careful when declaring Global Temporary Tables in triggers and scalar UDFs that are executed repeatedly many times.

By *session scope*, we mean that the Global Temporary Table will exist until the connection to the database is closed or until the job ends. To prevent this situation, the sample code includes an error handler for SQLSTATE 42710.

### 14.8 Debugging UDFs

When you are developing any kind of software, it is important to have a debugging tool. Debugging allows you to detect, diagnose, and eliminate run-time errors in a program. This section shows you debugging alternatives to test SQL PSMs.

Remember that when you create an SQL PSM, it is really creating an ILE C program underneath. For this reason, one of the alternatives for debugging an SQL UDF is by using the ILE source debugger for testing. In V5R2 DB2 Universal Database for iSeries simplified the debug of SQL stored procedure, functions and triggers with the SQL "SOURCE debug view and by the use of the Toolbox for Java iSeries System Debugger. For a complete explanation of the graphical debugger, refer to 3.10.1, “Graphical debugger” on page 38.

In this section, we show how to debug UDFs. SQL UDFs are always created as service programs. We recommend that you create external functions as service programs. Therefore, we show how to debug a service program here. The same technique needs to be used if you want to debug a program object that is being referenced by an external UDF.

In this example, we debug our TOTSAL UDF. Debugging UDFs may be a bit tricky since they are run on the OS/400 system in secondary threads. The following steps outline the debug procedure:

1. Open two native OS/400 5250 sessions and sign on to both sessions. From here onwards, we refer to the first session as **Session A** and to the second session as **Session B**.
2. Switch to Session B, and type the following command on the command line:
   ```sql
   STRSQL
   ```
   The interactive SQL session is started, and the SQL command line is displayed.
3. Switch to Session A and type the following command line:
   ```sql
   WRKACTJOB
   ```
   The Work with Active Jobs screen is displayed as shown in Figure 14-15. This screen displays a list of all jobs that are currently active on the system. The job in Session B will be listed among these.
4. After you find the job, in our case QPADEV0002, use option 5 to work with that job. Then the Work with Job window is displayed. Choose option number two and write these characteristics:

- **Job:** This is the name of the job with which you are working.
- **User:** This is the name of the user profile that is using the job.
- **Number:** This is the number assigned to the job you are working with. Every job on the OS/400 system is assigned a six-digit unique job number.

In our case this data are:

<table>
<thead>
<tr>
<th>Job</th>
<th>User</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPADEV0002</td>
<td>FREDYC</td>
<td>016859</td>
</tr>
</tbody>
</table>

Now start a service job for the session B job. Enter the following command on the command line:

```
STRSVRJOB 016859/FREDYC/QPADE0002
```

5. Start a debug session for the service program used in the TOTSAL function. Type the following command on the command line:

```
STRDBG UPDPROD(*YES) SRVPGM(SAMPLEDB01/TOTSAL)
```

6. You see the debug session on your screen with the source code loaded into the debugger. In this example we will analyze the ILE C function that the UDF generates. Go to the function SQLPROC1. This is the "IF" in SQL transformed in ILE C. Put a breakpoint in one of the executable statement in the program. In our case we choose this statement:

```
if (SQL_STRUCT_HV.SQL_DATA_RETURNED == '1')
```

This can be done by placing the cursor on the line of code at which you want to place the breakpoint and pressing the F6 key.

You see the following message at the bottom of the screen:

`Breakpoint added to line 396.`
This procedure is shown in Figure 14-16.

7. Press F12. This takes you back to the command line. Now you need to invoke the UDF from the interactive SQL run in session B.

8. Switch to session B and type the SQL statement that runs that function:

   SELECT FIRSTNME, LASTNAME, TOTSAL(SALARY, BONUS, COMM, LASTNAME) AS TOTAL FROM EMPLOYEE

The SELECT statement begins to execute. The TOTSAL (DECIMAL, DECIMAL, DECIMAL, VARCHAR) UDF is invoked. The following message is displayed at the bottom of the screen:

   Query running. 0 records selected, 1 processed.

This is shown in Figure 14-17. However, the result of the query does not show up. Instead, the session busy cross sign stays at the bottom of the screen.
9. Switch back to session A. You see the source code of the PICTCHECK service program displayed on the screen. The line of source code that is to be currently executed is highlighted in white on the screen. In our case, this is the line at which you set the breakpoint in step 6.

10. Press the F10 function key to execute the highlighted line of code. The line is executed and gets de-highlighted. The next line of code to be executed is highlighted. Each time you press the F10 key, the next line of code in sequence is executed.

11. You can check the value contained in any of the program variables. This can be done in two ways:
   - Pressing the F11 key after placing the cursor over the variable for which you want to check the value
   - Typing the EVAL command on the debug command line

We now check the value of the program variable SQLCODE. Place your cursor over the variable and press F11. The value of the variable is displayed on the bottom of the screen, as shown in Figure 14-18.

---

**Figure 14-18 Checking the value of the program variables using F11 key**

12. Continue to press the F10 key until you step through the entire program. At any time, you can run the program to completion by pressing the F12 key.

13. After you finish debugging the code, you return to the Work with Job display. On the command line, type the following CL commands:

   ```cl
   ENDDBG
   ENDSRVJOB
   ```

This ends the debug mode and the service job being run to debug the service program.
External user-defined functions

External user-defined functions (UDFs) are coded in one of the high-level languages available on the iSeries server. If you want to perform complex sophisticated processing, or plan to re-use code that already exists, external UDFs are the best choice for you.

This chapter describes external UDF written in high-level languages. It explains how to register and code external UDF. It also reviews the difference in coding external UDFs regarding the different parameter styling supported by DB2 Universal Database for iSeries and how to invoke external UDF and deal with error handling.

All the benefits of the UDF discussed in Chapter 13, “User-defined functions” on page 449, also apply to external UDFs.
15.1 User-defined function considerations

An UDF is one that has been written by the user in one of the programming languages on the iSeries system. External UDFs can be written in C, C++, RPG, COBOL, CL and Java. This allows SQL programmers to invoke business calculations or processes written on one of the listed languages from an SQL statement without having to know how the UDF is implemented. You can create external UDF based on programs or service programs. To create an external UDF, the high-level language source code should be compiled and the program or service program object created.

When an external UDF associated with an ILE external program or service program is created, an attempt is made to save the function’s attributes in the associated program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

The attributes can be saved for external functions subject to the following restrictions:
- The external program library must not be SYSIBM, QSYS, or QSYS2.
- The external program must exist when the CREATE FUNCTION statement is issued.
- The external program must be an ILE *PGM or *SRVPGM object.
- The external program or service program must contain at least one SQL statement.

When an external UDF is invoked, it runs in whatever activation group was specified when the external program or service program was created. However, ACTGRP(*CALLER) should normally be used so that the UDF runs in the same activation group as the calling program.

To be able to run Java functions, you should have the Developer Kit for Java (5722-JV1) installed on your system. Otherwise, an SQLCODE of -443 will be returned and a CPDB521 message will be placed in the job log.

15.2 Registering an external UDF

Before using an UDF, it should be registered in the database using the CREATE FUNCTION statement. When an external UDF is registered within the database, entries are made into SYSRoutines and SYSPARMS system catalog tables. The content of these system catalog tables is discussed in 4.4, “System catalog tables” on page 74.

15.2.1 Registering an external UDF with iSeries Navigator

The following steps show how to create an external scalar and table UDF using iSeries Navigator.

Registering a external scalar UDF

In the following example, we register an external UDF based in a RPG program, that converts a decimal number representing a date in format YYYYMMDD to a DATE value like 2003-10-10. To see the source code of this example, refer to 15.6.1, “Coding the SQL parameter style” on page 512.
1. Double-click the **iSeries Navigator** icon on your desktop. Expand **My Connections** and the iSeries server that you are working on.

2. Expand the **Database** icon and select the database where the UDF will be created. Expand **Libraries**, and right-click in the library where the UDF will be located. In our case, the name of the library is **SAMPLEDB01**. Select **New -> Function -> External** as shown in Figure 15-1.

![Figure 15-1 Creating an external UDF with iSeries Navigator](image-url)
3. The New External Function window (Figure 15-2) opens.
   a. In the General tab, enter a meaningful name for the UDF in the Function input field; in our case the function is called DEC2DATE. In the Description input field, type a description of the function. In the “Data returned to invoking statement” box, select the Single value radio button if you want to return a scalar, or Table if you want to create a user-defined table function (UDTF). In our DEC2DATE example, we choose Single value and the return type is DATE.

![New External Function window](image.png)

Figure 15-2 Creating External UDF - General tab

At the bottom part of the tab, you are presented with the following check boxes:

- Can run in parallel
- Program does not call outside of itself (No External Action)
- Same result returned from successive calls with identical input (Deterministic)
- Attempt to run in same thread as invoking statement (Not Fenced)

The explanations of these options are the same that we discussed in 14.4, “Creating an SQL UDF” on page 473.

In addition we define the data access and the specific name. The data access in our example is NO SQL, because this UDF will not execute any SQL statement inside. The specific name establishes a unique name for the function.
b. In the **Parameters** tab you define the input parameters for the UDF, as shown in Figure 15-3.

![Figure 15-3 Creating External UDF - Parameter tab](image)

In addition, we select the parameter style. Parameter styles are explained in 15.3, “Parameter styles in external UDFs” on page 506.

c. In the **External Program** tab you write the characteristics and the location of the external program. In our case the program is an RPG that returns a DATE.
Registering an external UDTF

In the following example, we want to register an external UDTF based in a C program. This program reads data from a stream file in the IFS and returns the result as a table. The source program is in 15.4, “Scratchpad in UDFs and UDTFs” on page 510.

1. Double-click the iSeries Navigator icon on your desktop. Under My Connections, double-click the iSeries server that you are working on.

2. Double-click the Database icon, and select and expand the database where the UDTF will be created. Expand the libraries, and select the library where the UDTF will be contained. In our case, the name of the library is SAMPLEDB01. Select New -> Function -> External.

3. The New External Function windows opens.

   a. In the General tab, enter a meaningful name for the UDF in the Function input field; in our case the function is called F1Results. In the description input field, type a description of the function. In the “Data returned to invoking statement” box, select the Table radio button. In our F1Results example we choose the following columns:

      - DRIVER_NBR
      - DRIVER_NAME
      - GENERAL_POSITION
Chapter 15. External user-defined functions

- LAST_RACE_POSITION
- CONSTRUCTOR
- WHEEL_BRAND
- DRIVERS_COUNTRY
- YTD_POOLS
- YTD_WINS
- YTD_POINTS

Figure 15-5  Creating External UDTF - General tab

At the bottom part of the tab, you are presented with the following check boxes:

- Program does not call outside of itself (No External Action)
- Same result returned from successive calls with identical input (Deterministic)
- Attempt to run in same thread as invoking statement (Not Fenced)

In addition, we define the data access and the specific name. The data access in our example is NO SQL, because this UDF will not execute any SQL statement. The specific name establishes a unique name for the function.
b. In the **Parameters** tab you define the input parameters for the UDTF. In our case this will be the name of the file in the iSeries IFS.

Also, we define the parameter style, and an amount of memory for the scratchpad. In our case we choose DB2SQL because it is not a JAVA program, and we define a space of 100 bytes for the scratchpad memory space.

Figure 15-6   Creating External UDF - Parameter tab
c. In the **External Program** tab you write the characteristics and the location of the external program. In our example the C program is a service program, with an entry point readFileToTable. A program (*PGM) object is identified by the library and program name. A service program may contain multiple entry points. When the entry point name differs from the object name, it is identified by the object name followed by the entry point in parenthesis, as shown in Figure 15-7.

![Creating External UDTF - External Program tab](image)

**Figure 15-7 Creating External UDTF - External Program tab**

### 15.2.2 Registering a Java UDF with iSeries Navigator

The following steps show how to create a Java scalar and table UDF using the iSeries Navigator.

#### Registering a Java scalar UDF

In the following example, we register an external UDF based in a Java program. The program calculates the number of the working days between two dates.

1. Double-click the **iSeries Navigator** icon on your desktop. Under My Connections, double-click the iSeries server that you are working on.
2. Double-click the **Database** icon and select the database where the Java UDF will be registered. Expand the libraries and right-click the library where the Java UDF will reside. In our case, the name of the library is SAMPLEDB01.
3. Select **New -> Function -> External** from the pop-up menu.
4. The New External Function window opens.

   a. The **General** tab is shown in Figure 15-8. Enter a meaningful name for the UDF in the Function input field; in our case the function is called WORKING_DAYS. In the Description input field, type a description of the function. In the “Data returned to invoking statement” box, select the **Single value** radio button. In our WORKING_DAYS example we choose **Single value** and the return type is BIGINT.

   ![Figure 15-8 Creating External UDF Java - General tab](image)

At the bottom part of the tab, you are presented with the following check boxes:

- Can run in parallel
- Program does not call outside of itself (No External Action)
- Same result returned from successive calls with identical input (Deterministic)
- Attempt to run in same thread as invoking statement (Not Fenced)

In addition we define the data access and the specific name. The data access in our example is READS SQL DATA, because this UDF will execute a SELECT statement in a table that contains the holidays for a specific calendar. Each country has its own holidays at different dates, and even in a country we may have different calendars for specific purposes.

The specific name is defined as WDAYS01.
b. In the **Parameters** tab (Figure 15-9) you define the input parameter. For example, we define the initial date, final date, and an identifier (CALENDARID) for a particular calendar.

In this case our program is developed in Java, and for Java, JAVA, and DB2GENERAL parameter styles are supported. We choose Java parameter style because it is more portable among platforms. Notice that the iSeries Navigator GUI does not allow DB2GENERAL parameter style. If a Java UDF or UDTF with DB2GENERAL parameter style has to be created, you will need to use another creation method, such as RUN SQL Statement, to create it.

Finally, we choose the option **Return null on null input**. In this way, when at least one of the parameters in the invocation is null, DB2 Universal Database will not execute the UDF, and will return a null.
c. In the **External Program** tab you write the characteristics and the location of the external program. As in the case of Java stored procedures, Java UDFs are implemented as methods of a class. One class may have multiple methods for implementing both UDFs and stored procedures. That class has to be located on the /QIBM/UserData/OS400/SQLLib/Function path. The method is identified by the class name followed by a dot or an exclamation mark followed by the method name, as shown in figure Figure 15-10. The Java class and method name is case sensitive. For performance reasons, we recommend that you use Java archive (JAR) files. DB2 Universal Database for iSeries provides some built-in stored procedures for managing JAR files presented in 7.7, “SQLJ procedures to manipulate JAR files” on page 198.

![Figure 15-10 Creating external UDF Java - External Program tab](image)

**Note:** Be careful when you define the type returned by the program. You can check the data type compatibility in Table 7-1 on page 177.

**Registering a Java UDTF**

In the following example, we register an external UDTF based in a Java program. This program returns a table that exposes the properties set in Java virtual machine (JVM) used for Java stored procedures and Java UDFs.

504 Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries
1. Double-click the iSeries Navigator icon on your desktop.

2. Under My Connections, double-click the iSeries server that you are working on. Double-click the Database icon and select the database where the Java UDTF will be registered. Expand the libraries and right-click the library where the Java UDTF will reside. In our case, the name of the library is SAMPLEDB01. Select New -> Function -> External.

3. The New External Function window opens.

   a. In the General tab (as shown in Figure 15-11), enter a meaningful name for the UDF in the Function input field; in our case the function is called JVMProperties. In the description input field, type a description of the function. In the “Data returned to invoking statement” box, select the Table radio button. In our JVMProperties example we choose a table with two columns, one with the property and the other with the value; both are varchar of 500.

![New External Function in SAMPLEDB01 - JVMProperties](image)

Figure 15-11 Creating external UDTF Java - General tab
b. In the Parameters tab we defined the parameters to be passed to the UDTF. Notice that even if no parameters are going to be passed to the UDTF, a parameter style has to be selected. In this case the parameter style is DB2GENERAL because it is the only parameter style supported by Java UDTFs.

![Parameters tab image](image1)

Figure 15-12 Creating external UDTF Java - Parameters tab

...c. In the External Program tab you write the characteristics and the location of the external program, as shown in figure Figure 15-13. The rules for identifying the method that implements the UDTF are the same as in the scalar Java UDF and stored procedure already presented.

![External Program tab image](image2)

Figure 15-13 Creating external UDTF Java - External Program tab

### 15.3 Parameter styles in external UDFs

The *parameter style* is used to specify the conventions used for passing parameters to and returning values from functions.

You can specify several different parameter styles for an external UDF. On the external function invocation, DB2 Universal Database passes a number of parameters to the function in *addition* to those that you provide as input parameters. The number and type of extra parameters passed by DB2 Universal Database depend on the parameter style. You can specify the required parameter style at the time the function is created. DB2 Universal Database for iSeries supports six parameter styles:

- SQL
- DB2SQL
- GENERAL
- GENERAL WITH NULLS
- DB2GENERAL
- Java
Those parameter styles are the same that were presented in 6.2, “Parameter styles in external stored procedures” on page 129, and 7.2, “Coding DB2 Universal Database for iSeries Java stored procedures” on page 174.

In this section we discuss the parameters that apply to UDFs. We also provide examples for each of these parameter styles.

### 15.3.1 SQL parameter style

The SQL parameter style conforms to the industry standard SQL. This parameter style can be used only in scalar UDFs. The required set of parameters for this parameter style are:

```c
ExternalUDF(
  IN parameter (repeated),
  OUT result,
  IN parameter null indicator (repeated),
  OUT result null indicator,
  OUT sqlstate,
  IN function name,
  IN specific name,
  OUT diagnostic message)
```

The main differences regarding SQL parameter style in UDFs compared to stored procedures are:

- **Parameters are input parameters**: All parameters are input parameters, except for the last one, which is the result.

- **Result**: This parameter is set by UDF before returning to DB2. DB2 UDF for iSeries provides the storage for the returned value.

- **Result indicator**: This parameter should be set by the UDF before returning to DB2. It is a two-byte signed integer that, if set to a negative value, the UDF result will be interpreted as null; and if set to zero or positive, the result will be interpreted as not null.

- **Function name**: This parameter is set by DB2 before calling the UDF. It is a VARCHAR(139) containing the fully qualified function name, following the SQL naming standard.

### 15.3.2 DB2SQL parameter style

All the parameters passed to a function for the SQL parameter style are also passed to a function with the DB2SQL parameter style. However, DB2SQL parameter style allows additional parameters to be passed. This parameter style can be used for both scalar and table UDF. The supported set of parameters for this parameter style are:

```c
externalUDF(
  IN parameter (repeated),
  OUT result,
  IN parameter null indicator (repeated),
  OUT result null indicator,
  OUT sqlstate,
  IN function name,
  IN specific name,
  OUT diagnostic message,
  INOUT scratchpad,
  IN call type,
  IN dbinfo)
```
The additional parameters, not covered in the previous section, are explained in the following list:

- **Scratchpad**: This parameter is set by DB2 before calling the UDF if the SCRATCHPAD clause was specified in the CREATE FUNCTION statement. This can be used by the UDF as an area where temporary values may be saved to be used between consecutive calls in the same statement scope. It can save the results of the last call in between calls to the UDF. Each invocation of the UDF will be able to see the results stored by the last invocation in the scratchpad. On the first call to the function, the contents of the scratchpad are initialized to zeros. Data can be stored into the scratchpad area by an UDF only during the processing of a given SQL statement. That can be very important for a UDTF used in a join or subquery. If it is necessary to maintain the content of the scratchpad across OPEN calls, then FINAL CALL must be specified in your CREATE FUNCTION statement. With FINAL CALL specified, in addition to the normal OPEN, FETCH, AND CLOSE calls, the table functions will also receive a *first* and a *final* call. These first and final calls may be used for scratchpad maintenance and resource releasing.

- **Call type**: This argument is set by DB2 before calling the UDF. For scalar functions it is only present if the CREATE FUNCTION statement of the UDF specified the FINAL CALL keyword, for table functions, is always present. These are the three values for scalar UDF:
  - 1 First call to UDF.
  - 0 Normal call to UDF. All the normal input argument values are passed.
  - 1 Final call to UDF. No SQL argument or SQL argument indicator values are passed. A UDF should not return any answer using the SQL result, SQL result indicator, SQL state or Diagnostic message arguments.

For UDTFs:

- -2 First call to UDF.
- -1 Open call to UDF. The scratchpad is initialized if NO FINAL CALL is specified. ALL SQL argument values are passed.
- 0 Fetch call to UDF. DB2 expects the table function to return either a row comprising the set of the result values, or an end of table condition indicated by SQLSTATE ‘02000’.
- 1 Close call to UDF. This call balances the OPEN call, and can be used to perform any CLOSE processing and resource release.
- 2 Final call to UDF.

This parameter is normally used with the SCRATCHPAD parameter. On the first call, the scratchpad area is set up by the function and then used in subsequent normal calls. On the last call to the function, the scratchpad area is cleaned up. This is an optional input parameter.

- **dbinfo**: A parameter for the DBINFO structure if the DBINFO clause is specified on the CREATE FUNCTION statement. Refer to the `sqludf.h` include file found in the QSYSINC library for a detailed definition of this structure.
15.3.3 GENERAL parameter style

The supported set of parameters for this parameter style is:

\[
\text{externalUDF(IN arguments (repeated))}
\]

When this parameter style is used, the result will be the value the program returns. For this reason, this parameter style can only be used with scalar UDFs.

15.3.4 GENERAL WITH NULLS parameter style

This parameter style can be used only with scalar UDFs. With this parameter style, the parameters are passed into the program or service program as follows:

\[
\text{ExternalUDF(IN parameter (repeated),}
\]
\[
\text{IN parameter indicator array,}
\]
\[
\text{OUT result indicator)}
\]

**Parameter indicator array**

This can be used by the UDF to determine if one or more parameters are not null. Each entry of the array is set to one of the following values:

- \( 0 \) Meaning that the parameter is present (not null)
- \( -1 \) Meaning that the parameter is empty or null

This parameter is treated as input only.

15.3.5 DB2GENERAL parameter style

Parameter style DB2GENERAL is one of the two parameter styles supported in Java UDFs. In this parameter style, the return value is passed as the last parameter of the function and must be set using a `set` method inherited from the `com.ibm.db2.app.UDF` class. When coding an UDF with DB2GENERAL parameter style, follow these conventions:

- The class including the Java UDF should extend, or be a subclass of the `com.ibm.db2.app.UDF` Java class.
- The Java method should be a `public void` method.
- The parameters for the Java method should be SQL compatible. Refer to 7.2.2, “Data type compatibility” on page 177.
- The Java method may test for an SQL NULL value using the `isNull()` method.
- The Java method should explicitly set the return parameter using the `set()` method.

The main advantages of DB2GENERAL parameter style over the Java parameter style are:

- It allows you to test for null parameters of any data type, including those data types that map to Java data types not supporting null values such as `INTEGER`.
- It supports UDTFs.

DB2GENERAL parameter style has a disadvantage that has to be considered: It is not standard, making it less portable than the Java parameter style.
15.3.6 Java parameter style

The following conventions should be considered when you create an UDF with Java parameter style:

- The Java method should be `public static`.
- The Java method should return an SQL compatible type. The returned value is the result of the UDF.
- The Java method may test for an SQL NULL for Java types that permit null values.

15.4 Scratchpad in UDFs and UDTFs

The scratchpad is a memory area provided by DB2 Universal Database for iSeries and that is conserved along the statement scope. The statement scope is the set of calls that a UDF receives for a reference of it in a single SQL statement. The scratchpad will contain an 8-byte binary number (equivalent to a C `long` field) containing the size of the scratchpad followed by a byte array of the specified size.

The size of the scratchpad is established in the SCATCHPAD clause of the CREATE FUNCTION statement. If a size is not specified, the size will be set to the default size, which is 100 bytes.

In the following SQL statement:

```sql
SELECT UDF_W_SCRATCHPAD(ORDERDATE), UDF_W_SCRATCHPAD(SHIPDATE)
FROM SAMPLEDB01.ITEM_FACT
WHERE SHIPMODE = 'EXPRESS'
```

DB2 Universal Database for iSeries will reserve two memory areas for scratchpad, one for each reference to the UDF_W_SCRATCHPAD function. We say that in this case we have two statement scopes for the UDF_W_SCRATCHPAD function. The scratchpad is initialized with binary zeros by the database manager before calling the UDF for the first time in a statement scope.

For UDTFs, the FINAL CALL clause of the CREATE FUNCTION statement affects the way in which the scratchpad is initialized. If FINAL CALL is specified, the scratchpad is initialized before the first call in a statement scope. If NO FINAL CALL is specified or defaulted for a table function, the scratchpad will be initialized for each OPEN call.

Scratchpad may be used as a mechanism to allow the UDF to conserve information between calls. In the Example 15-7 on page 518, the scratchpad is used to keep track of the next number to be generated as ID. In the Example 15-16 on page 528 a scratchpad is used to maintain information regarding a stream file opened at the first call and read among callings to form the return table.

**Note:** SCATCHPAD may be used with DB2SQL or DB2GENERAL parameter style.
15.5 UDF and UDTF calling sequence

Now that we know what an UDF and an UDTF is, understanding when our external programs called and how many times becomes our objective.

Basically, the external program implementing a UDF or UDTF is called once at the beginning of the statement scope, with the first row, once for each one of the subsequent retrieved rows in the statement and once at the end of the statement.

Note the following SQL statement:

```sql
SELECT EMPNO, SALARY*FX_RATE('US', 'COL', CURRENT DATE)
FROM EMPLOYEE
WHERE WORKDEPT = 'A11';
```

If EMPLOYEE has 10 rows for which the WORKDEPT is ‘A11’, then our FX_RATE UDF is called eleven times. Depending on the parameter style used, the call type may be retrieved by:

- **DB2SQL parameter style**: An input parameter provided by the database manager, as illustrated in the example presented in 15.3.2, “DB2SQL parameter style” on page 507.

- **DB2GENERAL parameter style**: Using the getCallType() Java method, inherited from the UDF class. The values returned by this method are the same values described in 15.3.2, “DB2SQL parameter style” on page 507.

The UDF class provides constant definitions for making your Java program code more readable. For C programmers, the sqludf.h header file defines the same set of constants. Table 15-1 shows constants for scalar UDFs.

<table>
<thead>
<tr>
<th>Constant name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLUDF_FIRST_CALL</td>
<td>First call, only made if FINAL CALL was specified in the CREATE FUNCTION statement</td>
</tr>
<tr>
<td>SQLUDF_NORMAL_CALL</td>
<td>Fetch next row</td>
</tr>
<tr>
<td>SQLUDF_FINAL_CALL</td>
<td>Final call, only made if FINAL CALL was specified in the CREATE FUNCTION statement</td>
</tr>
</tbody>
</table>

Table 15-2 shows the constants for UDTFs.

<table>
<thead>
<tr>
<th>Constant name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLUDF_TF_FIRST</td>
<td>First call, only made if FINAL CALL was specified in the CREATE FUNCTION statement</td>
</tr>
<tr>
<td>SQLUDF_TF_OPEN</td>
<td>Open table</td>
</tr>
<tr>
<td>SQL_TF_FETCH</td>
<td>Fetch next row</td>
</tr>
<tr>
<td>SQL_TF_CLOSE</td>
<td>Close table</td>
</tr>
<tr>
<td>SQLUDF_TF_FINAL</td>
<td>Final call, only made if FINAL CALL was specified in the CREATE FUNCTION statement</td>
</tr>
</tbody>
</table>
15.6 Coding an external UDF

External UDFs are regular HLL programs that are registered in DB2 Universal Database for iSeries. They do not differ significantly from a regular HLL program you have written, except for some conventions that have to be observed in order to coordinate the parameter passing between DB2 and your HLL program.

In this section, we present small programs to illustrate each one of the parameter passing techniques, as well as error handling and scratchpad usage.

15.6.1 Coding the SQL parameter style

This section looks at examples on how to code external UDF with SQL parameter style. It also demonstrates how the parameters passed by DB2 Universal Database for iSeries to the external UDF can be used within the function.

Let us suppose that you want to create a function, DEC2DATE, that converts a nonstandard date defined as DECIMAL(8,0) and stored in YYYYMMDD format to DATE data type, which in fact is a very common problem in real life.

Let us examine the CREATE FUNCTION statement for the DEC2DATE external UDF.

Example 15-1  CREATE FUNCTION for the RPG SQL parameter style of DEC2DATE

```
CREATE FUNCTION SAMPLEDB01.DEC2DATE2 (
    DECDATE DECIMAL(8, 0)
) RETURNS DATE
    LANGUAGE RPGLE
    SPECIFIC SAMPLEDB01.DEC2DATERPG
    DETERMINISTIC
    NO SQL
    RETURNS NULL ON NULL INPUT
    NO EXTERNAL ACTION
    NOT FENCED
    EXTERNAL NAME 'SAMPLEDB01/DEC2DATE'
    PARAMETER STYLE SQL ;
```

CREATE FUNCTION statement explanation

The following notes refer to Example 15-1.

1 Qualified procedure name: If qualification is not provided, the implicit qualification rules will apply, as described in 5.8, “Implicit object qualification and authorization resolution” on page 117. This function receives one input parameter called DECDATE of type DECIMAL(8,0). The parameter name is just for documentation purposes and does not have to be related to parameter or variable names used in the program. Function overloading rules apply, as explained in 13.5.1, “UDF overloading and function signature” on page 460.

2 Return type of the function. In this case the return is of DATE type.

3 LANGUAGE clause of the CREATE FUNCTION statement. The LANGUAGE clause specifies the language that was used to implement the UDF. In the example, it is ILE RPG. This information helps the database to pass parameters to the external UDF in the format required by the programming language.
External UDFs can be written in any of the following languages:

- C
- C++
- COBOL
- COBOLLE
- FORTRAN
- JAVA
- PLI
- RPG
- RPGLE

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program at the time the function is created. The language of the program is assumed to be C as default if:

- The program attribute information associated with the program does not identify a recognizable language.
- The program cannot be found.

4 **SPECIFIC NAME** clause of the CREATE FUNCTION statement. Every function created on the iSeries server must have a specific name. This name must be unique for the given library. This is an optional clause. If you do not specify a specific name for the function, the system will generate one. Normally the specific name is the same as the function's name. However, if a function with the specific name already exists, the system generates a unique name.

5 **EXTERNAL NAME** clause of the CREATE FUNCTION statement. Specifies the program, service program, or Java class that will be executed when the function is invoked in an SQL statement. The name must identify a program, service program, or Java class that exists at the server at the time the function is invoked. If the naming option is *SYS and the name is not qualified, the current path will be used to search for the program or service program at the time the function is invoked. The validity of the name is checked at the server. If the format of the name is not correct, an error is returned. If external-program-name is not specified, the external program name is assumed to be the same as the function name. The program, service program, or Java class need not exist at the time the function is created, but it must exist at the time the function is invoked.

**Note:** CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statements are not allowed in the external program of the function.

6 **PARAMETER STYLE** clause of the CREATE FUNCTION statement. DB2 Universal Database for iSeries passes additional parameters apart from the arguments defined in the CREATE FUNCTION statement based on the parameter style specified, as described in 15.3, “Parameter styles in external UDFs” on page 506.

Now let us examine the external program DEC2DATE referred to in the CREATE FUNCTION statement. We discuss the parameters that DB2 Universal Database for iSeries sends to the program and how the program makes use of these parameters. This program was written in RPG, as we show in Example 15-2. The DEC2DATE external program accepts an eight-digit decimal with a date that has to be converted to DATE format as the input argument.
Example 15-2  SQL parameter style -RPG program

```
0001.00  ****************************************************************
0002.00  H ALWNULL(*USRCTL)  1
0003.00  ****************************************************************
0004.00  d outdate      S    D datfmt(*ISO)  2
0005.00  d indate       S    8P 0  3
0006.00  d indatenul    S    2B 0
0007.00  d outdatenul   S    2B 0
0008.00  d sqlstate     S    5A  4
0009.00  d functname    S    517A VARYING
0010.00  d specname     S    128A VARYING
0011.00  d errormsg     S    70A VARYING
0012.00  *---------------------------------------------
0013.00  * PARAMETER DEFINITION
0014.00  *---------------------------------------------
0015.00  C *ENTRY        PLIST
0016.00  C                PARM                    indate
0017.00  C                PARM                    outdate
0018.00  C                PARM                    indatenul
0019.00  C                PARM                    outdatenul
0020.00  C                PARM                    sqlstate
0021.00  C                PARM                    functname
0022.00  C                PARM                    specname
0023.00  C                PARM                    errormsg  
0024.00  *---------------------------------------------
0025.00  C *iso          test(De)                indate
0026.00  C                 if                      %error
0027.00  C                  eval                  outdatenul = -1
0028.00  C                  else
0029.00  C                  move                  indate = outdate
0030.00  C                  eval                  outdatenul = 0
0031.00  C                  endif
0032.00  C                  eval                  *inlr = '1'
0033.00  *---------------------------------------------
```

Code sample notes

The function name DEC2DATE is the name of the source file member and also the name of the *PGM object, referred to in the CREATE FUNCTION statement as shown here:

```
EXTERNAL NAME 'DLEMA/DEC2DATE'
```

The following are some special comments regarding the source code in Example 15-2:

1. This program will not be called when null parameters are provided, but we allow nulls as input to add flexibility to the code.

2. The OUTDATE variable is defined as a date with ISO format.

3. The INDATE variable is defined as an 8-digit packed decimal.

4. From line 8 to 11, the parameters received by the RPG program are specified according to the parameter style used. In this case they are the SQLSTATE, function name, specific name and error message parameters. Some of those variables are very useful for error management.

5. This is something you do not want to have in your program. *INLR = ‘1’ causes the program to be released from memory once it finishes. But a UDF like this, typically is called many times, adversely affecting the performance.
A variation of this code is presented in Example 15-3. Notice the line in bold. Instead of disposing the program, it simply returns without forcing the program to release the memory. Although a rigorous test was not performed, we can say that the second DEC2DATE version outperforms the first one by a factor of three.

**Example 15-3 Variation on RPG ILE DEC2DATE UDF**

```
****************************************************************
H ALWNULL(*USRCTL)
****************************************************************
d outdate   S  D  datfmt(*ISO)
d indate    S  8P 0
d indatenul S  2B 0
d outdatenul S  2B 0
d sqlstate  S  5A
d functname S  517A VARYING
d specname  S  128A VARYING
d errormsg  S  70A VARYING
*---------------------------------------------
*PARAMETER DEFINITION
*---------------------------------------------
C  *ENTRY PLIST
C   PARM indate
C   PARM outdate
C   PARM indatenul
C   PARM outdatenul
C   PARM sqlstate
C   PARM functname
C   PARM specname
C   PARM errormsg
*---------------------------------------------
C  *iso  test(De)  indate
C   if %error
C   eval  outdatenul = -1
C   else
C   move  indate  outdate
C   eval  outdatenul = 0
C   endif
C  return
*---------------------------------------------
```

As mentioned earlier, these DEC2DATE programs are *PGM object. It is compiled into the *MODULE object, and then the *MODULE object is bound into a *PGM object, which allows us to specify the activation group parameter as *CALLER.

**Note:** The external program coded in any host language should be compiled with the Activation Group parameter as *CALLER.
The CRTBNDRPG CL command is used to compile and bind the DEC2DATE program as shown in Figure 15-14.

Create Bound RPG Program (CRTBNDRPG)

Type choices, press Enter.

Program ............... PGM > DEC2DATE
Library ............... > DLEMA
Source file ............ SRCFILE > RPGSRC
Library ............... > DLEMA
Source member ........ SRCMBR > DEC2DATE
Source stream file ... SRCSTMF

Generation severity level GENLVL 10
Text 'description' TEXT *SRCMBRTXT
Default activation group DFTACTGRP *YES

Figure 15-14 Create Bound RPG Program

When we deal with UDFs, triggers and stored procedures, it is more likely that we will use service programs instead of programs. Example 15-4 shows the same DEC2DATE code but with its *SRVPGM version.

Example 15-4 Second variation on RPG ILE DEC2DATE UDF as a *SRVPGM

****************************************************************
Hnomain
H ALWNULL(*USRCTL)
****************************************************************

Hnomain

H ALWNULL(*USRCTL)

ddec2date         PR
d indate                         8P 0
d outdate                         D
d indatenul                      2B 0
d outdatenul                     2B 0
d sqlstate                       5A

d functname                    517A   CONST OPTIONS(*VARSIZE) VARYING

d specname                     128A   CONST OPTIONS(*VARSIZE) VARYING

d errmsg                      70A   OPTIONS(*VARSIZE) VARYING

pdec2date         B                   export

*---------------------------------------------
*PARAMETER DEFINITION
*---------------------------------------------

C     *iso          test(De)                indate
C     if %error
C     eval outdatenul = -1
C     else
C     move indate outdate
A service source program may contain several different routines or entry points. This way, if we need different versions of our date casting function, one to convert from integer to DATE, another to convert from CHAR(8) to DATE, and so on, we can put them all together in the same source for making its management and maintenance easy. A service source program may contain routines for multiple purposes.

15.6.2 Coding the DB2SQL parameter style

This section describes how to code external UDFs with the DB2SQL parameter style.

Note: The DBINFO option works as in stored procedures, explained in 6.3.2, “Coding the DB2SQL parameter style” on page 137.

Suppose that we are interested in generating identifiers massively for a data warehousing data loading process. We can use the newly introduced GENERATED BY DEFAULT AS IDENTITY clause at table creation time, but our data modeler required that the generated identifiers do not repeat between tables. Our first approach was to consider a data area to keep track of the next number to be generated, but the application designer was interested in a code that may be ported to other platforms, and ask us not to use them. Then we thought of a table keeping track of the next consecutive number to be generated. To avoid this table becoming a bottleneck, instead of getting the next number in the sequence, we built an UDF that obtains the next group of numbers and uses the SCRATCHPAD to generate individual values until the end of the group is reached.

Let us examine the CREATE FUNCTION statement for the GENOID external UDF.

Example 15-5 CREATE FUNCTION statement for an external UDF using scratchpad

CREATE FUNCTION SAMPLEDB01.GENOID (CHAR(20))
  RETURNS DECIMAL(30, 0)
  LANGUAGE RPGLE
  SPECIFIC SAMPLEDB01.GENIOD00
  DETERMINISTIC
  MODIFIES SQL DATA
  RETURNS NULL ON NULL INPUT
  SCRATCHPAD 100
  EXTERNAL NAME 'SAMPLEDB01/GENOID(GENOID)'
  PARAMETER STYLE DB2SQL;

CREATE FUNCTION statement explanation

The elements that change from parameters style SQL to DB2SQL are explained in the following list:

- This argument is set by DB2 before calling the UDF. The scratchpad is a structure with an INTEGER that contains the length of the scratchpad and a space that is initialized to all binary zeroes by DB2 before the first call to the UDF. We defined here with a length area of
100 bytes. That means that the system will reserve 100 bytes of memory for the scratchpad area and send the address of this area to the function program.

This is the name of the external program that this function calls when it is invoked by the database. In this example, SAMPLEDB01 is the name of the library in which the program resides. GENOID is the name of the service program that is to be executed, and (GENOID) is the name of the RPGLE function inside the program that will be called when the function is invoked. The program does not need to exist at the time of the creation of the function, but it must be created before the function is invoked for the first time. This is an optional clause. If it is not specified, the system assumes that the name of the program to be executed is the same as the name of the function.

Now let us examine the service program GENOID referred to in the CREATE FUNCTION statement in Example 15-5. The program has two parts. The first one (Example 15-6) contains the definition of the RPG module.

Example 15-6  Part 1 - Module definition - Member GENOIDPR

```
0001.00 D GENOID          PR
0002.00 D Counter_name    20A
0003.00 D Output          30P 0
0004.00 D Ctr_name_ind   4B 0
0005.00 D Output_ind     4B 0
0006.00 D SQLstateRet    5A
0007.00 D FunctionName   517A   CONST OPTIONS(*VARSIZE) VARYING
0008.00 D SpecificName   128A   CONST OPTIONS(*VARSIZE) VARYING
0009.00 D DiagMsg        70A   OPTIONS(*VARSIZE) VARYING
0010.00 D ScratchPad     104A   OPTIONS(*VARSIZE) VARYING

The second part (Example 15-7) contains the module code.

Example 15-7  Part 2 - GENOID RPG module

```

0001.00 Hnomain
0002.00 H ALWNULL(*USRCTL)
0003.00 /COPY SAMPLEDB01/RPGSRC,GENOIDPR
0004.00 P GENOID          B          EXPORT
0005.00 D GENOID          PI
0006.00 D Counter_name    20A
0007.00 D Output          30P 0
0008.00 D Ctr_name_ind   4B 0
0009.00 D Output_ind     4B 0
0010.00 D SQLstateRet    5A
0011.00 D FunctionName   517A   CONST OPTIONS(*VARSIZE) VARYING
0012.00 D SpecificName   128A   CONST OPTIONS(*VARSIZE) VARYING
0013.00 D DiagMsg        70A   OPTIONS(*VARSIZE) VARYING
0014.00 D ScratchPad     104A   OPTIONS(*VARSIZE) VARYING
0015.00
0016.00 D ScratchPadDs   DS          BASED(ScratchPadPtr)
0017.00 D ScratchLenght  9B 0
0018.00 D FirstCall      4B 0
0019.00 D ChkValue       30P 0
0020.00 D Counter        30P 0
0021.00
0022.00 DWorkingStorage  DS
0023.00 D ScratchPadPtr  
0024.00 D CtrNamePtr     
0025.00
0026.00 C EVAL          ScratchPadPtr = %ADDR(ScratchPad)
0027.00 C IF        FirstCall  = 0
### Code sample notes

The following are some special comments of the source code:

1. **SQLState parameter used for returning the state of the function.** The SQLState is a 5-character string that should have a value in one of the following groups:
   - 00000: When the UDF completes without errors or warnings.
   - 01Hxx: Where xx can be any digits or uppercase letters: when the UDF completes without errors but with a warning.
   - 3Byxx: Where y is an uppercase letter between I and Z, and xx are any two digits or uppercase letters. When the UDF ends with an error condition.

2. **Fully qualified function name.** As the same program may be used in several UDFs, the name of the function, as registered in the DB2 Universal Database for iSeries catalogs, is received as a parameter and the programmers may use it in their programs.
Specific name for the function.

Diagnostic text field that may be passed back by the UDF with the warning or error message text.

Scratchpad: input/output parameter with the scratchpad.

The scratchpad received by the UDF has two values: A long integer containing the size of the scratchpad data and the data. This and the following four values redefine the scratchpad in order to be usable in the program.

A pointer to the scratchpad is set in order to make it usable in the program.

Since the scratchpad is initialized with binary zeroes the first time the UDF is called in an SQL statement scope, when the function is called the first time, FirstCall is zero.

If Counter reached the upper value of the group (and at the first call, both Counter and ChkValue are zero), a new upper value has to be calculated and recorded on the COUNTER table.

SQLCode -204 is fired if the COUNTER table does not exist. In that case, the table has to be created and the operation retried.

SQLCode 100 is fired if there is no row for the particular counter. A new row will be added to the COUNTER table.

As mentioned earlier, the GENOID program was created as a *SRVPGM object. In this case, GENOID is a program written in RPG.

The CRTSQLRPGI CL command is used to create the module of the program, if the program does not contain SQL embedded the CL command is CRTRPGMOD.

```plaintext
Create SQL ILE RPG Object (CRTSQLRPGI)

Type choices, press Enter.

Object .................. > GENOID          Name
Library ................... > DLEMA          Name, *CURLIB
Source file ............... > RPGSRC         Name, QRPGLSRC
Library ................... > DLEMA          Name, *LIBL, *CURLIB
Source member ............. > GENOID         Name, *OBJ
Commitment control ...... > *CHG            *CHG, *ALL, *CS, *NONE...
Relational database ...... > *LOCAL
Compile type .............. > *MODULE        *PGM, *SRVPGM, *MODULE
Listing output ......... > *NONE           *NONE, *PRINT
Text 'description' ...... > *SRCMBRTXT

Additional Parameters

Precompiler options .... *XREF          *XREF, *NOXREF, *GEN...
+ for more values

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

Figure 15-15  CRTSQLRPGI CL command

Then you must bind the GENOID service program; we use the following CL command:

CRTSRVPGM SRVPGM(DLEMA/GENOID) EXPORT(*ALL) TEXT('GENOID SVRPGM')
```
15.6.3 Coding the GENERAL parameter style

GENERAL parameter style is ideal for reusing existing code because it simply passes all parameters as input parameters and takes the result of the called program to be the result of the UDF. GENERAL parameter style in UDFs are only supported for external service programs.

In Example 15-8, a UDF is created based on an existing COBOL program that returns an exchange rate given an original currency, target currency at a specific date.

Example 15-8  CREATE FUNCTION statement for GENERAL parameter style

```sql
CREATE FUNCTION SAMPLEDB01.GET_FX_RATE (
    ORIG_CCY CHAR(3) ,
    TRGT_CCY CHAR(3) ,
    FX_DATE DATE )
RETURNS DECIMAL(10, 5)
LANGUAGE COBOL
SPECIFIC SAMPLEDB01.GET_FX_RATE01
DETERMINISTIC
READS SQL DATA
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'SAMPLEDB01/UDF_CBL(GET_FX_RATE)'
PARAMETER STYLE GENERAL ;
```

CREATE FUNCTION statement explanation

By this time, we are already familiar with SQL CREATE statements, but we must highlight the following points:

1. This function is deterministic because no matter when it is called, it always will return the same value for an equal set of parameters. By specifying the DETERMINISTIC clause, DB2 Universal Database for iSeries will avoid unnecessary executions for the program behind the UDF, improving performance.

2. Since in UDFs, GENERAL parameter style is only supported for service programs, the external name has to refer to one service program. In this case, the service program corresponds to the GET_FX_RATE entry point in the UDF_CBL service program object, located in the SAMPLEDB01 library.

Example 15-9  COBOL service program for GENERAL parameter style UDF

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. UDF_CBL3.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-AS400.
OBJECT-COMPUTER. IBM-AS400.

INPUT-OUTPUT SECTION.

DATA DIVISION.

WORKING-STORAGE SECTION.

*----------------------------------------------------------------------------------------------------------------*
* PARAMETERS NEEDED TO SIGNAL AN EXCEPTION *
*----------------------------------------------------------------------------------------------------------------*
```
01 SNDPGMMSG.
  03 SND-MSG-ID   PIC X(7) VALUE "UDF0005".
  03 SND-MSG-FILE PIC X(20) VALUE "CSTMSGF SAMPLEDB01".
  03 SND-MSG-DATA PIC X(30) VALUE "UDF ERROR ".
  03 SND-MSG-LEN PIC 9(8) BINARY VALUE 0.
  03 SND-MSG-TYPE PIC X(10) VALUE "**ESCAPE".
  03 SND-MSG-QUEUE PIC X(10) VALUE "**".
  03 SND-MSG-STACK PIC 9(8) BINARY VALUE 1.
  03 SND-MSG-KEY PIC X(4) VALUE " ".
  03 SND-ERROR-CODE.
    05 PROVIDED  PIC 9(8) BINARY VALUE 66.
    05 AVAILABLE PIC 9(8) BINARY VALUE 0.
    05 EXCEPTION-ID PIC X(7) VALUE " ".
    05 FILLER    PIC X(1) VALUE " ".
    05 EXCEPTION-DATA PIC X(50) VALUE " ".

*---------------------------------------------------------------*
*  IMPORT THE SQL COMMUNICATION AREA STRUCTURE                  *
*---------------------------------------------------------------*
EXEC SQL
  INCLUDE SQLCA
END-EXEC.

*---------------------------------------------------------------*
*  WORKING VARIABLES                                            *
*---------------------------------------------------------------*
77 EXCHANGE_RATE PIC S9(5)V9(5) PACKED-DECIMAL.

LINKAGE SECTION.

*---------------------------------------------------------------*
*  PARAMETERS                                                   *
*---------------------------------------------------------------*
77 ORIGINAL_CURRENCY PIC XXX.
77 TARGET_CURRENCY PIC XXX.
77 EXCHANGE_DATE FORMAT DATE.

PROCEDURE DIVISION USING
  BY REFERENCE ORIGINAL_CURRENCY
  BY REFERENCE TARGET_CURRENCY
  BY REFERENCE EXCHANGE_DATE
  RETURNING EXCHANGE_RATE.

A000-MAIN.
  EXEC SQL
    WHENEVER SQLERROR GO TO ED10-ERROR
END-EXEC.
  EXEC SQL
    WHENEVER NOT FOUND GO TO A200-ALTERNATE-SEARCH
END-EXEC.
  EXEC SQL
    SELECT FX_RATE INTO :EXCHANGE_RATE
    FROM SAMPLEDB01/CCY_FX_RATE
    WHERE ORIG_CCY = :ORIGINAL_CURRENCY
    AND   TRGT_CCY = :TARGET_CURRENCY
    AND   :EXCHANGE_DATE BETWEEN EFF_DT AND END_DT
END-EXEC.
A100-DONE1.
  GOBACK.
A200-ALTERNATE-SEARCH.
EXEC SQL
  WHENEVER NOT FOUND GO TO E020-EXCHANGE-RATE-NOT-FOUND
END-EXEC.
EXEC SQL
  SELECT DECIMAL((1.0/FX_RATE), 10,5) INTO :EXCHANGE_RATE
  FROM   SAMPLEDB01/CCY_FX_RATE
  WHERE  ORIG_CCY = :TARGET_CURRENCY
  AND    TRGT_CCY = :ORIGINAL_CURRENCY
  AND    :EXCHANGE_DATE BETWEEN EFF_DT AND END_DT
END-EXEC.
A200-DONE2.
GOBACK.
E020-EXCHANGE-RATE-NOT-FOUND.
MOVE "UDFO002" TO SND-MSG-ID.
E010-ERROR.
  CALL "QMHSNDPM" USING SND-MSG-ID,
          SND-MSG-FILE,
          SND-MSG-DATA,
          SND-MSG-LEN,
          SND-MSG-TYPE,
          SND-MSG-QUEUE,
          SND-MSG-STACK,
          SND-MSG-KEY,
          SND-ERROR-CODE.
GOBACK.

This code was compiled with the following command:
CRTSQLCLBI OBJ(SAMPLEDB01/UDF_CBL3) *SRCPGM(SAMPLEDB01/CBLSRC) *SRCMBR(UDF_CBL3)
*OBJTYPE(*MODULE) REPLACE(*NO)

The *SRCPGM was created with the following command:
CRTSRVPGM SRVPGM(SAMPLEDB01/UDF_CBL3) EXPORT(*ALL)

Notice that the parameters defined for the function are the parameters that the external program receives. Also notice that the program return will be the UDF return.

15.6.4 Coding the GENERAL WITH NULLS parameter style

It is a variation based on the GENERAL parameter style, closely related to the GENERAL parameter style explained in 15.6.3, “Coding the GENERAL parameter style” on page 521.

Example 15-10  CREATE FUNCTION statement for GENERAL WITH NULLS parameter style

CREATE FUNCTION SAMPLEDB01.GET_FX_RATE (  
  ORIG_CCY CHAR(3),
  TRGT_CCY CHAR(3),
  FX_DATE DATE ) 
  RETURNS DECIMAL(10, 5) 
  LANGUAGE COBOLLE 
  SPECIFIC SAMPLEDB01.GET_FX_RATE01 
  DETERMINISTIC  
  READS SQL DATA 
  RETURNS NULL ON NULL INPUT 
  EXTERNAL NAME 'SAMPLEDB01/UDF_CBL(GET_FX_RATE)' 
  PARAMETER STYLE GENERAL WITH NULLS;
For a UDF defined as in Example 15-10, the HLL program will have four input parameter and one output parameter, being the fourth a vector of null indicators corresponding to the null state for the first three parameters. The fifth parameter corresponds to the null state for the result the UDF will return.

15.6.5 Coding the DB2GENERAL parameter style

Let us examine the CREATE FUNCTION statement for the JVMPROPPERTIES external UDF.

Example 15-11  CREATE FUNCTION statement for DB2GENERAL parameter style

```sql
CREATE FUNCTION SAMPLEDB01.JVMPROP01 ( )
RETURNS TABLE ( 
  PROPERTY VARCHAR(500) , 
  VALUE VARCHAR(500) )
LANGUAGE JAVA
SPECIFIC SAMPLEDB01.JVMPROP01
NOT DETERMINISTIC
NO SQL
CALLED ON NULL INPUT
DISALLOW PARALLEL
EXTERNAL NAME 'JVMProp!dump'
PARAMETER STYLE DB2GENERAL ;
```

Note: The following notes refer to Example 15-11:
1. DB2GENERAL applies only for JAVA language.
2. JVMProp is the name of the Java class extending com.ibm.db2.app.UDF class, and dump is a public method implementing the UDF.
3. The parameter style clause is DB2GENERAL.

Now let us examine the external program JVMProp referred to in the CREATE FUNCTION statement. We discuss the parameters that DB2 Universal Database for iSeries sends to the program and how the program makes use of these parameters. This program was written in JAVA, as shown in Example 15-12.

Example 15-12  Source code JVMProp.java DB2GENERAL parameter style

```java
import COM.ibm.db2.app. *
import java.util.*

public class MyUDTFs extends UDF {
    Enumeration propertyNames;
    Properties properties;

    public void ranking (String property, String value) throws Exception {
        int callType = getCallType();
        switch (callType) {
            case SQLUDF_TF_FIRST:
                break;
            case SQLUDF_TF_OPEN:
                properties = System.getProperties();
                propertyNames = properties.propertyNames();
                break;
            case SQLUDF_TF_FETCH:
                if (propertyNames.hasMoreElements()) {
                    property = (String) propertyNames.nextElement();
                    break;
                }
        }
    }
}
```
value = properties.getProperty(property);
set(1, property);
set(2, value);
} else {
    setSQLstate("02000");
}
break;
case SQLUDF_TF_CLOSE:
    break;
case SQLUDF_TF_FINAL:
    break;
default:
    throw new Exception("UNEXPECTED call type of " + callType);
}
the number of working days between two given dates for a given calendar, provided that there are calendars with different holiday dates.

Example 15-13 CREATE FUNCTION for the Java parameter style

```sql
CREATE FUNCTION SAMPLEDB01.WORKING_DAYS (  
    INITIALDATE DATE ,  
    FINALDATE Date ,  
    CALENDARID INTEGER  
)  
RETURNS BIGINT  
LANGUAGE JAVA  
SPECIFIC SAMPLEDB01.WDAYS01  
DETERMINISTIC  
READS SQL DATA  
RETURNS NULL ON NULL INPUT  
NO EXTERNAL ACTION  
NOT FENCED  
EXTERNAL NAME 'WrkDays!wrkDays'  
PARAMETER STYLE JAVA ;
```

Note: The following notes refer to Example 15-13:

1. The Java parameter style is allowed only when the language is Java.
2. Since this function has an SQL SELECT operation inside, it is very important to avoid its execution when the same couple of dates have been provided as parameters. Because of that, it has been defined as DETERMINISTIC.
3. In this example, WrkDays is the name of the class and wrkDays is the public static method implementing this particular UDF.

Now let us examine the external program WrkDays referred to in the CREATE FUNCTION statement. We discuss the parameters that DB2 Universal Database for iSeries sends to the program and how the program makes use of these parameters. This program was written in Java, as shown in Example 15-14.

Example 15-14 Source code WrkDays.java in the Java parameter style

```java
import java.sql.*;

public class WrkDays {

    public static long wrkDays(Date initialDate, Date finalDate, int calendarKey) 
    throws SQLException {
        int count;
        long result;
        Connection con = DriverManager.getConnection("jdbc:default:connection");
        String stmt = "SELECT COUNT(*) FROM SAMPLEDB01.HOLIDAY " + 
                        "WHERE HOLIDAY_DATE BETWEEN ? AND ? AND CALENDAR_KEY = ?";
        PreparedStatement ps = con.prepareStatement(stmt);
        ps.setDate(1, initialDate);
        ps.setDate(2, finalDate);
        ps.setInt(3, calendarKey);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            count = rs.getInt(1);
            if (initialDate.compareTo(finalDate) > 0) count = -count;
            result = ((long)(finalDate.getTime()-initialDate.getTime()))/ 
                     ((long)86400000) // 8640000 = 1000*24*60*60
                - count;
        }
    }

    public static void main(String[] args) {
        System.out.println(wrkDays(new Date(2002, 12, 25), new Date(2002, 12, 29), 1));
    }
}
```
else  
    throw new SQLException("Error reading holiday table", "38ZZZ");
if (rs != null) rs.close();
if (ps != null) ps.close();
if (con != null) con.close();
return result;
}

Code sample notes
The following are some special comments about the source code in Example 15-14:

1. For the Java parameter style, a public Java class should be defined. That class may contain one or more methods, one for each UDF. UDFs and stored procedures may be combined in the same class. The method implementing the UDF has to be public static.

2. If the Java method implementing the UDF is going to perform database operations, it has to connect to the database manager using jdbc:default:connection URL.

3. The difference between two dates is calculated in milliseconds and then converted to days. The number of holidays is then subtracted.

4. In case any SQL error occurs, an SQLException is thrown by the method to the caller.

5. The value returned by the method should be compatible with SQL types.

15.7 Error handling in external UDFs

UDFs may report errors and warnings basically by the same mechanisms described in Chapter 8, “Stored procedure error handling” on page 221. These depend on the parameter passing convention being used.

When DB2SQL parameter passing is being used, the UDF program reports errors and warnings using the SQLUDF_MSGTX and SQLUDF_STATE parameters. When DB2GENERAL or Java parameter passing is being used, the UDF program reports errors by throwing SQLException or SQLWarning exceptions. If a warning message is sent back to DB2, the SQL statement will continue running. When an error message is sent back to DB2, the SQL statement will be stopped. Reported SQL errors and warnings should follow the same conventions as in stored procedures.

15.7.1 Error handling with the DB2SQL parameter style

The following UDTF receives a file name corresponding to a stream file in the IFS and returns it as a table. Some errors may occur during the execution:

- File may not exist
- I/O error may occur during processing the file

Also, some warnings may occur during the execution of the program:

- Rows in the file may have an inappropriate format.
- End of file is reached.
Example 15-15  CREATE FUNCTION statement for F1Result UDTF

CREATE FUNCTION SAMPLEDB01.F1RESULTS(
    FILENAME VARCHAR(255))
RETURNS TABLE (
    DRIVER_NUMBER SMALLINT,
    DRIVER_NAME,
    GENERAL_POSITION SMALLINT,
    LAST_RACE_POSITION,
    CONSTRUCTOR VARCHAR(50),
    WHEEL_BRAND VARCHAR(50),
    DRIVER_COUNTRY VARCHAR(50),
    YTD_POOLS SMALLINT,
    YTD_WINS SMALLINT,
    YTD_POINTS SMALLINT)
LANGUAGE C
SPECIFIC SAMPLEDB01.F1RESULTS
NOT DETERMINISTIC
NO SQL
RETURNS NULL ON NULL INPUT
DISALLOW PARALLEL
NOT FENCED
CARDINALITY 22
EXTERNAL NAME 'SAMPLEDB01/CUDF(readFileToTable)
PARAMETER STYLE DB2SQL;

Notice that UDTFs require DISALLOW PARALLEL to be specified in 1. We specified
CARDINALITY to 22 to help the optimizer provide an estimate of the average number of rows
returned by the UDTF.

The code has some features to highlight, such as error handling and pointer arithmetic.
Pointer arithmetic was used to align the pointer to the file structure and will be explained in
15.8, "Pointer arithmetic and the scratchpad" on page 534.

Example 15-16  HLL UDTF showing error handling, scratchpad and pointer adjustment

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqludf.h>
#include <sqlstate.h>
define MAX_LEN 255

/* function that receives a text, sets the value to the reference parameter */
/* value and returns -1 (SQLNull) if text is null or 0 (not null) if text */
/* is not null */
short scanShort(short *value, char* text) {
    if (text == NULL) return (short)-1;
    *value = (short)atoi(text);
    return (short)0;
}

/* function that receives a text, sets the value to the reference parameter */
/* value and returns -1 (SQLNull) if text is null or 0 (not null) if text */
/* is not null */
short scanText(char *value, char* text) {
    if (text == NULL) return (short)-1;
    strcpy(value, text);
return (short)0;
}

/* User defined table function that receives a file name as parameter and */
/* reads it from the IFS and returns it as a table of Formula 1 pilots */
void SQL_API_FN readFileToTable(
    /* UDTF parameter */
    SQLUDF_VARCHAR *fileName,         /* input */
    /* Columns in the returned table */
    SQLUDF_SMALLINT *driverNumber,     /* output */
    SQLUDF_VARCHAR  *driverName,       /* output */
    SQLUDF_SMALLINT *generalPosition,  /* output */
    SQLUDF_SMALLINT *lastRacePosition, /* output */
    SQLUDF_VARCHAR  *constructor,      /* output */
    SQLUDF_VARCHAR  *wheelBrand,       /* output */
    SQLUDF_VARCHAR  *driverCountry,    /* output */
    SQLUDF_SMALLINT *ytdPools,         /* output */
    SQLUDF_SMALLINT *ytdWins,          /* output */
    SQLUDF_SMALLINT *ytdPoints,        /* output */
    /* null indicator for the received parameter */
    SQLUDF_NULLIND *fileNameInd,
    /* null indicators for table columns */
    SQLUDF_NULLIND *driverNumberInd,
    SQLUDF_NULLIND *driverNameInd,
    SQLUDF_NULLIND *generalPositionInd,
    SQLUDF_NULLIND *lastRacePositionInd,
    SQLUDF_NULLIND *constructorInd,
    SQLUDF_NULLIND *wheelBrandInd,
    SQLUDF_NULLIND *driverCountryInd,
    SQLUDF_NULLIND *ytdPoolsInd,
    SQLUDF_NULLIND *ytdWinsInd,
    SQLUDF_NULLIND *ytdPointsInd,
    /* rest of the arguments */
    SQLUDF_TRAIL_ARGS_ALL)
{
    char line[MAX_LEN], *result;
    char *token;

    /* scratchpad structure */
    typedef struct {  
        FILE *f;
        int rowNumber;
    } strScratchPad;
    strScratchPad *strSPad;
    strScratchPad **ptrAlignmentPointer;

    /* get the address of the scratchpad buffer passed by the DB2 UDB */
    /* and align the pointer for the internal scratchpad structure at */
    /* the 16 byte boundary */
    ptrAlignmentPointer = ((strScratchPad**)(sqludf_scratchpad))+1;
    strSPad = (strScratchPad*)ptrAlignmentPointer;

    switch (SQLUDF_CALLT) {
        case SQLUDF_TF_OPEN:
            /* open a text stream file and store the pointer */

Chapter 15. External user-defined functions 529
/* on the scratch pad. Check to see if it opened */
/* successfully */
if ((strSPad->f = fopen(fileName, "r")) == NULL) {
    strcpy(SQLUDF_MSGTX, "Could not open file ");
    strncat(SQLUDF_MSGTX, fileName,
        SQLUDF_MSGTEXT_LEN -
        strlen(SQLUDF_MSGTX) -1);
    strncpy(SQLUDF_STATE, "38200",
        SQLUDF_SQLSTATE_LEN);
    return;
}
strSPad->rowNumber = 0;
break;
case SQLUDF_TF_FETCH:
    /* count the row */
    strSPad->rowNumber++;
    /* read a new line from the stream file */
    if ((result = fgets(line, MAX_LEN, strSPad->f))==NULL) {
        /* end of file reached */
        strncpy(SQLUDF_STATE, SQL_NODATA_EXCEPTION,
            SQLUDF_SQLSTATE_LEN);
        return;
    }
    /* scans for the values in the line */
    *driverNumberInd = scanShort(driverNumber, 
        strtok(result, ","));
    *driverNameInd = scanText(driverName, 
        strtok(NULL, ","));
    *generalPositionInd =scanShort(generalPosition, 
        strtok(NULL, ","));
    *lastRacePositionInd = scanShort(lastRacePosition, 
        strtok(NULL, ","));
    *constructorInd = scanText(constructor, 
        strtok(NULL, ","));
    *wheelBrandInd = scanText(wheelBrand, 
        strtok(NULL, ","));
    *driverCountryInd = scanText(driverCountry, 
        strtok(NULL, ","));
    *ytdPoolsInd = scanShort(ytdPools, 
        strtok(NULL, ","));
    *ytdWinsInd = scanShort(ytdWins, 
        strtok(NULL, ","));
    *ytdPointsInd = scanShort(ytdPoints, 
        strtok(NULL, ","));
    break;
    case SQLUDF_TF_CLOSE:
        /* close the file */
        fclose(strSPad->f);
        strSPad->f = NULL;
        strSPad->rowNumber = 0;
        break;
}
In 3, the routine opens the file for which the name was received as a parameter. If for any reason the file cannot be opened, an error message with SQLSTATE of 38200 is returned. Error message text and error state 38200 are established in lines 4 and 5, respectively.

When the end of the file containing the Formula 1 scores is reached, line 6 signals a warning message indicating to DB2 Universal Database for iSeries that the end of the returning table has been reached. In line 7, the file is closed.

**Note:** To obtain access to files in the IFS, the CRTCMD command was issued with the SYSIFCOPT option set to *IFSIO, as follows:

```
CRTCMD MODULE(DLEMA/F1UDTF) SRCFILE(DLEMA/CSRC) OUTPUT(*print) DBGVIEW(*SOURCE) SYSIFCOPT(*IFSIO)
```

### 15.7.2 Error handling with the DB2GENERAL parameter style

Java UDFs and UDTFs that use DB2GENERAL parameter style inherit methods for reporting exceptions and warnings:

- `setSQLstate`
- `setSQLmessage`

Use care with the state and size of the message returned. The state code should follow the rules explained in 8.1, “Database error reporting strategy” on page 222. Message text should not exceed 70 characters.

Example 15-17 presents a Java version of Example 15-16 on page 528.

**Example 15-17  Java version of the F1Result UDTF**

```java
public class F1RESULTS extends UDF {

    private BufferedReader f1BufferedReader;
    private FileReader f1FileReader;
    private int lines = 0;
    private static int MAXLINES = 100;

    private void openFile(String fileName) throws IOException {
        f1FileReader = new FileReader(fileName);
        f1BufferedReader = new BufferedReader(f1FileReader);
    }

    private void closeFile() throws IOException {
        if (f1BufferedReader != null) f1BufferedReader.close();
        if (f1FileReader != null) f1FileReader.close();
    }

    private void parseLine(String line) throws Exception {
        int i = 2, index = 0;
        for (; i <= 10; i++) {
            int newIndex = line.indexOf(',', index);
            if (index <= line.length() && newIndex != -1 &&
                newIndex <= line.length()) {
                if (i == 2 || i == 4 || i == 5 || i == 9 || i == 10) {
                    try {
                        set(i, Short.parseShort((
                            line.substring(index, newIndex).trim())));
                    }
                    catch (NumberFormatException e) {
                        throw new Exception("Conversion Error");
                    }
                }
            }
        }
    }
}
```
```java
} catch (NumberFormatException nfe) {
    setSQLmessage("nfe in line " + lines);
    setSQLstate("01H04");
}
} else
    set(i,line.substring(index, newIndex).trim());
index = newIndex + 1;
if (index < line.length()) {
    try {
        set(11, Short.parseShort(
                line.substring(index).trim()));
    } catch (NumberFormatException nfe) {
        setSQLmessage("nfe in line " + lines);
        setSQLstate("01H04");
    }
}
}

public void f1results (
    String fileName,
    short driverNumber,
    String driverName,
    short generalPosition,
    short lastRacePosition,
    String constructor,
    String wheelBrand,
    String driversCountry,
    short ytdPools,
    short ytdWins,
    short ytdPoints) throws Exception {

    int callType;
    callType = getCallType();
    try {
        switch (callType) {
            case SQLUDF_TF_OPEN: openFile(fileName); break;
            case SQLUDF_TF_FETCH:
                if (lines++ >= MAXLINES) {
                    setSQLstate("02000");
                    return;
                }
            case SQLUDF_TF_CLOSE:
                closeFile(); break;
            default:
                setSQLstate("38H06");
                setSQLmessage("Improper call type");
                break;
        }

        String line = f1BufferedReader.readLine();
        if (line == null) {
            setSQLstate("02000");
            return;
        }
        parseLine(line);
        break;
        case SQLUDF_TF_CLOSE:
            closeFile();
            break;
        default:
            setSQLstate("38H06");
            setSQLmessage("Improper call type");
            break;
        }
    }
```
Code sample notes

The following are some special comments of the source code in Example 15-17:

1. Java UDTFs are only supported in DB2GENERAL parameter style. DB2GENERAL parameter style Java UDFs must extend UDF class.
2. When a number format exception is detected, a message indicating the row in which the problem was found is established.
3. When a number format exception is detected, the UDTF returns an SQL Warning 01H04 and continues.
4. When the end of file is reached, the UDTF sets a SQL state of 02000 indicating that there are no more rows.
5. When a call type other than open, fetch, or close will fire an SQL error with state 38H06 and message will be "improper call type".
6. I/O errors will cause an SQL error with state 38H01. Notice that the program controls the maximum length of the error message.

A simple client program using this UDTF is illustrated in Example 15-18.

Example 15-18  Client program using F1RESULTS UDTF

```java
public class F1Client {

    private static String STMT = "SELECT * FROM TABLE(SAMPLEDB01.F1RESULTS(\";
    private static String STMT2 = \\"); AS A";

    private static void printMessage(String state, String message) {
        System.err.println("SQL State: " + state);
        System.err.println("SQL Message: " + message);
    }

    public static void main(String[] args) {
        try {
            // first argument gives the class name for the JDBC driver
            Class.forName(args[0]).newInstance();
        }
        catch (Exception e) {
            System.err.println("Error: the JDBC driver is not valid");
            System.exit(1);
        }

        try {
            Connection con = DriverManager.getConnection(
                // subsequent arguments supply the URL, user and password
                args[1].trim(), args[2].trim(), args[3].trim());

            // fourth argument provides the file name to be passed to the UDTF
            PreparedStatement ps = con.prepareStatement(STMT + args[4] + STMT2);
        }
    }
}
```
ResultSet rs = ps.executeQuery();

while (rs.next()) {
    SQLWarning sqlw = rs.getWarnings();
    if (sqlw != null)
        printMessage(sqlw.getSQLState(), sqlw.getMessage());
    // now print the row columns... omitted for simplicity
}

} catch (SQLException sqle) {
    printMessage(sqle.getSQLState(), sqle.getMessage());
    System.exit(2);
}


Code sample notes
The following are some special comments of the source code in Example 15-18:

1. UDTFs may be used from any kind of program, regardless of the original programming language used for it. This sample code is a Java program that receives the driver name, database URL, user, password, and file name as parameters; and assemble the SELECT statement, including an UDTF that reads the forecasted Formula 1 results for next year and prints them. Actually, the code does not print it, for simplicity.

2. After fetching each row, the program test for warning messages. If any warning message is received, it is printed.

3. If an SQL error message is received, it is printed.

15.8 Pointer arithmetic and the scratchpad

Each hardware platform used to have its own requirements regarding alignment of certain variables. When you code a UDF with scratchpad, those alignment rules must be followed. In iSeries servers, pointers have to be aligned in 16-byte boundaries. The following snippet gives the definition of the scratchpad structure as it is defined in the include file sqludf.h:

```c
SQL_STRUCTURE sqludf_scratchpad {
    unsigned long length;       /* length of scratchpad data */
    char          data[SQLUDF_SCRATCHPAD_LEN]; /* scratchpad data, init. to all \0 */
};
```

In Example 15-16 on page 528, we have to store a file pointer in the scratchpad, but the data part of it may not be aligned to a 16 boundary, so we can code something like the following snippet to align that pointer to a 16 boundary:

```c
typedef struct {
    char filler[8];
    FILE *f;
    integer rowNumber;
} strScratchpad;
...
strScratchpad *ScratchPadData;
...
ScratchPadData = *sqludf_scratchpad->data;
...
However, this code is platform dependent and if we plan to move the code to another platform or even if the platform architecture changes, the program will not work anymore. It may be possible that in the future, when 64 bit architecture becomes obsolete and the iSeries evolves into a 128-bit machine, the boundary for pointers may change. A more platform-independent code is required, as that shown in the following snippet:

```c
typedef struct {
    FILE *f;
    integer rowNumber;
} strScratchpad;
...
strScratchPad *strSPad;
strScratchPad **ptrAlignmentPointer;
...
ptrAlignmentPointer = ((strScratchPad**)(sqludf_scratchpad))+1;  
strSPad = (strScratchPad*)ptrAlignmentPointer;
```

In the previous code snippet, you see that a structure, called strScratchPad, has been declared. The variable f is a pointer to the input stream file to be read, and rowNumber is an integer to keep track of the line being read, to report warning or error messages referring to the failing row. We declare a pointer to this structure called strSPad. The scratchpad that is passed to the program itself is a structure of two elements.

In this program, the data element of the scratchpad structure is cast to the strScratchPad structure. We use the data element of the sqludf_scratchpad structure as a memory buffer for our internal strScratchPad structure. The method of casting, such as the previous one, is used to align the strSPad pointer on a 16-byte boundary (or whatever the boundary for the specific platform is). If your code fails to align addresses properly, an exception is thrown at the runtime, and the application is terminated.

### 15.8.1 Debugging external UDFs

The same technique of the debugging that we defined in 14.8, “Debugging UDFs” on page 488 applies here.

### 15.9 Coding example for an external user-defined table function

In this section, we provide sample coding for an external UDTF. Since SQL does not have a function that is equivalent to DIR command in Windows or LS in UNIX, we write a program that does this work and create an external UDTF for this program. This external UDTF receives one parameter of the OS/400 integrated file system directory path name (less than 128 characters in length) and returns a list all the subdirectories of the specified path.

We give this UDTF a name of IFSDIR and stores it in a schema named SQLLIB. The external program is coded in RPG IV. Example 15-19 shows the code of the RPG IV program:
Example 15-19  RPG IV codes of IFSDIR program

```sql
*=================================================================
*  FolderList UDTF
*  CREATE FUNCTION XXX/IFSDIR (IFSFOLDER VARCHAR(128))
*  RETURNS TABLE (FileName    VARCHAR(128),
*                 CreateStamp TIMESTAMP,
*                 AccessStamp TIMESTAMP,
*                 ModStamp    TIMESTAMP,
*                 Type        CHAR(10),
*                 DataSize    BIGINT,
*                 AllocSize   BIGINT,
*                 Owner       CHAR(10))
*  EXTERNAL
*  LANGUAGE RPGLE
*  DISALLOW PARALLEL
*  RETURNS NULL ON NULL INPUT
*  PARAMETER STYLE DB2SQL
*
*  Compile Instructions:
*  CRTBNDRPG PGM(OBJLIB/IFSDIR) SRCFILE(SRCLIB/QRPGLESRC)
*=================================================================
H DftActGrp(*No) ActGrp(*Caller) BndDir(‘QC2LE’)
*
*  Table Function Inputs
D Folder          S            128    Varying
*
*  Table Function Columns
D FileName        S            128    Varying
D CreateStamp     S               Z
D AccessStamp     S               Z
D ModStamp        S               Z
D Type            S             10
D DataSize        S             20I 0
D AllocSize       S             20I 0
D Owner           S             10
*
*  NULL Indicator Variables
D Folder_NI       S              5I 0
D FileName_NI     S              5I 0
D CreateStamp_NI  S              5I 0
D AccessStamp_NI  S              5I 0
D ModStamp_NI     S              5I 0
D Type_NI         S              5I 0
D DataSize_NI     S              5I 0
D AllocSize_NI    S              5I 0
D Owner_NI        S              5I 0
*
*  UDTF Call Type Parm
D CallType        s             10i 0
*
*  UDTF call parameter constants
D UDTF_FirstCall  s             10i 0 Inz(-2)
D UDTF_Open       s             10i 0 Inz(-1)
D UDTF_Fetch      s             10i 0 Inz(0)
D UDTF_Close      s             10i 0 Inz(1)
D UDTF_LastCall   s             10i 0 Inz(2)
*
*  SQL States
D SQLSTATEOK      c                   '00000'
```
**Chapter 15. External user-defined functions**

- **NULL Constants**
  - `ISNULL` c -1
  - `NOTNULL` c 0

- **Prototypes for all procedures**
  - `GetAttr` pr n
    - `QPathName` 128 Const
    - `CountryID` 2 Const
    - `LangID` 3 Const

- **Get IFS Object Attributes**
  - `QpOlGetAttr` pr 10i 0 ExtProc('QpOlGetAttr')
    - `pPath` * Value
    - `pAttrArr` * Value
    - `pBuffer` * Value
    - `pBufferSize` 10u 0 Value
    - `pBufferSizeN` * Value
    - `pNoBytes` * Value
    - `pSymLinkUse` 10u 0 Value

- **Directory Prototype Entries**

- **Open an IFS Directory**
  - `OpenDir` PR * EXTPROC('opendir')
    - `dirname` * Value

- **Read IFS Directory Contents**
  - `ReadDir` PR * EXTPROC('readdir')
    - `dirhndl` * Value

- **Close IFS Directory**
  - `CloseDir` PR 10i 0 EXTPROC('closedir')
    - `dirhndl` * Value

- **IFS Entry Handles**
  - `dirName` s 128
  - `dirHandle` s *
  - `DirPtr` s *
  - `dsDirEntry` ds Based(DirPtr)
    - `MiscInfo` 40
    - `DirEntryCCSID` 10i 0
    - `DirEntryCntry` 2
    - `DirEntryLang` 3
    - `DirEntryNLSR` 3
    - `DirEntryL` 10u 0
    - `DirEntry` 640

- **C API Error Retrieval**

- `dGetErrNo` pr * Extproc('_errno')
- `dStrError` pr * Extproc('strerror')
  - `dErrNo` s 10i 0 Based(pErrNo)
  - `dErrStr` s 128 Based(ptrStrError)
  - `dptrStrError` s *
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dErrDesc</td>
<td>s</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>dStatus</td>
<td>s</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>dNULL</td>
<td>s</td>
<td>1</td>
<td>Inz(X'00')</td>
</tr>
</tbody>
</table>

* Parameters for DB2SQL parameter style

* SQL State - Input/Output

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_State</td>
<td>s</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

* Function Name - Schema.Def name - Input only

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function_Name</td>
<td>s</td>
<td>517</td>
<td></td>
</tr>
</tbody>
</table>

* Function Specific Name - Input Only

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific_Name</td>
<td>s</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

* Message Text - Input/Output

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msg_Text</td>
<td>s</td>
<td>70</td>
<td>Varying</td>
</tr>
</tbody>
</table>

* Error Code Data Structure for API Calls

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DdsErrCode</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BytesPrv</td>
<td>10i</td>
<td>0</td>
<td>Inz(%Size(dsErrCode))</td>
</tr>
<tr>
<td>BytesAvl</td>
<td>10i</td>
<td>0</td>
<td>Inz(%Size(dsErrCode))</td>
</tr>
<tr>
<td>ExcID</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ErrData</td>
<td>128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Entry PLlist

* Function input parameters

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Function output parameters

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>FileName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>CreateStamp</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>AccessStamp</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>ModStamp</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>DataSize</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>AllocSize</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>Owner</td>
<td></td>
</tr>
</tbody>
</table>

* Function NULL input parameter indicators

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>Folder_NI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Function NULL output parameter indicators

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>FileName_NI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>CreateStamp_NI</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>AccessStamp_NI</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>ModStamp_NI</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>Type_NI</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>DataSize_NI</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>AllocSize_NI</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>Owner_NI</td>
<td></td>
</tr>
</tbody>
</table>

* DB2SQL Style Parms

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>SQL_State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>Function_Name</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>Specific_Name</td>
<td></td>
</tr>
<tr>
<td>Parm</td>
<td></td>
<td>Msg_Text</td>
<td></td>
</tr>
</tbody>
</table>

* UDTF CallType flag parm (Open,Fetch,Close)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>CallType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
* Process Table Request

C/Free

SQL_State=SQLSTATEOK;
pErrNo=GetErrNo;
ErrNo=*Zero;

Select;

//
// Step 1. When CALL TYPE is UDTF_OPEN, perform
// initialization work. Data is not
// returned at this time. If an error
// occurs here, resource cleanup has
// to be done because the UDTF will
// not be called again with a UDTF_CLOSE.
//
When CallType=UDTF_Open;

dirName=%TrimR(Folder)+NULL;
dirHandle=OpenDir(%Addr(DirName));
If dirHandle=*NULL;
pErrNo=GetErrNo;
Status=*Blanks;
ptrStrError=StrError(ErrNo);
If (ptrStrError<>*Null);
    Status=%Subst(ErrStr:1:
        %Scan(NULL:ErrStr)-1);
    SQL_State=UDTF_ERROR;
    Msg_Text=%Char(ErrNo)+' '+Status;
    *InLR=*On;
Endif;
EndIf;

//
// Step 2. When CALL TYPE is UDTF_FETCH, retrieve
// data on a row by row basis. The output
// column parameters should be filled in
// as well as the appropriate NULL indicator
// variables set. This call will in effect
// return one row to the "virtual" table.
//
// When the table has no more data to fetch,
// the SQL_State should be set to '02000'
// (No Data). Neglecting to return this
// end of table marker will result in an
// infinite loop.
//
When CallType=UDTF_Fetch;

// Read directory entry
DirPtr=ReadDir(dirHandle);

If DirPtr=*Null;
pErrNo=GetErrNo;
If ErrNo>*Zero;
    Status=*Blanks;
    ptrStrError=StrError(ErrNo);
    If (ptrStrError<>*Null);
        Status=%Subst(ErrStr:1:
            %Scan(NULL:ErrStr)-1);
        SQL_State=UDTF_ERROR;
        Msg_Text=%Char(ErrNo)+' '+Status;
        *InLR=*On;
    Endif;
Endif;
Else;

//End of Table Condition
SQL_State=ENDOFTABLE;
EndIf;
Else;
  // Get attributes for directory entry
  If GetAttr(Folder+'/%Subst(dirEntry:1:DirEntryL):
    DirEntryCntry:DirEntryLang)=*On;
    // Set Indicator Flags to NULL
    CreateStamp_NI=ISNULL;
    AccessStamp_NI=ISNULL;
    ModStamp_NI=ISNULL;
    Type_NI=ISNULL;
    DataSize_NI=ISNULL;
    AllocSize_NI=ISNULL;
    Owner_NI=ISNULL;
  Else;
    // Set Indicator Flags to NOT NULL
    CreateStamp_NI=NOTNULL;
    AccessStamp_NI=NOTNULL;
    ModStamp_NI=NOTNULL;
    Type_NI=NOTNULL;
    DataSize_NI=NOTNULL;
    AllocSize_NI=NOTNULL;
    Owner_NI=NOTNULL;
  EndIf;
  FileName_NI=NOTNULL;
EndIf;
//
// Step 3. When CALL TYPE is UDTF_CLOSE, perform
// cleanup work. This value is passed when
// the FETCH routine signals an error or
// signals an end of table condition.
//
When CallType=UDTF_Close;
  CloseDir(dirHandle);
  *InLR=*On;
EndS1;
Return;
BegSr *PSSR;
  SQL_State=UDTF_ERROR;
  Msg_Text='General Program Error';
  *InLR=*On;
EndSr;
/End-Free
*=================================================================
* GetAttr (call the Qp01GetAttr API)
* Get Attribute for Stream File or Object
*
* Returns:
*   Error flag (Indicator - *on/*Off). If *on then the
*   attributes were not retrieved.
*
* Parameters:
*   QPathName    INPUT    Fully qualified path name of object
*                  to retrieve attributes for.
*=================================================================
P GetAttr           b
D GetAttr pi n
D QPathName 128 Const
D CountryID 2 Const
D LangID 3 Const
*
D dsPath ds
D pCCSID 10i 0 Inz(0)
D pCountryID 2
D pLangID 3
D pResrv 3 Inz(x'000000')
D pPathType 10i 0 Inz(0)
D pPathNameLen 10i 0 Inz(%Size(pPathName))
D pPathNameDlm 2 Inz('/')
D pResrv2 10 Inz(x'00000000000000000000')
D pPathName Like(qPathName)
*
* IFS Object attribute list to retrieve
D dsAttrArray ds
D NoAttribs 10i 0 Inz(7)
D ReqAttrID1 10i 0 Inz(cATTR_OBJType)
D ReqAttrID2 10i 0 Inz(cATTR_Auth)
D ReqAttrID3 10i 0 Inz(cATTR_AccTime)
D ReqAttrID4 10i 0 Inz(cATTR_ModTime)
D ReqAttrID5 10i 0 Inz(cATTR_CrtTime)
D ReqAttrID6 10i 0 Inz(cATTR_DataSize)
D ReqAttrID7 10i 0 Inz(cATTR_AllcSize)
*
D BufferSizeReq s 10u 0
D BytesReturned s 10u 0
D ReturnCode s 10i 0
*
* Return Buffer to Receive File System Attributes
D dsAttrRetBuf ds based(ptrRetBuffer)
D AttrOffset 10i 0
D AttrID 10i 0
D AttrSize 10i 0
D AttrReserved 10i 0
D AttrData 1024
D AttrDataN 10u 0 Overlay(AttrData)
*
D ptrRetBuffer s * Inz(%Addr(RetBuffer))
D RetBuffer s 2048
D i s 3 0
*
D zEpochTime s z Inz(z'1970-01-01-00.00.00.000000')
D zResultDate s z
*
* Constants for GET FILE ATTRIBUTES Qp01GetAttr
D cATTR_OBJType c const(0)
D cATTR_DataSize c const(1)
D cATTR_AllcSize c const(2)
D cATTR_ExtAtrSz c const(3)
* NOTE: Times are returned in seconds from Epoch Date
D cATTR_CrtTime c const(4)
D cATTR_AccTime c const(5)
D cATTR_ChgTime c const(6)
D cATTR_ModTime c const(7)
*
D cATTR_StgFree c const(8)
D cATTR_CheckOut c const(9)
D cATTR_LclRemot c const(10)
D cATTR_Auth c const(11)
D cATTR_FileID c const(12)
D cATTR_ASP c const(13)
*  
  *  Object Authority Structure
D dsAuthority ds based(ptrAuthority)
D Auth_owner 10
D Auth_prigroup 10
D Auth_Autl 10
D Auth_rsv1 10
D Auth_Offset 10i 0
D Auth_NoUsers 10i 0
D Auth_EntSize 10i 0
*      
D ptrAuthority s *
*  
  *  This structure is used for authorization entries for a given object. Note that the single char flags (us_read, etc.) will contain a hex 00 or 01, not an EBCDIC 0 or 1.
*      
D dsUserStruc ds based(ptrUserStruc)
D us_Name 10
D us_dataauth 10
D us_objmgt 1
D us_objexist 1
D us_objalter 1
D us_objref 1
D us_rsv1 10
D us_objoper 1
D us_read 1
D us_add 1
D us_update 1
D us_delete 1
D us_execute 1
D us_excluse 1
D us_rsv2 7
*      
D ptrUserStruc s *
*  
  *  UTC Time Offset  (Sign Hours Minutes, ex: EST is -0500)
D dsUTCOffset ds Static
D Sign 1
D Hours 2s 0
D Minutes 2s 0
*      
D UTCAdjust s 10i 0 Static
*  
  *  Prototype for UTC Time Offset
D QWCRSVAL PR ExtPgm('QWCRSVAL')
D Buffer 100 Const
D BufferLen 10u 0 Const
D NoConsts 10u 0 Const
D ConstList 10 Const
D ErrorStuc Like(dsErrCode)
D Buffer S 100
C/Free
//  
//  Retrieve UTC Offset for this machine, the first time through

542  Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries
// If the time stamp shown by the function is incorrect, comment
// out the code in this IF/ENDIF block.

If dsUTCOffset=*Blanks;
    CallP(E) QWCRSVAL(Buffer:%Size(Buffer):1:
            'QUTCOFFSET':dsErrCode);
    // Calculate the number of seconds to adjust the Epoch Time
    // dsUTCOffset=%Subst(Buffer:93:5);
    UTCAdjust=Hours*3600 + Minutes*60;
    If Sign='-';
        UTCAdjust=-UTCAdjust;
    EndIf;
EndIf;

// Call API

pPathName=qPathName;
pPathNameLen=%Len(%Trim(qPathName));
pCountryID=CountryID;
pLangID=LangID;

// Retrieve Object Attributes

ReturnCode=Qp01GetAttr(%Addr(dsPath):
            %Addr(dsAttrArray):
            %Addr(RetBuffer):
            %Size(RetBuffer):
            %Addr(BufferSizeReq):
            %Addr(BytesReturned):0);

// If creation time attribute successfully, received then convert
// to date. (Creation time is given in the number of seconds
// passed since the Epoch time of Jan 1, 1970.)

If ReturnCode=*Zero;
    FileName=%Subst(DirEntry:1:DirEntryL);
    For i=1 To NoAttribs;
        Select;
        // Calculate Time Attributes
        // When AttrID=cATTR_CrtTime Or
        AttrID=cATTR_AccTime Or
        AttrID=cATTR_ModTime;
        // Add the number of seconds to the epoch time
        // Then add the number of seconds to adjust for the UTC offset
        // zResultDate=zEpochTime + %Seconds(AttrDataN)
            + %Seconds(UTCAdjust);
        // Fill in return parameters
        Select;
        When AttrID=cAttr_CrtTime;
            CreateStamp=zResultDate;
        EndIf;
    EndFor;
EndIf;
When AttrID=cAttr_AccTime;
  AccessStamp=zResultDate;
When AttrID=cAttr_ModTime;
  ModStamp=zResultDate;
EndSl;

//
// Set Object Type & Data Sizes
//
When AttrID=cATTR_ObjType;
  Type=%Subst(AttrData:1:10);
When AttrID=cATTR_DataSize;
  DataSize=AttrDataN;
When AttrID=cATTR_AlcSize;
  AllocSize=AttrDataN;
When AttrID=cATTR_Auth;
  ptrAuthority=%Addr(AttrData);
  Owner=Auth_Owner;

//
// To evaluate the user entries/authorities, cycle through the list as follows
//
//         ptrUserStruc=%Addr(RetBuffer)+Auth_offset;
//         For j=1 to Auth_NoUsers;
//             ptrUserStruc=ptrUserStruc+Auth_EntSize;
//         EndDo;
//
EndSl;

// Set Pointer for next attribute
//
ptrRetBuffer=%Addr(RetBuffer)+AttrOffset;
EndFor;
Return *Off;
Else;
  // Error Occurred
  pErrNo=GetErrNo;
  ErrNo=0;
  Return *On;
EndIf;
/EEND-FREE

You compile the program with the following CL command:

CRTBNDRPG PGM(SQLLIB/IFSDIR) SRCFILE(XXXXXX/QRPGLESRC) SRCMBR(IFSDIR)

After the program object is created in the library SQLLIB, register the external UDTF named IFSDIR with the following SQL statement:

CREATE FUNCTION SQLLIB.IFSDIR (IFSFOLDER VARCHAR(128))
RETURNS TABLE (FILENAME VARCHAR(128), CREATESTAMP TIMESTAMP, ACCESSSTAMP TIMESTAMP, MODSTAMP TIMESTAMP, TYPE CHAR(10), DATASIZE BIGINT, ALLOCSIZE BIGINT, OWNER CHAR(10))
EXTERNAL NAME 'SQLLIB/IFSDIR'
LANGUAGE RPGLE
Chapter 15. External user-defined functions

DISALLOW PARALLEL
RETURNS NULL ON NULL INPUT
PARAMETER STYLE DB2SQL;

1A. The column definition of the returned table at 1C must match those at 1A and 1B.
2. The parameter of EXTERNAL NAME must be enclosed between a pair of single quotation marks.

After a UDTF is registered, you must invoke it only from the FROM clause of the SELECT statement. It must be cast to a table type by the built-in TABLE() function as shown in the following example:

SELECT * FROM TABLE( SQLLIB.IFSDIR ('/QIBM') ) AS X;

Figure 15-16 shows a sample result of the IFSDIR UDTF.
The following code shows a sample ILE C program where the Retrieve File Description API is used. This program shows how to obtain information about the triggers associated with a physical file. The program can also be easily modified to retrieve information about the referential integrity constraints.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "QSYSINC/H/TRGBUF"
#include "QSYSINC/H/QDBRTVFD"

#define BUF_SIZE 70

void proc_fild0100(void);
struct error_code {
  int bytes_provided;
  int bytes_available;
  char message_id[7];
} error_code;

char buf[BUF_SIZE];
char in_data[200];
char return_data[5000];
Qdb_Rfd_Input_Parms_t *in_parms;
Qdb_Qdbfh_t *fdt_100;
Qdb_Qdbfphys_t *phy_100;
Qdb_Qdbftrg_t *trg_100;

main(int argc, char *argv[]) {
  char *library, *file;
  int i;

  in_parms = (Qdb_Rfd_Input_Parms_t *) in_data;

  proc_fild0100(void);
}
```
memset(in_parms->File_And_Library_Name, ' ', 20);
memset(in_parms->Record_Format_Name, ' ', 10);
if (argc == 1) /*.... Analyzing the parameter list ....*/
{
    printf("Invalid number of parameters\n");
    exit(1);
}
else if (argc >= 2)
{
    library = strtok(argv[1], "/");
    if((file = strtok(NULL, "/")) == NULL)
    {
        memcpy( in_parms->File_And_Library_Name, argv[1],
                strlen(argv[1]));
        memcpy( in_parms->File_And_Library_Name+10, "*LIBL", 5);
    }
    else
    {
        memcpy( in_parms->File_And_Library_Name, file,
                strlen(file));
        memcpy( in_parms->File_And_Library_Name+10, library,
                strlen(library));
    }
    if (argc >= 3)
        memcpy( in_parms->Record_Format_Name, argv[2],
                strlen(argv[2]));
    else
        memcpy( in_parms->Record_Format_Name,
                in_parms->File_And_Library_Name, 10);
}
for (i=0; i<20; i++)
{
    in_parms->File_And_Library_Name[i] =
        toupper(in_parms->File_And_Library_Name[i]);
}
for (i=0; i<10; i++)
{
    in_parms->Record_Format_Name[i] =
        toupper(in_parms->Record_Format_Name[i]);
}
memset(buf, ' ', BUF_SIZE);
/***********************************************************/
/* Set up the parameters that are passed for the API.*/
/***********************************************************/
in_parms->Length_Of_Receiver_Var = 5000;

memcpy( in_parms->File_Override_Flag, "0", 1);
memcpy( in_parms->System, "*LCL", 10);
memcpy( in_parms->Format_Type, "*EXT", 10);
memcpy( in_parms->Format_Name, "FILD0100", 8);
error_code.bytes_provided = 15;
/***********************************************************/
/* Call the API. */
/***********************************************************/
QDBRTVFD(return_data,
    in_parms->Length_Of_Receiver_Var,
    &inparms->Returned_File_And_Library,
    in_parms->Format_Name,
&in_parms->File_And_Library_Name,
    in_parms->Record_Format_Name,
    in_parms->File_OVERRIDE_Flag,
    in_parms->System,
    in_parms->Format_Type,
    &error_code);

/*******************************/
/* If the retrieve was successful. */
/*******************************/
if (error_code.bytes_available == 0)
{
    fdt_100 = (Qdb_Qdbfh_t *) return_data;
    /* If the file is a physical file. */
    if (!fdt_100->Qdbfhflg_t.Qdbfhfpl)
    {
        phy_100 = (Qdb_Qdbfphys_t *) (return_data + fdt_100->Qdbpfof);
        /* If the file has a valid number of triggers. */
        if ((phy_100->Qdbftrgn > 0) && (phy_100->Qdbftrgn < 7))
            proc_fild0100();
        else /* Else the file has invalid # of triggers. */
            printf("No triggers or invalid number of triggers...
");
    }
    else /* Else the file is not a physical file. */
        printf("The file is not a physical file...
");
}
else /* Else the retrieve failed. */
{
    printf("Bad error code from QDBRTVFD : %s\n",
        error_code.message_id);
    if (!strncmp(error_code.message_id, "CPF5715", 7))
        printf("File %10.10s in library %10.10s not found...
",
            in_parms->File_And_Library_Name,
            in_parms->File_And_Library_Name + 10);
}
} /* End of main program function. */
/*******************************/
/****** Process the format for FILD0100 on the api call. */
/*******************************/
void proc_fild0100() {
    int j;
    printf("Trigger information for file %10.10s in library %10.10s\n",
        in_parms->File_And_Library_Name+10,
        in_parms->File_And_Library_Name);
    printf("Number of triggers: %i\n", phy_100->Qdbftrgn);
    /* Set poointer to the trigger information area ....*/
    trg_100 = (Qdb_Qdbftrg_t *) (return_data + phy_100->Qdbfotrg);
    /* Print a header line and start for loop. */
    memset(buf, '\*', BUF_SIZE);
    buf[BUF_SIZE] = '\0';
    printf("\%s\n", buf);
    printf("    Physical File Trigger Information   \n");
    for (j=1; j <= phy_100->Qdbftrgn; j++) {
        printf("\%s\n", buf);
        printf("    Trigger program: %10.10s in library %10.10s\n",
            trg_100->Qdbftpgm, trg_100->Qdbftplb);
        printf("    Trigger Time: ");
        switch(trg_100->Qdbftrgt) /*... print TRIGGER TIME */
```c
{
    case '1': printf(" *AFTER\n");
        break;
    case '2': printf(" *BEFORE\n");
        break;
}
printf("Trigger Event:");
switch(*trg_100->Qdbftrge) /*... print TRIGGER EVENT ...*/
{
    case '1': printf(" *INSERT\n");
        break;
    case '2': printf(" *DELETE\n");
        break;
    case '3': printf(" *UPDATE");
        switch(*trg_100->Qdbftupd)
        {
            case '1': printf(" *ALWAYS\n");
                break;
            case '2': printf(" *CHANGE\n");
                break;
        }
        break;
}
/* Increment the pointer to the next trigger.*/
trg_100 ++;
} /* end of for loop in trigger processing */
```

On the display, this program shows the list of triggers associated with a database file. The list is retrieved by using the QDBRTVFD APIs. A complete description of this API can be found in the System API Programming, SC41-5800. This utility can be created and called in the following way:

```c
CRTBNDC PGM(T4249TRGI)
    SRCFILE(C)
    CALL T4249TRGI PARM('mylib/filename' 'Record-format-name')
```

The second parameter, if not specified, defaults to the file name. If you do not specify the library, it defaults to *LIBL.*
Order Entry application: Detailed flow

This appendix provides detailed flow charts of each of the modules included in the Order Entry application scenario. Figure B-1 shows a functional description of the various components of this application scenario.
Program flow for the Insert Order Header program

DB2 Universal Database for iSeries functional highlights in this program include:

- Referential integrity constraints for the Order Header table
- Insert trigger on the Order Header file

Figure B-1  Insert Order Header program flow
Program description for the Insert Order Header program

The idea of this program is to show how to use the following new database functions in a real application:

- **Referential integrity**: When a record is inserted in the Order Header file, the system checks for an existing customer in the Customer table.

- **Database trigger**: Before the insert operation is completed, the database manager activates a program that can verify if the sales representative is assigned to the customer, and log any violation attempt.

- **Program description**: The sales person periodically calls the customer over the phone and places an order. The sales person enters the customer number, the order and delivery date, and other general information. Our application does not automatically generate an order number. For the sake of simplicity, this is entered by the sales representative.

A more detailed flow of this program is described as follows:

1. The program inserts a row into the Order Header table.
2. If the database referential constraint enforcement detects a customer number not defined in the Customer table, a program message is sent explaining that the customer number is invalid. A correct customer number must be entered.
3. The customer name is displayed at the terminal.
4. A row is inserted into the Order Header table.
5. Since an insert trigger is defined on this table, a program is automatically triggered by the database manager.
6. The trigger program checks if the current user profile is associated with the customer in the Sales/Customer table. If there is no match, the program writes an audit trail entry to an audit table.
7. If the insert is successful, the program returns a positive return code to the main program, which calls the Insert Order Detail program.

Program flow for the Insert Order Detail program

DB2 Universal Database for iSeries functional highlights in this program include:

- Referential integrity constraints for the Order Detail table
- Referential integrity constraints for the Stock table (on remote system)
- Two-phase commit and DRDA Level 2
- Remote stored procedure
The program flow for Insert Order Detail is shown in Figure B-2.
Program description for the Insert Order Detail program

The idea of this program is to show how to use the following new database functions in a real application:

- **Referential integrity:** When a record is inserted into the Order Detail table for a new order item, the system checks for a matching order number in the Order Header table.

- **Two-phase Commit with DRDA, Level 2:** This procedure inserts a record in a local file and updates the remote inventory file (STOCK file). At the end of this process, you want to release the locks on the inventory record, and the transaction is committed. The two-phase commit support guarantees the integrity of this transaction.

- **Stored procedure:** To update the remote inventory file, this program calls a remote stored procedure. The stored procedure checks the availability of the product. If the product has low inventory levels, the stored procedure looks for an alternative and sends the new product code and description back to the calling application. The selected product information is displayed at the terminal and the user has the choice of accepting or rejecting the substitute item.

- **Program description:** This program can:
  a. Get the customer number and the order number from the Insert Order Header program.
  b. Get the product number and quantity for every single item from the display.
  c. Issue a SET CONNECTION statement to the remote system. All the necessary CONNECT statements are performed by the main program.
  d. Call a stored procedure at the remote system to:
     - Look for the product number in the remote inventory.
     - Update the Stock table, reducing the quantity on hand if the quantity available is sufficient.
     - Look for an alternative product if the requested one is out of stock, and update the corresponding quantity.
     - Pass the product information back to the calling program.
  e. The stored procedure then passes control back to the calling program.
  f. At this point, the program sets a connection to the local system, and if the user accepts the record, the new item is inserted in the Order Detail file, and the whole transaction is committed. If the user rejects the item, a rollback brings the stock quantity on hand back to its original value.
  g. A rollback is also performed if the referential integrity checking on the Order Detail table fails. This happens if you insert the record with the wrong order number.
  h. The user also has the option of cancelling the whole order. In this case, a Cancel Order program is called.
  i. The program keeps a work field with the final totals of the whole order. When the entire order is completed, this value is passed to the next program—Finalize Order.
Program flow for the Finalize Order program

DB2 Universal Database for iSeries functional highlights in this program include the trigger on the Update Order Header row. See Figure B-3 for the program flow.

Figure B-3  Finalize Order program flows
Program description for the Finalize Order program

The idea of this program is to show how to use the following new database functions in a real application:

- **Database triggers**: In this scenario, a program is triggered after the Order Header row is updated with the total amount of the order. This program prints the invoice at the branch office as soon as the order has been completed.

  The program also updates the credit limit on the customer file. If the current balance exceeds 90 percent of the credit limit, a “warning” fax is automatically sent to the customer by a trigger program to allow the customer to take the appropriate actions (for example, apply for a credit limit increase, based on the credit history of the customer).

- **Program description**: This program can:

  a. Get the customer number and the order number from the previous process along with the order grand total.

  b. Check the customer record. If the credit limit is exceeded, the order is cancelled. To delete the order, the detail is scanned, and the inventory quantity that is on hand for each item is updated by adding the amount reserved for this order. When this process is complete, the Order Header is deleted, and all the order detail disappears as a result of the *CASCADE constraint on the Order Header file. The entire transaction is finally committed. Again, the two-phase commit support ensures that the local database and the remote stock file are kept synchronized.

  c. If the credit limit is OK, this program updates the following fields:

      - The total amount in the customer file to keep track of the customer balance
      - The total amount in the Sales Representative/Customer table to reflect the sales person’s turnover with the customer
      - The total amount in the Order Header table items at invoice time

  d. Because an update trigger is specified on the Order Header table, an invoice program is started immediately. The invoice for the completed order is printed in the branch office. For more information about triggers, see Part 3, “Triggers” on page 279.

  e. After the preceding updates are done, COMMIT is executed.

  f. If there are more orders, the Insert Order Header program is started again.

  g. If there are no more orders, this Order Entry application has ended.
Stored procedures and trigger porting considerations

For the latest comprehensive information and white papers on relational database management system (RDBMS) porting guides to DB2 Universal Database for iSeries from other DBMS, refer to the Migration Toolkit Web site at:


This Web site offers information about:

- Porting stored procedures from SQL Server 7.0 to DB2 Universal Database for iSeries
- Porting stored procedures from Oracle to DB2 Universal Database for iSeries
- Porting stored procedures from DB2 Universal Database V7.1 UWO (UNIX, Windows, OS/2®) to DB2 Universal Database for iSeries
- Porting Triggers from SQL Server 7.0 to DB2 Universal Database for iSeries
- Porting Triggers from Oracle to DB2 Universal Database for iSeries
- Porting Triggers from DB2 Universal Database V7.1 UWO (UNIX, Windows, OS/2) to DB2 Universal Database for iSeries
Additional material

This IBM Redbook also contains additional material that is available on the Web. See the following sections for instructions on using or downloading the material.

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/SG246503

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, SG246503.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbadvfun.zip</td>
<td>iSeries and client source image</td>
</tr>
<tr>
<td>readme.zip</td>
<td>Readme documentation</td>
</tr>
</tbody>
</table>

System requirements for downloading the Web material

The following list contains the most important requirements:

- iSeries requirements
  - OS/400 Version 5 Release 1
  - 5722-ST1 - DB2 Query Manager and SQL Development kit
  - 5722-SS1 - Host servers
PC Software
- Windows 95/98, Windows NT or Windows 2000
- iSeries Access Express for Windows
- PC5250 Emulation

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.

The readme.txt contains the instructions for restoring the iSeries libraries and directories, as well as installing the PC clients and run-time notes.
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 about ordering these publications, see “How to get IBM Redbooks” on page 564.

- A Fast Path to AS/400 Client/Server Using AS/400 OLE DB Support, SG24-5183
- Advanced Functions and Administration on DB2 Universal Database for iSeries, SG24-4249
- DB2 UDB for AS/400 Object Relational Support, SG24-5409
- DB2 Universal Database for iSeries Administration: The Graphical Way on V5R3, SG24-6092
- Building Java Applications for the iSeries Server with VisualAge for Java, SG24-6245

Other resources

These publications are also relevant as further information sources:

- Backup and Recovery, SC41-5304
- Database Programming, SC41-5701
- SQL Programming Guide, SC41-5611
- SQL Reference, SC41-5612
- DB2 Universal Database for iSeries SQL Messages and Codes

This document is located at the iSeries Information Center at:


Simply click Database and file systems -> DB2 UDB for iSeries-> Manuals and redbooks->SQL Messages and Codes.

Referenced Web sites

These Web sites are also relevant as further information sources:

- IBM Toolbox for Java and JTOpen
  http://www.iSeries.ibm.com/toolbox
- DB2 Universal Database for iSeries
  http://www.ibm.com/iSeries/db2
- iSeries Information Center

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:
ibm.com/redbooks

Help from IBM

IBM Support and downloads
ibm.com/support

IBM Global Services
ibm.com/services
Index

A
access path 5
ACTGRP(“CALLER”) 434
activation group 257, 263
creation 435
ADDPFTRG 359
ADO 104
  retrieving result sets 104
adopted authority 36, 38, 78, 292
advanced technology 283
AFTER trigger 323
Allow Repeated Change (ALWREPCHG) 332
application
  call using SQL 63
  example 11
  independence 283
  integration with advanced technology 283
  logic 18
  logic simplified 18
  maintenance 66, 283
  with commitment control 444
application development
  improved ease 439
  modularity in 66
application requester (AR) 6
AR (application requester) 6
array result sets 150
assignment statement 21
atomic transactions with triggers 378
audit trail 283
authority 38
authorization 78, 292

B
BEFORE trigger 322
benefits
  distributed environment 64
  stored procedures 18, 64
business rules 282

C
C trigger 373
  buffer example 369
C++ client code using ODBC 272
call message queue 376
calling
  external procedures 29
  external stored procedure 136
  Java stored procedures 206
CASE statement 23
CATCH block 247, 251
change in database 282
character conversion 203
check constraint 223
check_constraint_error 223
CL triggers 373
clearance marker 64
CLI client
  error conditions 253
  retrieving result sets 151
client and server systems distributing logic 64
client application, user-defined errors 250
client code, Java version 275
client program 151
client/server application
  development 283
  with stored procedures 64
ClientGetSupplier 216
  Java client 206
ClientGetSupplierGUI 216
  Java GUI client 210
CMTLVL 370
COBOL triggers 373
code reusability 283
  improvement 66
columns in SQL 5
combining results with stored procedures 64
command, CL
  ADDPFTRG 359
  CMTLVL 370
  RMVPFTRG 362
  SBMRMTCMD (Submit Remote Command) 66
  Submit Remote Command (SBMRMTCMD) 66
COMMIT in SQL stored procedure 259
commit lock level 434
COMMIT statement 259
commitment control 256
  in stored procedures 263
  with referential integrity 444
commitment definition 378
  triggers 434
communications network performance 64
compiling a client program 151
compiling Java code 185
compiling Java stored procedures 185
completion status for stored procedures 242
compound SQL 84
  procedure 29
compound statement
  nested 30
  with transaction management 261
CONCATENATE operator 22
condition declaration 223
conditional control 278
consistency
  in data across different files 283
  of data 378, 439
consistent error handling 222
null
| **EXECUTE IMMEDIATE statement** | 35 |
| **EXECUTE statement** | 35 |
| **external procedure** | 82 |
| calling from SQL procedure | 29 |
| **external stored procedures** | 63, 67, 123 |
| calling | 143 |
| coding | 131 |
| coding DB2SQL parameter style | 137 |
| coding GENERAL WITH NULLS parameter style | 140 |
| coding SQL parameter style | 132 |
| error handling | 242 |
| example | 268 |
| parameter styles | 129 |
| registering | 124 |
| registering with Operations Navigator | 124 |
| returning array result sets | 150 |
| returning cursor result sets | 144 |
| with SQL parameter style | 136 |
| **external trigger** | 285, 357 |

**F**
- failures detected by triggers | 372 |
- field | 5 |
- FOR EACH ROW | 326 |
- FOR EACH STATEMENT | 327 |
- FOR statement | 26 |

**G**
- GENERAL parameter style | 131, 246 |
- GENERAL WITH NULLS parameter style | 131, 246 |
- coding example | 140 |
- Generate SQL | 55 |
- GET DIAGNOSTICS EXCEPTION statement | 232 |
- GET DIAGNOSTICS statement | 232 |
- GetSupplier stored procedure | 204 |
- GetSuppliers example | 111, 211 |
- implementation with no result sets | 204 |
- stored procedure with Java parameter style | 211 |
- stored procedure with the DB2GENERAL parameter style | 214 |
- Global Temporary Table | 106 |
- access from SQL trigger | 354 |
- result set access | 110 |
- result set storage | 107 |
- granularity | 321 |
- granularity of triggers | 326 |

**H**
- handler declaration | 223 |
- in an SQL stored procedure | 224 |

**I**
- I/O feedback area | 373 |
- IBMDA400 provider | 104 |
- IF-THEN statement | 22 |
- IF-THEN-ELSE statement | 23 |
- IF-THEN-ELSEIF statement | 23 |
- ILE recommendations | 434 |
- ILE source debugger | 38, 46, 343, 488 |
- image record | 372 |
- index in SQL | 5 |
- inoperative trigger | 337 |
- insert operations | 440 |
- insert order detail | 158 |
- Instead Of Trigger | 355 |
- integrated relational database | 4 |
- integrity referential | 439 |
- invocation referential | 439 |
- Isolation level | 256 |
- ITERATE simulation | 28 |
- ITERATE statement | 28 |
- iterative control statement | 24 |

**J**
- JAR file | 198 |
- jar-id | 198 |
- Java calling stored procedure | 206 |
- Java classes | 186 |
- Java clients | 216 |
- Java connection object | 177 |
- Java GUI client calling stored procedure | 210 |
- Java parameter style | 174, 178 |
- result sets | 178 |
- Java programs | 187 |
- Java property file | 253 |
- Java SQL exception | 247 |
- Java stored procedures | 68, 123 |
- calling | 193 |
- coding DB2 Universal Database for iSeries | 174 |
- compiling code | 185 |
- consistent error handling | 248 |
- creating Java programs | 187 |
- database connection | 177 |
- DB2GENERAL parameter style | 175 |
- error handling | 247 |
- example | 270 |
- Java parameter style | 174 |
- JDBC | 177 |
- problem determination | 217 |
- registering | 188 |
- registering with Operations Navigator | 189 |
- result sets | 211 |
- returning result sets | 178 |
- Run SQL Scripts utility | 192 |
- See also stored procedures Java
- SQLJ | 178 |
- using SQL NULL | 196 |
- using the native interface | 192 |
- where to place Java classes | 186 |
- JavaInsertCus | 185–186, 193 |
- stored procedure | 181 |
- JavaScriptInsertCus | 181, 185–186, 193 |
- JDBC
  - coding examples | 180 |
  - compilation | 186 |
  - in Java stored procedure | 177 |
  - Open Source driver | 248 |
- JDBC client error conditions | 251 |
job log 373
job stack 376
journal entries with triggers and referential integrity 445

K
keyed logical file 5

L
LEAVE statement 27
library list 434
local system parameter passing 64
lock level 434
logical consistency example 12
logical file 5
logical unit of work (LUW) 256
LOOP statement 24
looping structure 24
LUW (logical unit of work) 256

M
maintenance 283
improved ease 439
map record 372
message queue call 376
messages
CPF502B 373
signaling to database interface 375
MODE DB2ROW 324
MODE DB2SQL 323
MODIFIES SQL DATA 69
modularity in application development 66
MSDASQL provider 104

N
native interface 192
nested compound statement 30
nested savepoint 256, 258
nested SQL procedure 79
network communications performance 64
NEW ROW AS correlation-name 330
NEW_TABLE 330
NO SQL 69
non-database functions 64
NOT FOUND 224
NULL in Java stored procedure 196

O
object qualification 338
object save and restore 156
object saving and restoring 203
ODBC
C++ client code 272
Client Access driver 104
driver 248
error conditions 253
OLD ROW AS correlation-name 330
OLD_TABLE 330
OLE DB provider 104
open data path 435
open database files 429
operations at a remote site 64
Operations Navigator
create Java stored procedure 189
creating SQL triggers 301
external stored procedure 124
result sets 114
SQL stored procedure 85
stored procedures properties 102, 485
order detail insert 158
Order Entry application 11–12
calling a stored procedure 158
database overview 13
DB2 stored procedures and triggers 17
stored procedures examples 157
ordering of actions 440
orientation 321
overloaded stored procedures 75, 77

P
parameter styles
external stored procedures 129
GENERAL 131
GENERAL WITH NULLS 131
SQL 130
passing parameters 64, 372
performance
distributed applications 64
how to improve 439
improvements from stored procedures 18
triggers 435
performing in a sequence with stored procedures 64
Persistent Stored Module (PSM) 19–20, 343
physical data 5
physical files 5
associated with triggers 358
trigger activation 439
PREPARE statement 34
Print Trigger Programs (PRTTRGPGM) command 288
problem determination 217
procedure overloading 132
procedures 64
program failure while trigger program running 372
program structure triggers 364
programming development 439
properties of a stored procedure 102
property file 253
PSM (Persistent Stored Module) 19–20, 343
public static void method 174
public void method 176

Q
QDBPUT 373
QMHSNDPM 373
query functions at a remote site 64
SQL call from an application 63
SQL columns 5
SQL compound statement 83
SQL CONNECT 151
SQL control statement 21
SQL NULL value 196
SQL parameter style 130, 243
coding example 132
SQL procedure
calling external procedures 29
compound 29
scrollable cursor 33
SQLCODE, SQLSTATE 231
SQL RPG stored procedure 164
SQL statements
COMMitt 259
compound 84
CREATE PROCEDURE 68, 132, 512
DECLARE PROCEDURE 70
LEAVE 27
LOOP 24, 26
REPEAT 25
ROLLBACK 260
SAVEPOINT 260
SET TRANSACTIONS 260
single 83
WHILE 25
SQL stored procedure 19, 67, 81, 123
adopted authority 36
calling from Visual Basic client application 115
consistent error handling 248
creating 85, 111
creating with 5250 tools 98
creating with Operations Navigator 85
creating with Run SQL Scripts 88
dynamic SQL 34
error handling 223
eXample 267
planning 20
See also stored procedures SQL
structure 21, 82
system requirements 20
testing 49
testing in a distributed environment 52
verifying properties 102
SQL trigger 284, 295–296
access to a Global Temporary Table 354
authority 342
creating 301
creating with Operations Navigator 301
creating with traditional interfaces 312
debugging 344
definition components 298
deleting, replacing 318
error handling 332
example 300
examples 350
invoking external programs 351
invoking Java stored procedures or UDFs 352
modes 323
program object 342
Run SQL Scripts utility 308
saving, restoring definitions 338
structure 297
system requirements, planning 296
testing 346
testing in a client/server environment 347
transition tables 330
verifying properties 316
SQL view 5
SQLCODE 222, 231, 242, 254
SQLERRORO() function 253
SQLEXCEPTION 224
SQLException 247, 249
SQLJ
coding examples 180
translation and compilation 186
using in Java stored procedure 178
SQLJ procedures 198
SQLJ.INSTALL_JAR 199
SQLJ.RECOVERJAR 202
SQLJ.REMOVE_JAR 201
SQLJ.REPLACE_JAR 202
SQLJ.UPDATEJARINFO 202
SQLMoreResults function 154
SQLSTATE 222, 224, 231
code 254
user-defined 243, 247
Sqlstate parameter 242
SQLWARNING 224
SQLWARNING in SQL stored procedure 225
statement-level trigger, row-level trigger 321
static SQL 34
statistical data, generating with stored procedures 64
stored procedures 6, 61, 63
calling 158
commitment control 263
completion status 242
DB2 Universal Database for iSeries 6
debugging 155
deleting/replacing 76
developing in SQL 19, 81
dropping overloaded procedures 77
error handling 221
eXamples 157
external 63
calling 136
coding with GENERAL WITH NULLS parameter style 140
coding with SQL parameter style 135
create with Operations Navigator 124
error handling 242
parameter styles 129
returning result sets 144
user-defined SQLSTATE 135
external call 63
invocation 64
Java
calling 193
compile 185
Stored procedures, triggers, and user-defined functions (UDFs) are the key database features for developing robust and distributed applications. DB2 Universal Database for iSeries has supported these features for many years, and they have been enhanced in V5R1, V5R2, and V5R3 of IBM OS/400 and V5R4 of IBM i5/OS.

This IBM Redbook includes some of the announced features for stored procedures, triggers, and UDFs in V5R1, V5R2, V5R3, and V5R4. Among the topics in this IBM Redbook, you will find suggestions, guidelines, and practical examples on how to effectively develop DB2 Universal Database for iSeries stored procedures, triggers, and UDFs. Some of the topics that are covered in this book include:

- Introduction to the SQL Persistent Stored Module Language
- SQL stored procedures
- External stored procedures and triggers
- Java stored procedures
- SQL triggers
- External triggers
- SQL UDFs
- External UDFs

This IBM Redbook also offers examples that were developed in several programming languages, including RPG, COBOL, C, Java, and Visual Basic, using native and SQL data access interfaces.