Redbooks
ibm.com/redbooks

External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i

Hernando Bedoya

Fredy Cruz

Daniel Lema

Satid Singkorapoom

Information Management

IBM

Redbooks

International Technical Support Organization

External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i

April 2016

SG24-6503-03

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

Fourth Edition (April 2016)

This edition applies to V5R1, V5R2, and V5R3 of IBM OS/400 and V5R4 of IBM i5/OS, Program Number 5722-SS1.

Contents

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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 grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

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 jurisdictions 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

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

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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 actual people or business enterprises 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. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

DB2®	Integrated Language Environment®	Redbooks (logo)  ®
DB2 Universal Database™	Language Environment®	System i®
DRDA®	Lotus®	VisualAge®
eServer™	Operating System/400®	WebSphere®
i5/OS™	OS/400®	
IBM®	Redbooks®	

The following terms are trademarks of other companies:

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

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

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

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

Preface

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

This IBM Redbooks® publication includes several of the announced features for procedures, triggers, and UDFs in V5R1, V5R2, V5R3, and V5R4. This book includes suggestions, guidelines, and practical examples to help you effectively develop IBM DB2 for i procedures, triggers, and UDFs. The following topics are covered in this book:

- External stored procedures and triggers
- Java procedures (both Java Database Connectivity (JDBC) and Structured Query Language for Java (SQLJ))
- External triggers
- External UDFs

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

This book is part of the original IBM Redbooks publication, *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*, SG24-6503-02, that covered external procedures, triggers, and functions, and also SQL procedures, triggers, and functions. All of the information that relates to external routines was left in this publication. All of the information that relates to SQL routines was rewritten and updated. This information is in the new IBM Redbooks publication, *SQL Procedures, Triggers, and Functions on DB2 for i,* SG24-8326.

This book is intended for anyone who wants to develop IBM DB2 for i procedures, triggers, and UDFs. Before you read this book, you need to know about relational database technology and the application development environment on the IBM i server.

Note: With the release of IBM i5/OS V5R4, the name of DB2 Universal Database for iSeries changed to *DB2 for i*.

Authors

This book 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 for i. Before Hernando joined the ITSO more than six years ago, he worked for IBM Colombia as an IBM AS/400 IT Specialist performing presales support for the Andean countries. He has 24 years of experience in the computing field and 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 Independent Software Vendor (ISV) Coordinator at IBM Colombia. He helps developers with infrastructure migration over software, including IBM WebSphere® and DB2 on both Linux on IBM System i® and i5/OS. He also teaches ISVs and clients about the utilities and technologies that relate to this task. His responsibilities are to show clients, IBM System i sales representatives, and IBM System i technical representatives how to implement the new utilities in the IBM System i environment. His areas of expertise include working with the Linux and Microsoft Windows environments on the System i platform, DB2 and WebSphere in OS/400, i5/OS, and Linux, and IBM Lotus® over the OS/400 and i5/OS environment.

Daniel Lema is an IT Architect at IBM Andean, with 15 years of experience. Several 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 clients and salespeople to design AS/400 and DB2/400 solutions. He was 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 finished his academic activities.

Satid Singkorapoom is an Advisory Product Specialist for the IBM 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 for 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 coauthored five IBM Redbooks publications and three DB2 technical training materials from ITSO Rochester over the past 13 years. He also teaches System i clients regularly about various product technology updates, SQL performance analysis and tuning, OS/400 for SAP deployment, and System i administration.

This book is based on projects that were conducted in 1994, 1997, 2000, 2001, and 2006 by the ITSO Rochester Center.

The following individuals were advisors of the projects:

Michele Chilanti
Jarek Miszczyk
ITSO Poughkeepsie Center

The following authors were involved in the previous editions of this book:

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

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an email 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

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

Summary of changes

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

Summary of Changes
for SG24-6503-03
for External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i
as created or updated on April 1, 2016.

April 2016, Fourth Edition

This revision includes the following new and changed information.

New information

This publication includes several of the announced features for procedures, triggers, and UDFs in V5R1, V5R2, V5R3, and V5R4. This book includes suggestions, guidelines, and practical examples about how to effectively develop IBM DB2 for i procedures, triggers, and UDFs. The following topics are covered in this book:

- External stored procedures and triggers
- Java procedures (both Java Database Connectivity (JDBC) and Structured Query Language for Java (SQLJ))
- External triggers
- External UDFs

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

Changed information

This book is part of the original IBM Redbooks publication, *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*, SG24-6503-02, that covered external procedures, triggers, and functions and also SQL procedures, triggers, and functions. All of the information that relates to external routines was left in this publication. All of the information that relates to SQL routines was rewritten and updated. It is in the new IBM Redbooks publication, *SQL Procedures, Triggers, and Functions on DB2 for i,* SG24-8326.

Find and read thousands of IBM Redbooks publications

- Search, bookmark, save and organize favorites
- Get personalized notifications of new content
- Link to the latest Redbooks blogs and videos

Get the latest version of the Redbooks Mobile App

iOS **Download Now** Android

Promote your business in an IBM Redbooks publication

Place a Sponsorship Promotion in an IBM® Redbooks® publication, featuring your business or solution with a link to your web site.

Qualified IBM Business Partners may place a full page promotion in the most popular Redbooks publications. Imagine the power of being seen by users who download millions of Redbooks publications each year!

ibm.com/Redbooks
About Redbooks → Business Partner Programs

THIS PAGE INTENTIONALLY LEFT BLANK

1

Introducing IBM DB2 for i

This chapter includes the following topics:

- An integrated relational database
- DB2 for i overview
- DB2 for i sample schema

Note: With the release of IBM i5/OS V5R4, the name of DB2 Universal Database for iSeries changed to *DB2 for i*.

1.1 An integrated relational database

Integration is one of the major elements of differentiation of the IBM i on Power Systems server in the information technology marketplace. The advantages and drawbacks of fully integrated systems were the subject of endless disputes in the last few years. The success of the AS/400 system, iSeries server, and the IBM i on Power Systems 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 were designed in an integrated way on the AS/400 system, iSeries server, and IBM i on Power Systems server. They work according to a common logic with a common user interface. They fit together perfectly because all of them are part of the same software, the IBM Operating System/400® (OS/400), IBM i5/OS, or IBM i.

The integrated relational database manager was always one of the most significant facilities that the IBM i on Power Systems server provided to users. Relying on a database manager that is integrated into the operating system means that virtually all of the user data on the IBM i on Power Systems server is stored in a relational database and that the access to the database is implemented by the operating system itself. Certain database functions are implemented at a low level in the IBM i on Power Systems server architecture, while other database functions are even performed by the hardware.

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

During the last few years, each new release of OS/400 enhanced DB2 for i with a dramatic set of new functions. As a result of these enhancements, the IBM i server is one of the most functionally rich relational platforms in the industry.

DB2 for i is a member of the DB2 Universal Database family of products, which includes DB2 Universal Database for OS/390 and DB2 for i. 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 participate in the family.

1.2 DB2 for i overview

This section provides a quick overview of the major features of DB2 for i. You can obtain 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 for i basics

The major distinguishing characteristic of the DB2 for i database manager is that it is part of the operating system. Most of your IBM i on Power Systems server data is stored in the relational database. Although the IBM i server also implements other file systems in its design, the relational database on the IBM i server is the most commonly used database by the clients. 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 IBM i server is stored in objects that are 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 for i, which is 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 that are provided by the *Structured Query Language* (SQL). This capability is also possible on the IBM i server. The SQL terminology can be mapped to the native DB2 for i 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 for i, 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 column subsetting, record selection, joining multiple database files, and so on. Logical files 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.

Similar concepts exist on the SQL side. 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 in the same way as a keyed logical file. Still, SQL views and indexes are treated differently than native logical files by DB2 for i, and they cannot be considered the same.

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

Terminology

Because DB2 for i evolved from the built-in database that was present in the iSeries and the AS/400 before SQL was widely used, IBM i uses different terminology than SQL to refer to database objects.

The terms and their SQL equivalents are in Table 1-1. The terms are used interchangeably throughout this book.

Table 1-1 Cross-reference of SQL terms and IBM i terms

SQL term	IBM i term
Table	Physical file
View	Non-keyed logical file
Index	Keyed logical file
Column	Field
Row	Record
Schema	Library, collection, or schema
Log	Journal
Isolation level	Commitment control level

1.2.2 Stored procedures, triggers, and user-defined functions

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

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 for i 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 for i 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 various purposes in your database design.

Two examples are data validation and audit trail creation. DB2 for i supports two types of triggers:

- SQL triggers
- External triggers

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 receive parameters, perform operations, and return a unique value, such as converting Fahrenheit to Celsius degrees or calculating the net present value when provided the final amount, monthly payment, number of payments, and interest rate.

UDTFs are functions that return a table for a certain 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 specific pair of currencies.

DB2 for i supports three types of UDFs:

- SQL UDFs
- External UDFs
- Sourced UDFs

1.3 DB2 for i sample schema

Within the code of OS/400 V5R1M0, a stored procedure 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 because the program ships with the OS/400 V5R1M0 code. By calling a simple program, you can create a duplicate of this database on any system that is running V5R1M0. This capability enables clients and support staff to work on the same database for problem determination.

Working on the same database provides the ability for clients 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 client.

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 list box of the Run SQL Scripts window (Figure 1-1).

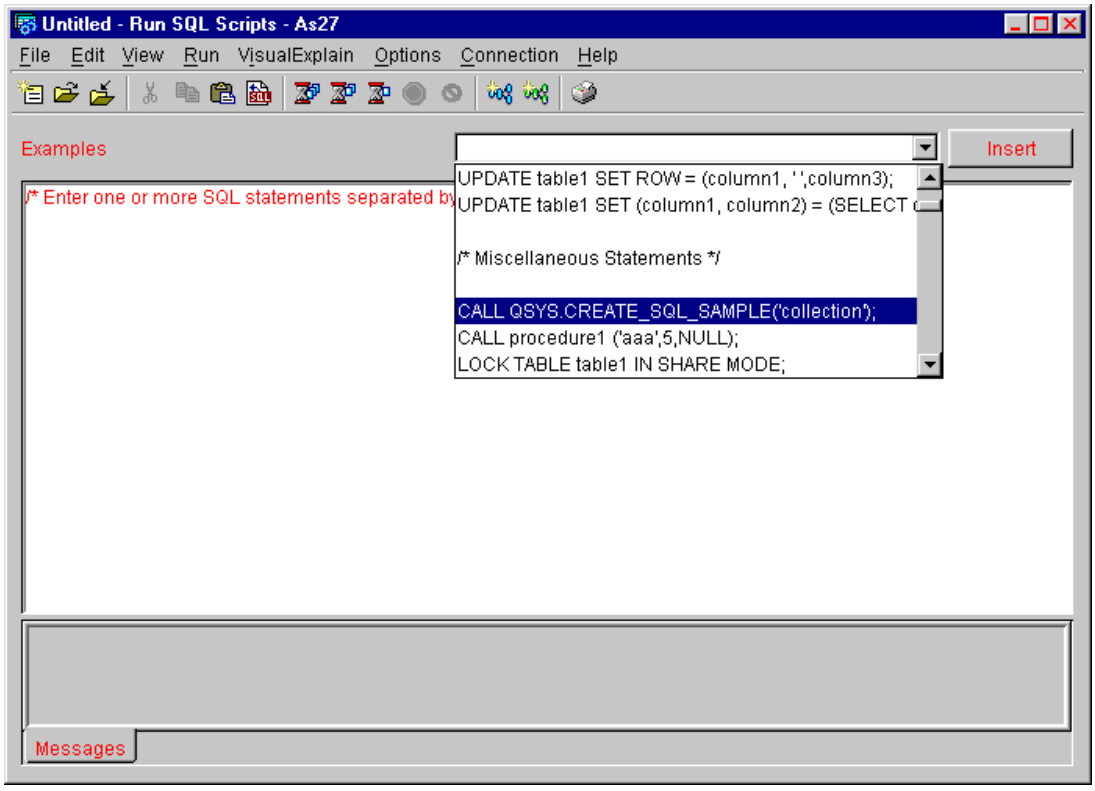


Figure 1-1 Example display that shows the schema CREATE statement

Note: The schema name must be uppercase. This sample schema is also used in future DB2 for i documentation.

As a group, the tables include information that describes employees, departments, projects, and activities. This information makes up a sample database that demonstrates the features of DB2 for i.

An entity-relationship (ER) diagram of the database is shown in Figure 1-2.

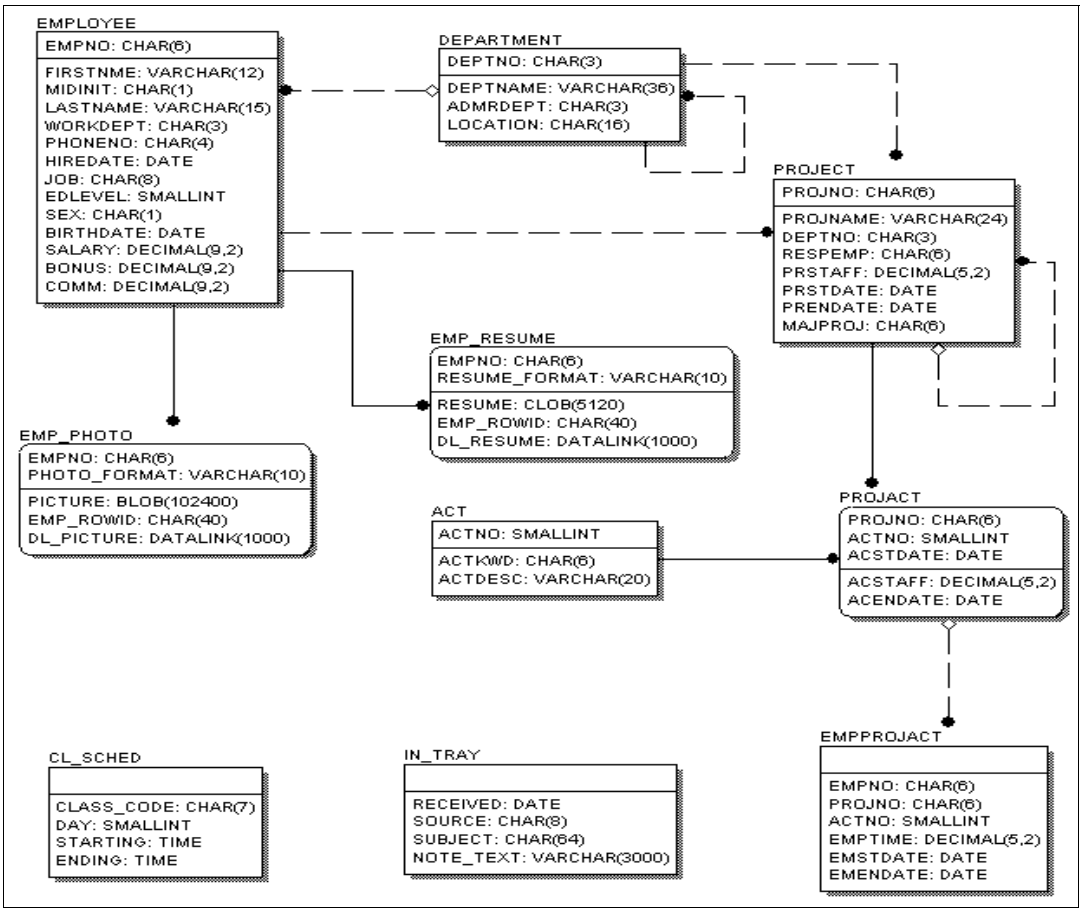


Figure 1-2 Sample schema: ER diagram

The tables are listed:

- 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. Three other tables are created that are not related to the first set:

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

Note: Several of the examples in this book use the sample database that was described.

2

Stored procedures, triggers, and user-defined functions for an 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 that is available with DB2 for i. It describes the complete application, in terms of logical flow and database structure. You can obtain the actual implementation of this application in the specific chapters that use 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 might seem simplistic in certain respects (for example, the user interface or specific design choices). We present a simple, easy-to-understand scenario that includes most of the aspects that are described throughout this book.

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

This chapter covers the following topics:

- Order Entry application overview
- Order Entry database overview
- Stored procedures and triggers in the Order Entry database

2.1 Order Entry application overview

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

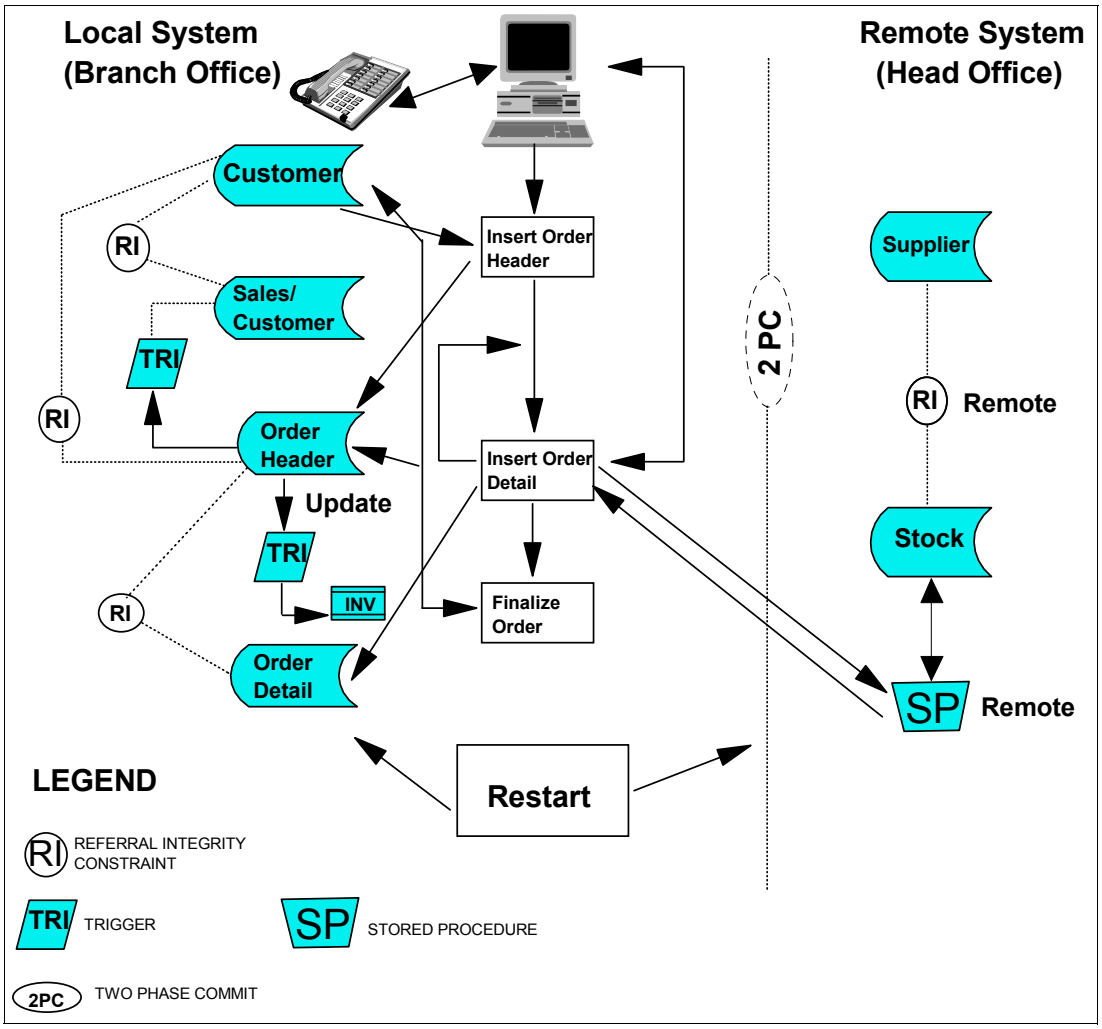


Figure 2-1 Application overview: Interaction of the DB2 Universal Database for iSeries functions

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.
- Therefore, data is stored in a distributed relational database. Information about customers and orders are stored at the branch office. 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 relate to a customer, and all of the products in the inventory must relate 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 using a trigger program that automatically checks the relationship between a customer and the sales representative when the order is placed. (See 8.4.2, "Audit trail trigger example programs" on page 244.)
- When the sales representative places an order, the sales representative first introduces general data, such as the order date and the customer code. 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. Because the inventory table is remote, we use an IBM Distributed Relational Database Architecture (DRDA®) connection between the systems. In addition, because the process of searching the inventory might involve many accesses to the remote database, a stored procedure is called to carry out this task.
- When the item or a replacement is identified, the inventory is updated, and a row is inserted in the local order detail table.
- We want to release the inventory row to allow other people to place a new order for the same product. We commit the transaction now. 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 are entered, the order is finished and a finalizing order program is called. This program can perform the following functions:
  - Add the total amount of the order to the Customer table to reflect the customers' turnover.
  - Update the total revenue that is 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.
- The "atomic" logical transaction is completed when a single item in the order is inserted to reduce the locking exposures. If the system or the job fails, we must be able to detect incomplete orders, which can be done when the user restarts the application. A simple restart procedure will check for orders with the total equal to zero (not "finalized"). These orders are deleted and the stock quantity of all of 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 need to be finalized.

2.2 Order Entry database overview

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

The local database consists of these tables, as shown in Figure 2-2:

- **CUSTOMER** table: Contains the information that relates to the customers
- **ORDERHDR** table: With the data that relates 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 that relates to the suppliers

Figure 2-2 shows an Entity-Relationship chart of the database model.

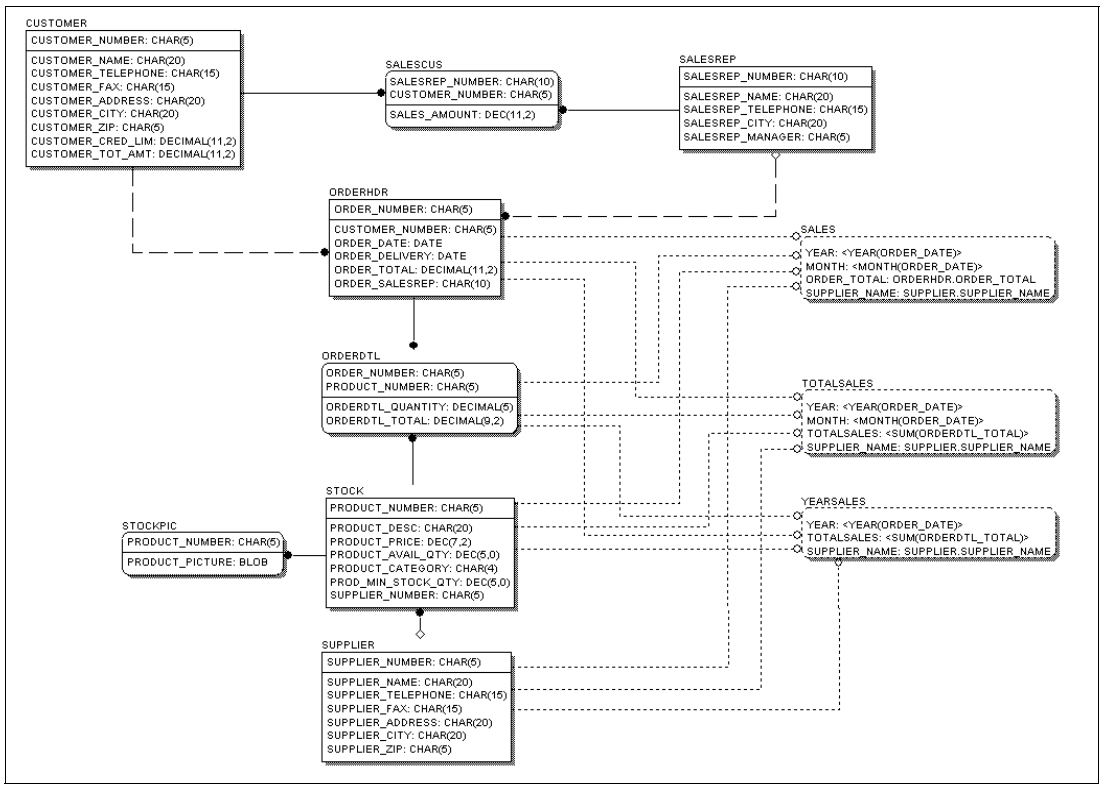


Figure 2-2 Order Entry database model

Table 2-1 through Table 2-8 on page 15 show the row layouts for the tables of both the local and central databases.

Table 2-1 shows the CUSTOMER table.

Table 2-1 CUSTOMER table

Field name	Alias	Type	Description
CUSTOMER_NUMBER	CUSBR	CHAR(5)	Customer number
CUSTOMER_NAME	CUSNAM	CHAR(20)	Customer name
CUSTOMER_TELEPHONE	CUSTEL	CHAR(15)	Customer phone number
CUSTOMER_FAX	CUSFAX	CHAR(15)	Customer fax number
CUSTOMER_ADDRESS	CUSADR	CHAR(20)	Customer address
CUSTOMER_CITY	CUSCTY	CHAR(20)	Customer city
CUSTOMER_ZIP	CUSZIP	CHAR(5)	Customer ZIP code
CUSTOMER_CRED_LIM	CUSCRD	DEC(11,2)	Customer credit limit
CUSTOMER_TOT_AMT	CUSTOT	DEC(11,2)	Customer total amount

Table 2-2 shows the ORDERHDR table.

Table 2-2 ORDERHDR table

Field name	Alias	Type	Description
ORDER_NUMBER	ORHNBR	CHAR(5)	Order number
CUSTOMER_NUMBER	CUSBR	CHAR(5)	Customer number
ORDER_DATE	ORHTE	DATE	Order date
ORDER_DELIVERY	ORHDLY	DATE	Order delivery date
ORDER_TOTAL	ORHTOT	DEC(11,2)	Order total
ORDER_SALESREP	SRNBR	CHAR(10)	Sales representative number

Table 2-3 shows the ORDERDTL table.

Table 2-3 ORDERDTL table

Field name	Alias	Type	Description
ORDER_NUMBER	ORHNBR	CHAR(5	Order number
PRODUCT_NUMBER	PRDNBR	CHAR(5)	Product number
ORDERDTL_QUANTITY	ORDQTY	DEC(5,0)	Order detail quantity
ORDERDTL_TOTAL	ORDTOT	DEC(9,2)	Order detail total

Table 2-4 shows the SALESREP table.

Table 2-4 SALESREP table

Field name	Alias	Type	Description
SALESREP_NUMBER	SRNBR	CHAR(10)	Sales representative number
SALESREP_NAME	SRNAM	CHAR(20)	Sales representative name
SALESREP_TELEPHONE	SRTEL	CHAR(15)	Sales representative telephone number
SALESREP_CITY	SRCTY	CHAR(20)	Sales representative city
SALESREP_MANAGER	SRMGR	CHAR(10)	Sales representative manager, who is also a sales representative

Table 2-5 shows the SALESCUS table.

Table 2-5 SALESCUS table

Field name	Alias	Type	Description
SALESREP_NUMBER	SRNBR	CHAR(10)	Sales representative number
CUSTOMER_NUMBER	CUSBR	CHAR(5)	Customer number
SALES_AMOUNT	SRAMT	DEC(11,2)	Sales representative total amount for this customer

Table 2-6 shows the SUPPLIER table.

Table 2-6 SUPPLIER table

Field name	Alias	Type	Description
SUPPLIER_NUMBER	SPLNBR	CHAR(5)	Supplier number
SUPPLIER_NAME	SPLNAM	CHAR(20)	Supplier name
SUPPLIER_TELEPHONE	SPLTEL	CHAR(15)	Supplier phone number
SUPPLIER FAX	SPLFAX	CHAR(15)	Supplier fax number
SUPPLIER ADDRESS	SPLADR	CHAR(20)	Supplier address
SUPPLIER_CITY	SPLCTY	CHAR(20)	Supplier city
SUPPLIER_ZIP	SPLZIP	CHAR(5)	Supplier ZIP code

Table 2-7 shows the STOCK table.

Table 2-7 STOCK table

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

Table 2-8 shows the STOCKPIC table.

Table 2-8 STOCKPIC table

Field name	Alias	Type	Description
PRODUCT_NUMBER	PRDNBR	CHAR(5)	Product number
PRODUCT_PICTURE	PRDPIC	BLOB	Product picture

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

Table 2-9 shows the SALES view.

Table 2-9 SALES view

Field name	Type	Description
YEAR	INTEGER	Order year
MONTH	INTEGER	Order month
SUPPLIER_NAME	CHAR(20)	Supplier name
SALES	DECIMAL(11,2)	Sales

Table 2-10 shows the TOTALSALES view.

Table 2-10 TOTALSALES view

Field name	Type	Description
YEAR	INTEGER	Order year
MONTH	INTEGER	Order month
SUPPLIER_NAME	CHAR(20)	Supplier name
TOTALSALES	DECIMAL(11,2)	Total sales

Table 2-11 shows the YEARSALES view.

Table 2-11 YEARSALES view

Field name	Type	Description
YEAR	INTEGER	Order year
SUPPLIER_NAME	CHAR(20)	Supplier name
TOTALSALES	DECIMAL(11,2)	Total sales

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 are defined.

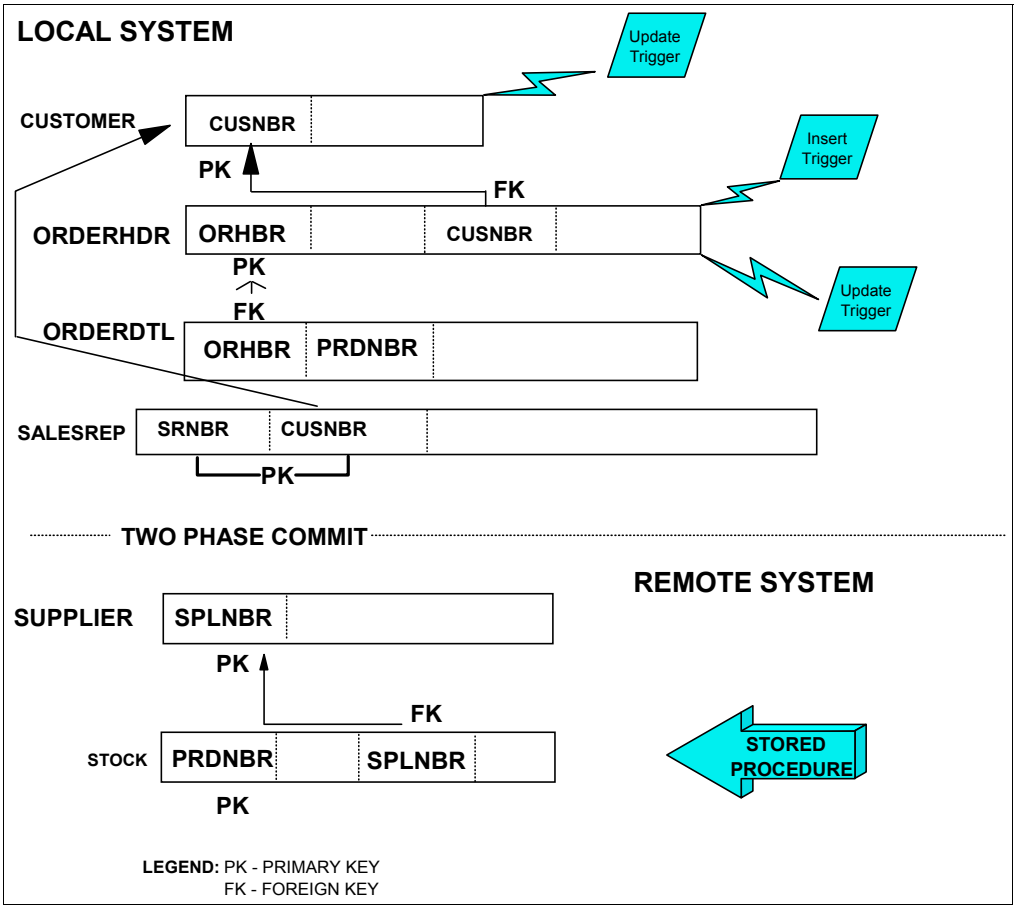


Figure 2-3 Order Entry application database structure

The main objective of presenting this application scenario with this specific database design is to show how the stored procedures and triggers that are provided with DB2 Universal Database for iSeries can be used and how they can work together in a single application. We analyze Figure 2-3 from each function's standpoint.

2.3.1 Stored procedures

Figure 2-3 shows a stored procedure that is 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 on page 16, we defined three trigger programs (two trigger programs in the ORDERHDR table and one trigger program 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 work with that customer. In addition, we want to track any attempts to violate the rule.
- When the order is completed and accepted by the customer, we want to print the related invoice.
- If the total amount of an order exceeds 90% 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, which is recognized by a customer number that starts with the digit 9, the credit limit is automatically increased by 30%.

Because we want these functions performed each time that an ORDERHDR insertion, an ORDERHDR update, or a CUSTOMER update occurs, we associate an Insert trigger and an Update 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 that is illustrated with the following examples:

- A casting UDF is required to convert from the DECIMAL(8) date representation to DATE. This hypothetical company has other existing systems that store several dates as 8-digit decimals in the *YYYYMMDD* format. To enhance the support of DB2 for i, a UDF was added to support the casting function. If the input parameter contains an invalid date, another requirement is to return a null value and signal a user-defined warning message with SQLSTATE `01HDI`.
- A table UDF is required to show a certain number of top-performing salespeople.

3

Stored procedures

This chapter explains how you can take advantage of stored procedures when you develop 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 following topics:

- Introduction
- Stored procedure types
- Registering stored procedures
- System catalog tables
- Procedure signature and procedure overloading
- Deleting or replacing stored procedures
- Authorization and adopted authority
- Returning result sets from stored procedures

3.1 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 useful in a distributed environment because they can 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 that 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 part 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 application purposes:

- Distributing the logic between a client and a server
- Performing a sequence of operations at a remote site
- Combining the results of query functions at a remote site
- Controlling access to database objects
- Performing non-database functions

We look at a typical example where stored procedures can be effective. A company runs its business on a server at the headquarters and on client systems at every branch office. A user at a branch office is working with an invoice clearance application, which must update three tables on the server:

- The invoice table is named INVOICE.
- The customer table is named CUSTOMER.
- The accounts receivable balance table is named ARBLNCE.

An invoice record is flagged with a “clearance” marker. Then, 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, also.

Figure 3-1 shows a distributed application for the invoice clearance process that was implemented without resorting to stored procedures. The client system must access the server database several times for every update event, sending and receiving data across communication lines for every request. In addition, all of the application logic is implemented at the client site.

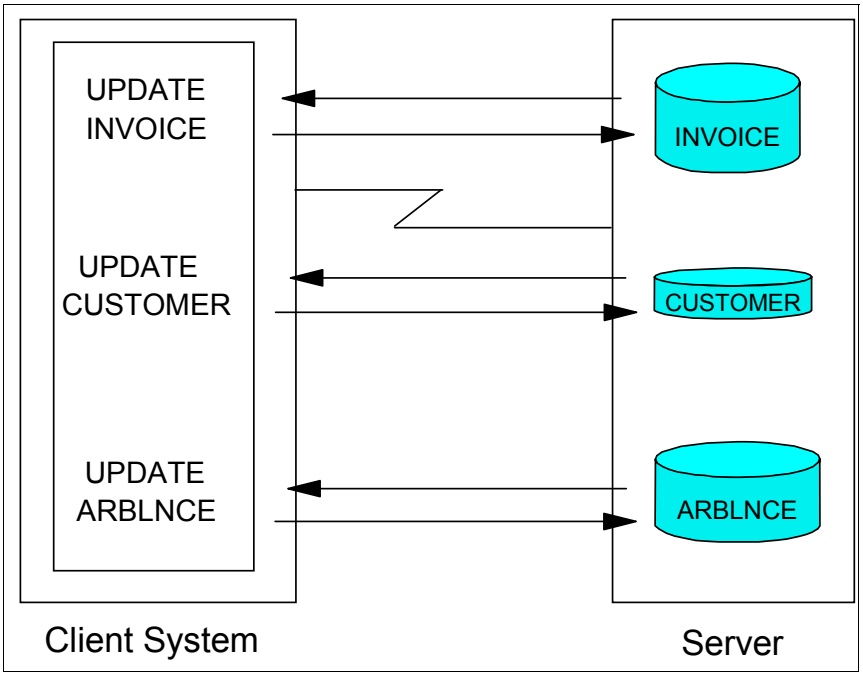


Figure 3-1 Distributed application without stored procedures

Figure 3-2 shows how we can take advantage of stored procedures to develop this application.

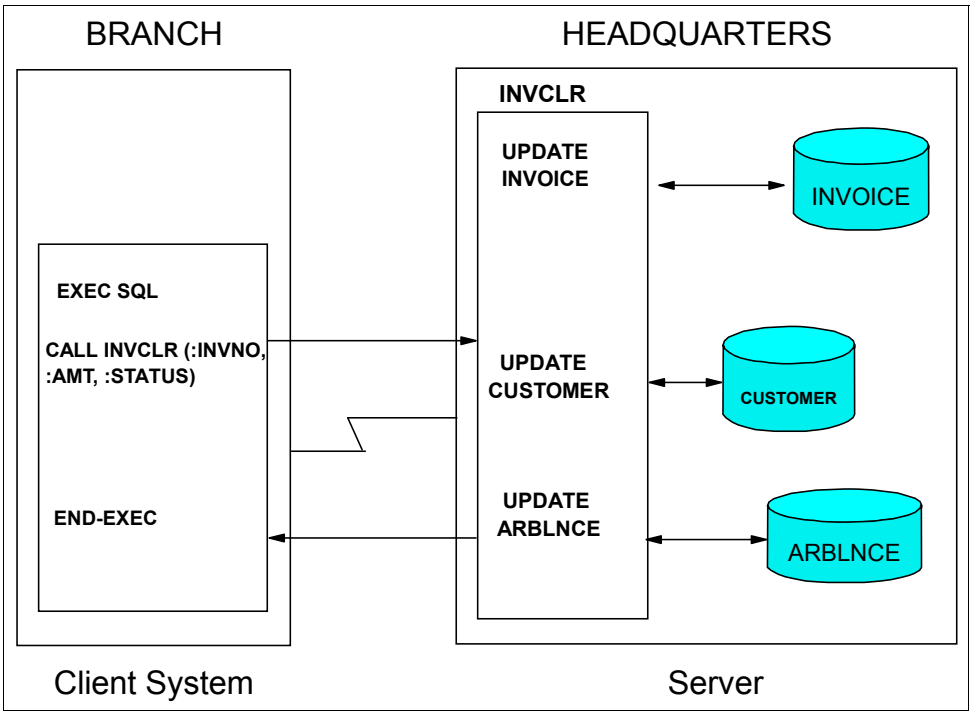


Figure 3-2 Distributed application with stored procedures

The same application functions can be performed 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 modularity facilitates application maintenance and improves code reusability.

It is useful to compare stored procedures to other tools and techniques for distributed application development, such as Distributed Relational Database Architecture (DRDA) SQL, distributed data management (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 because you can use them to easily split the application logic.
- The Submit Remote Command (**SBMRMTCMD**) command submits a control language (CL) command by using distributed data management (DDM) support to run on the target system. A user at a client system can use the **SBMRMTCMD** command to perform object management operations rather than running remote applications. You might 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 that were made by the server application on the server system cannot be committed or rolled back by the client application.

- *Triggers* are user-written programs that are 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 the productivity of application development. Stored procedures can return result sets, which makes them flexible and efficient in client/server environments.

  For more information about triggers, see Chapter 8, “External triggers” on page 217.

Table 3-1 summarizes the comparison of stored procedures, triggers, UDFs, and DRDA SQL for distributed applications.

Table 3-1 Comparing stored procedures, triggers, UDFs, and DRDA SQL

	Stored procedure	Trigger	UDFs	DRDA SQL
Invocation	Execution of SQL CALL statement	Database I/O	As a function in an SQL statement	Execution of a single remote SQL statement at a time
Environment	Distributed or non-distributed applications	Distributed or non-distributed applications	Distributed or non-distributed applications	Distributed Relational Database
Language	Any high-level language (HLL) program, including Java (might include SQL)	Any HLL program (might include SQL) - No Java support	Any HLL program, including Java (might include SQL)	Embedded SQL and Interactive SQL
Conversation method	Explicit 2-way parameter passing	Implicit system parameter passing	Explicit 2-way parameter passing	Application requester (AR) sends an SQL request, and application server (AS) sends an SQL request
Advantage	Performance improvement. Easy program invocation. Capable of returning result sets.	Automated consistent process. Performance improvement.	Functionality improvement. Extends object support.	Easier programming. Common SQL interface to other IBM platforms and platforms that are not IBM.

3.2 Stored procedure types

Stored procedures can be divided into two categories:

- SQL stored procedures
- External stored procedures

3.2.1 SQL stored procedures

SQL stored procedures are written in the SQL language, which makes it easier to port stored procedures from other database management systems (DBMS) to the IBM i server and from the IBM i server to other DBMS. Implementation of the SQL stored procedures is based on procedural SQL that is standardized in SQL99. For more details about SQL stored procedures, see *SQL Procedures, Triggers, and Functions on DB2 for i*, SG24-8326.

3.2.2 External stored procedure

An *external stored procedure* is written by the user in one of the programming languages on the IBM i server. You can compile the host language programs to create *PGM objects or a 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 that is registered in the following example returns the name of the supplier with the highest sales in a specific month and year. The procedure is implemented in IBM Integrated Language Environment® (ILE) RPG with embedded SQL:

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

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

```
c/EXEC SQL
c+ CALL HSALE(:PARM1, :PARM2, :PARM3, :PARM4)
c/END-EXEC
```

An external stored procedure might contain no SQL statements. For example, you can create a stored procedure that uses the native interface to access the DB2 Universal Database for iSeries data.

The IBM i server implementation of external stored procedures is described in detail in Chapter 4, “External stored procedures” on page 39.

Java stored procedures

Java stored procedures were first introduced in the IBM i server, starting with V4R5, as a particular case of external stored procedures that were limited to not being able to return result sets. Starting with V5R1, the support of result sets was added to Java stored procedures. Growing interest in Java and its portability across platforms makes Java stored procedures an interesting option to consider. The IBM i server implementation of Java stored procedures is described in detail in Chapter 5, “Java stored procedures” on page 91.

3.3 Registering stored procedures

Before a stored procedure can be called by a client program, it must be registered with the database by using the DECLARE PROCEDURE or the CREATE PROCEDURE statement. The stored procedure can also be defined by using either of these statements. The CREATE PROCEDURE statement differs from the DECLARE PROCEDURE statement because 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 that is running on the local or the remote system.

Because the information about the stored procedure is stored in the system catalog tables, the CREATE PROCEDURE needs to be performed only one time 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 description of the CREATE PROCEDURE, refer to *SQL Reference*, SC41-5612.

3.3.1 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 that supports the SQL CALL statement.

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

SPECIFIC specific-name

DB2 for i identifies each stored procedure with a specific name that, combined with the specific schema, must be unique in the system. This requirement 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 for i 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 DB2 generating 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

You can use these options to set limits about what the stored procedure can use. The options are described in Table 3-2.

Table 3-2 SQL statements in stored procedures

Attribute	Description
CONTAINS SQL	The stored procedure contains SQL. It can contain the following information only: ► Non-executable statements (such as DECLARE statements) ► CALL statements to procedures with the NO SQL or CONTAINS SQL attribute ► FREE LOCATOR ► SET RESULT SET ► SET assignment and VALUES INTO if only variables or constants are referenced ► COMMIT, ROLLBACK, or SET TRANSACTION ► CONNECT, DISCONNECT, RELEASE, or SET CONNECTION
NO SQL	The stored procedure does not contain SQL statements.
READS SQL DATA	The stored procedure possibly reads data by using SQL. It can contain SQL statements except for the following statements: ► COMMIT, ROLLBACK, or SET TRANSACTION ► CONNECT, DISCONNECT, RELEASE, or SET CONNECTION ► DELETE, INSERT, or UPDATE ► ALTER TABLE, COMMENT ON, any CREATE, DROP, GRANT, LABEL ON, RENAME, or REVOKE statement
MODIFIES SQL DATA	The stored procedure possibly modifies data by using SQL. It can contain SQL statements except for the following statements: ► COMMIT, ROLLBACK, or SET TRANSACTION ► CONNECT, DISCONNECT, RELEASE, or SET CONNECTION

SET OPTION

The SET OPTION clause gives you more control over how the SQL stored procedures are created. You can control the following options with SET OPTION:

- OUTPUT: Specifies whether a listing file is required. If it is set to *PRINT, it generates two listing spool files: one file for the intermediate C code and another file for the corresponding precompiled C code.
- DBGVIEW: Controls the level of debug information that is contained in the program object. The default is none, but you can set *STMT, *LIST, or *SOURCE values. *STMT allows the compiled module object to be debugged by using program statement numbers and symbolic identifiers. *LIST generates the listing view for debugging the compiled module object. You can use *SOURCE to 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. The 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 the use of several of the options in a stored procedure creation:

```
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;
```

OUTPUT and DBGVIEW are of great value when you debug an application. For more information about these values, see the online help for the Run SQL Statement (**RUNSQLSTM**) command.

3.3.2 DECLARE PROCEDURE

DECLARE PROCEDURE is a kind of temporal procedure definition in which the stored procedure is declared in a program. Originally, you were only able to embed it in an application program as a static SQL statement. However, starting in V5R1 (and we advise that you look for the latest iSeries/IBM i Access Open Database Connectivity (ODBC) driver available), DECLARE PROCEDURE 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 certain cases, represents a performance improvement.

Several differences exist in the syntax of DECLARE PROCEDURE compared with CREATE PROCEDURE, as you can see in Example 3-1.

Example 3-1 DECLARE PROCEDURE

```
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 that is highlighted in bold) compared to the equivalent CREATE PROCEDURE GETSUPPLIERSDB2GENERAL.

The C++ program sample in Example 3-2 shows how to use DECLARE PROCEDURE from an ODBC program.

Example 3-2 DECLARE PROCEDURE from an ODBC program

```
#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 PROCEDURE ()");                1
  strcat((char *)SQLStmt, "PARAMETER STYLE DB2GENERAL LANGUAGE JAVA ");             1
  strcat((char *)SQLStmt, "EXTERNAL NAME 'ClassName!MethodName'");                  1
  cout << "Procedure to prepare:" << endl;
  cout << SQLStmt << endl;
  rc = SQLPrepare(DclStmtHandle, SQLStmt, SQL_NTS);                                 2
  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);                                3
    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 3-2 on page 28:

1. We assemble the DECLARE PROCEDURE statement. The DECLARE PROCEDURE statement syntax is different from the CREATE PROCEDURE syntax.
2. The statement that was assembled in 1 is then prepared. *It is not executed, only prepared.*
3. Now, we are ready to create and use statements that call to the stored procedure that is declared in 2.

Keep in mind that DECLARE PROCEDURE is a nonstandard SQL command that will not be enhanced. If you are interested in portability, you must avoid the use of the DECLARE PROCEDURE statement.

3.4 System catalog tables

The database manager maintains a set of tables that contain information about the data in each relational database. These tables are collectively known as the *catalog*. The *catalog tables* store information about every routine (procedure or function) that is registered with the database:

- Tables
- User-defined functions
- Distinct types
- Parameters
- Procedures
- Packages
- Views
- Indexes
- Aliases
- Constraints
- Triggers
- Languages that are supported by DB2 for i

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

3.4.1 SYSROUTINES catalog

All stored procedures that are registered with the CREATE PROCEDURE statement or that use System i Navigator are stored in the SYSROUTINES catalog. For a detailed description of the catalog views, see *SQL Reference*, SC41-5612.

Note: The SYSROUTINES catalog contains information about both stored procedures and user-defined functions (UDFs). You can 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:

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

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

```
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 3-3.

select Specific_schema,routine_name,routine_type,routine_body,parameter_style from qsys2/sysroutines where routine_schema = 'SPROCLIB' and ro

	SPECIFIC_SCHEMA	ROUTINE_NAME	ROUTINE_TYPE	ROUTINE_BODY	PARAMETER_STYLE
1	SPROCLIB	HSALES	PROCEDURE	EXTERNAL	GENERAL
2	SPROCLIB	RANKTOT1	PROCEDURE	EXTERNAL	GENERAL
3	SPROCLIB	RANKTOT2	PROCEDURE	EXTERNAL	SQL
4	SPROCLIB	RANKTOT3	PROCEDURE	EXTERNAL	SQL
5	SPROCLIB	RANKTOT0	PROCEDURE	EXTERNAL	GENERAL
6	SPROCLIB	RANKTOT3	PROCEDURE	EXTERNAL	SQL
7	SPROCLIB	DELPGM2	PROCEDURE	EXTERNAL	NULLS
8	SPROCLIB	DELPGM3	PROCEDURE	EXTERNAL	SQL

Figure 3-3 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:

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

3.4.2 SYSPARMS catalog

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

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

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

3.5 Procedure signature and procedure overloading

DB2 for i supports the concept of *procedure overloading*. Procedure overloading 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 exists, the system registers a new procedure instance rather than overlaying the existing one. For more information, see 4.6, “Moving into production (save and restore)” on page 73.

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 )
```

For a detailed description of UDFs, see Chapter 10, “User-defined functions” on page 313.

3.6 Deleting or replacing stored procedures

When you create a procedure, its signature must be unique to register the procedure in the catalog. As described in 3.5, “Procedure signature and procedure overloading” on page 33, 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: 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.

3.6.1 Using a command line to drop a procedure

Several ways are available to drop a stored procedure from the IBM i server:

- In the traditional “green screen” environment, start the interactive SQL (ISQL) session with the command:

  STRSQL NAMING(*SQL)

  At the ISQL prompt, type the following SQL statement:

  ```
  DROP PROCEDURE library.procedure-name
  ```

  Figure 3-4 shows the message that is issued after the procedure is successfully deleted.

```
                              Enter SQL Statements

Type SQL statement, press Enter.
   > DROP PROCEDURE ordapplib.caseproc
     DROP PROCEDURE statement complete.
===>

F3=Exit    F4=Prompt    F6=Insert line    F9=Retrieve    F10=Copy line
F12=Cancel              F13=Services      F24=More keys
```

Figure 3-4 Dropping a procedure in an interactive SQL session

- In the System i Navigator environment, in the right panel of the main System i Navigator window, right-click the procedure that you want to drop, and select the **Delete** option, as shown in Figure 3-5. A window opens that shows the stored procedure object that is selected for deletion. Confirm that this procedure is the procedure that you want to delete, and click **Delete**.

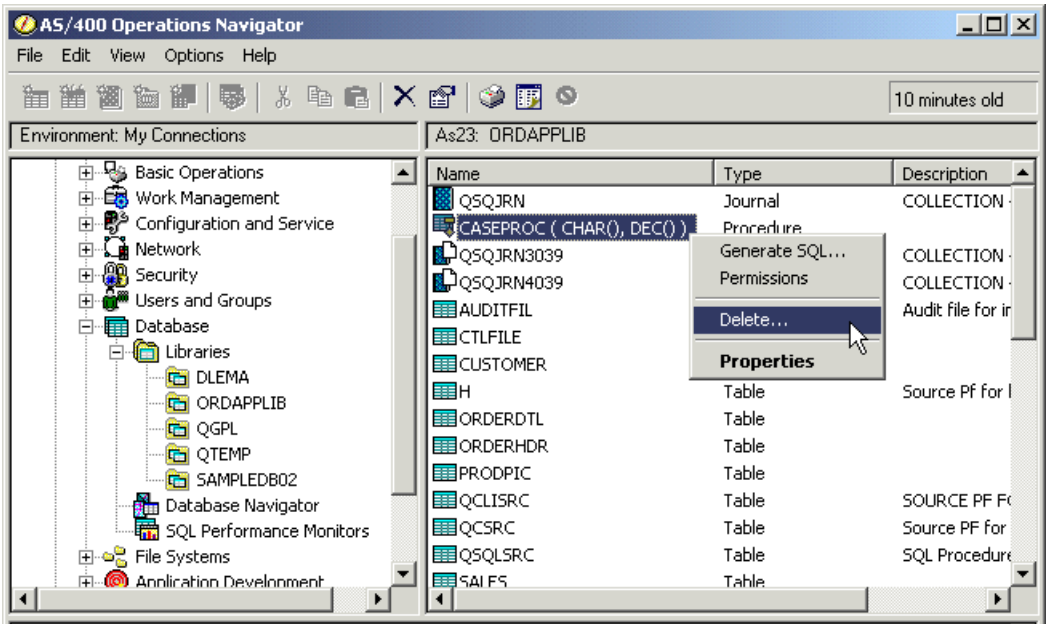


Figure 3-5 Deleting a stored procedure

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

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

3.6.2 Dropping overloaded procedures

Dropping overloaded procedures can be tricky. Because the procedure name is overloaded, it is not sufficient to supply its name on the DROP PROCEDURE statement. Two methods can be used to correctly resolve the overloaded name.

Assume 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 following methods:

- Specify the specific procedure name:

  ```
  drop specific procedure spintchar;
  ```

- Include the parameter types on the DROP PROCEDURE statement:

  ```
  drop procedure myStorproc( int, char );
  ```

3.7 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 that corresponds to that stored procedure, depending on how it was defined in the USRPRF attribute for that program object.

When USRPRF is set to *USER, 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, a mechanism is available that is 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. The adopted authorities mechanism is effective only if USRPRF is set to *OWNER.

Table 3-3 summarizes the effects of authorization and adopted authorities.

Table 3-3 Description of authorization and adopted authorities

Authorization (USRPRF)	Adopted authorities (USEADPAUT)	Description
*OWNER	*YES	The program uses authorization from both the user and the program owner profiles. In addition, it inherits the authorities of the caller program.
	*NO	The program uses authorization from both the user and the program owner profiles. But, it does not inherit the authorities of the calling program.
*USER	*YES or *NO	The program only uses the user profile authorities.

Authorities and adopted authorities provide mechanisms for improving security. These mechanisms might include giving access to sensitive objects to one user or maybe a controlled set of users, and then making those users the owners of the programs (including stored procedure programs, except for Java stored procedures) that access and modify them. Then, the other 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 risky). Instead, you grant them execution authority on the clerk front-end banking application.

3.8 Returning result sets from stored procedures

An SQL stored procedure can call another procedure, which in turn calls another procedure in a chain, which 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, which is called the *returnability attribute*.

Starting in V5R3, two new syntaxes were added to DB2 for i SQL. These syntaxes provide flexibility in designing whether the result set of a stored procedure, which is called in nested procedures, is returned to the *immediate calling procedure* or to the calling procedure that is at the *beginning of the calling chain,* as shown in Figure 3-6.

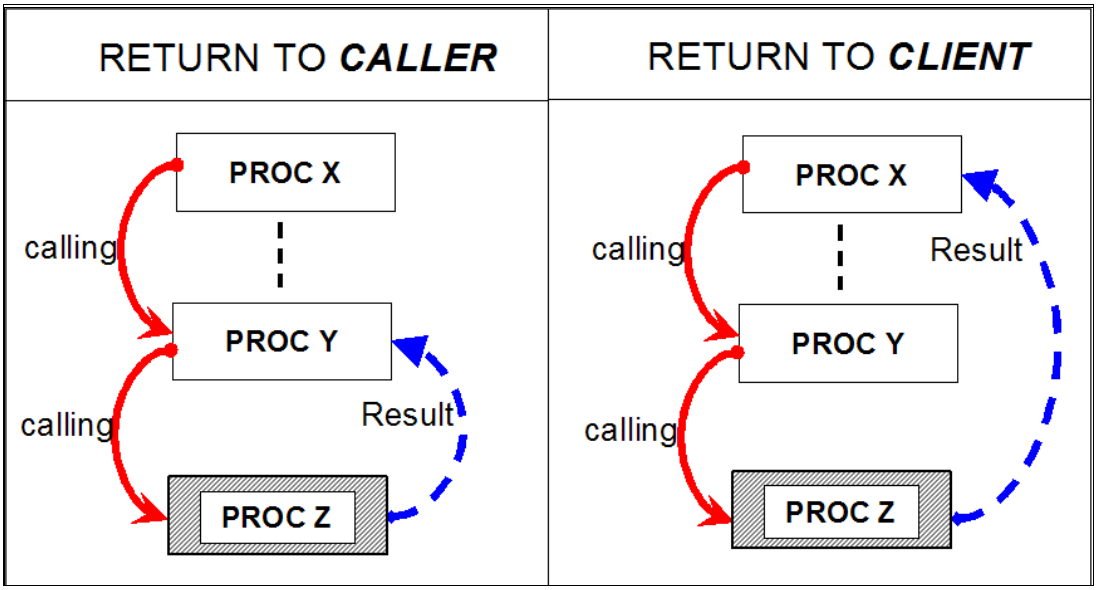


Figure 3-6 Returning result sets to a caller versus to a client

You can return result sets to a caller or client 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 of 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.

Important: You must install the latest Database Group program temporary fix (PTF).

4

External stored procedures

Stored procedures can be written in two ways on DB2 for i. One approach is described as *SQL stored procedures*. This approach is based on procedural extensions to the SQL language. This approach, which is highly used by other database management system (DBMS) providers, is documented in *SQL Procedures, Triggers, and Functions on DB2 for i,* SG24-8326. The other approach, which is described as *external stored procedures,* is based on high-level languages (HLLs) that you are familiar with, such as C, CL, RPG, and COBOL.

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

This chapter describes external stored procedures that are 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 in relation to the different parameter styling that is supported by DB2 for i and how to invoke external stored procedures and handle errors.

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 described in Chapter 5, “Java stored procedures” on page 91.

All of the benefits of the stored procedures that are described in Chapter 3, “Stored procedures” on page 19 also apply to external stored procedures.

This chapter describes the following topics:

- Registering external stored procedures
- Parameter styles in external stored procedures
- Coding external stored procedures
- Returning result sets from external procedures
- CLI client program that calls a procedure that returns multiple result sets
- Moving into production (save and restore)
- The Order Entry application: Stored procedure examples
- External stored procedure that uses a service program
- RPG IV example for an external stored procedure

4.1 Registering external stored procedures

Before you can use an external stored procedure, it must be registered within the database. You can use the CREATE PROCEDURE statement or System i 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 described in 3.4, “System catalog tables” on page 31.

When you register an external stored procedure, you must 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 describes the different parameter passing styles in 4.2, “Parameter styles in external stored procedures” on page 45. 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 describes the different coding techniques for result sets in 4.4, “Returning result sets from external procedures” on page 60.

The external program that is executed when the external stored procedure is called by the CALL statement needs to be a *PGM object that is compiled with the Activation Group parameter *CALLER. The external program can contain host language statements and SQL statements.

Important: A service program cannot be registered as an external stored procedure.

Note: Certain IBM i interfaces, such as STRSQL and System i Navigator, allow programs to be called without registering the program first. However, we recommend that you always register the programs as external stored procedures.

An external stored procedure can be called by an application program that runs on the same IBM i server, where the stored procedure resides. Or, it can be called across the network by a client program. The client program can run on a workstation and communicate with the server through programming interfaces, such as Open Database Connectivity (ODBC), ActiveX Data Object (ADO), Java Database Connectivity (JDBC), and Structured Query Language for Java (SQLJ). It can also run on another server machine and communicate through Distributed Relational Database Architecture (DRDA).

Examples of client programs that can call the external stored procedure are described in 4.5, “CLI client program that calls a procedure that returns multiple result sets” on page 68.

4.1.1 Registering an external procedure with System i Navigator

For example, we describe 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 specific year and month. The parameter passing style that is used is GENERAL. These input and output parameters are based on the columns of the TOTALSALE view. For the detailed structure of this view, see 2.2, “Order Entry database overview” on page 11.

This section shows how to create an external stored procedure by using the System i Navigator New External Procedure window. The required steps are listed:

1. In System i Navigator, expand **Databases** and the ***database folder*** for which the external procedure will be registered. Expand the **Libraries** object. You see all of 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 4-1.

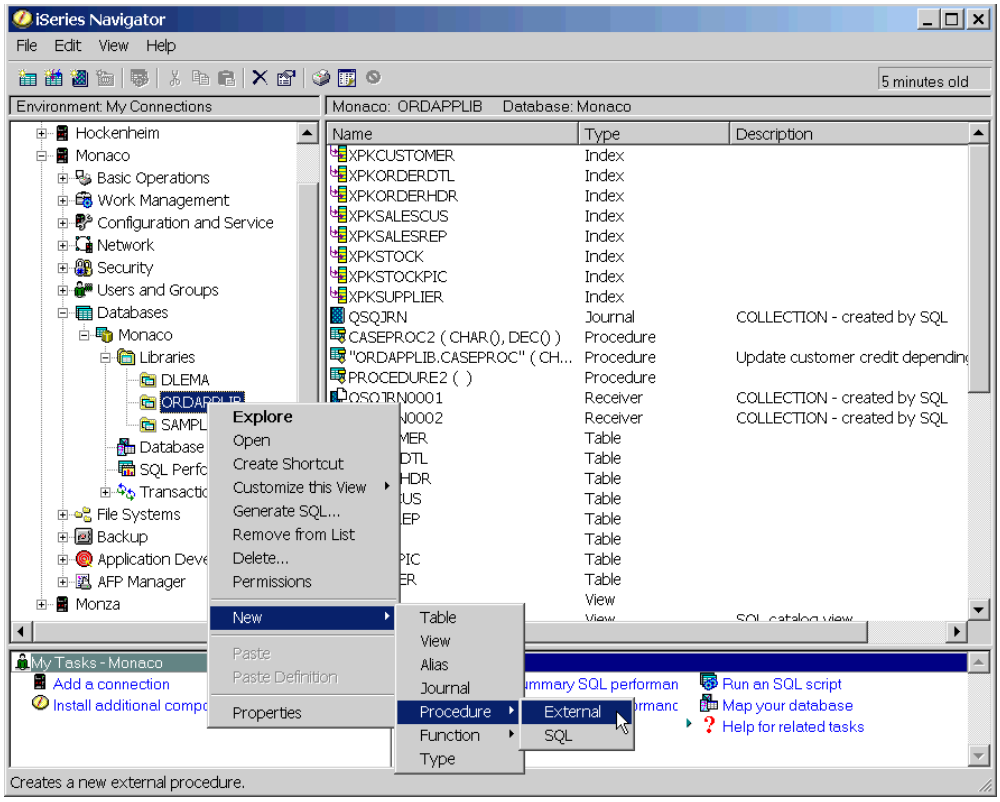


Figure 4-1 Creating an external stored procedure by using the Create procedure window

2. The New External Procedure window (Figure 4-2) opens in which you perform the following actions.

   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 can be defined at this time. For example, you can define whether an automatic commit must be performed when the stored procedure returns control to the callers, whether the stored procedure must run at an inner savepoint level, and the maximum number of result sets that the stored procedure will return.

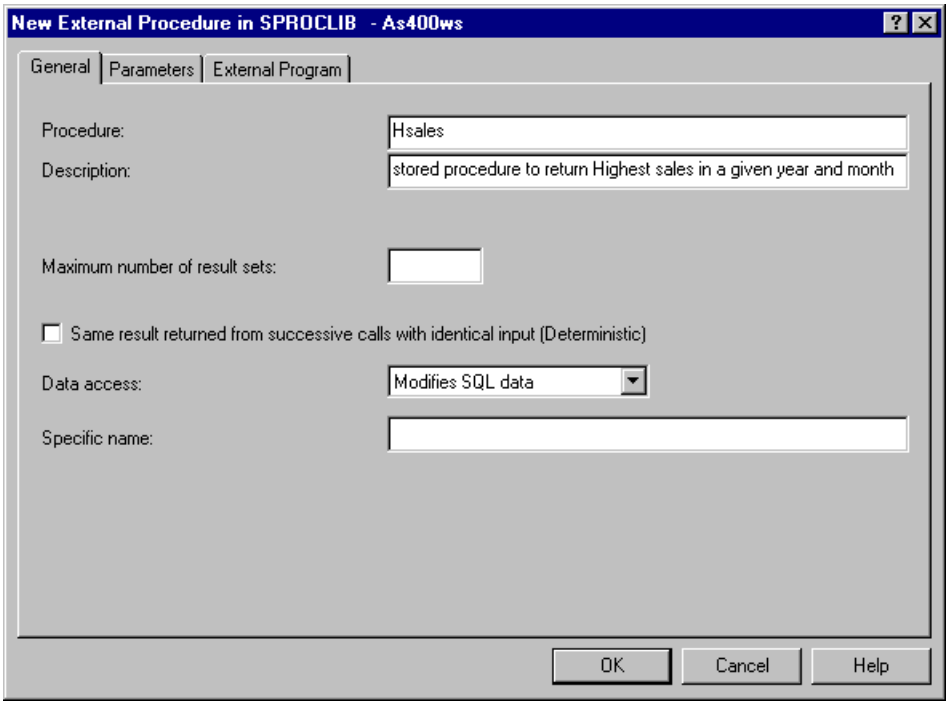


Figure 4-2 New external procedure

3. Click the **Parameters** tab and follow these steps:
   a. Click **Insert**. Type the name of the parameter.
   b. Choose the data type of the parameter from the pull-down list box. Enter the length of the parameter, if required. Select the input/output type of the parameter as shown in Figure 4-3.
   c. Choose the parameter style. Click **Simple, no null values allowed**. If you do not choose the parameter style, the default value is SQL.

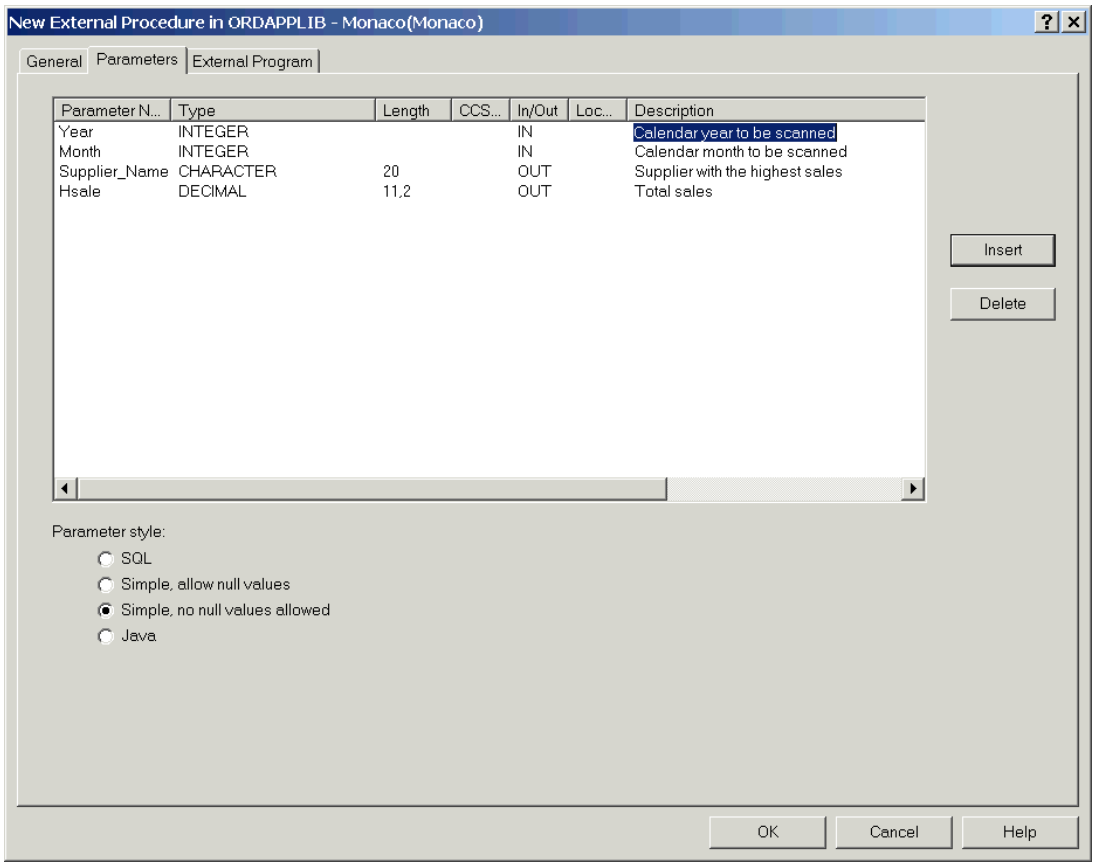


Figure 4-3 Defining input/output parameters

4. Click the **External Program** tab and follow these steps:

   a. Type the name of the external program to execute when this external stored procedure is called by 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 4-4. If you leave the language field empty, the system tries to guess the implementation language. The default value for this field is Integrated Language Environment (ILE) C.

   b. Click **OK** to register the stored procedure.

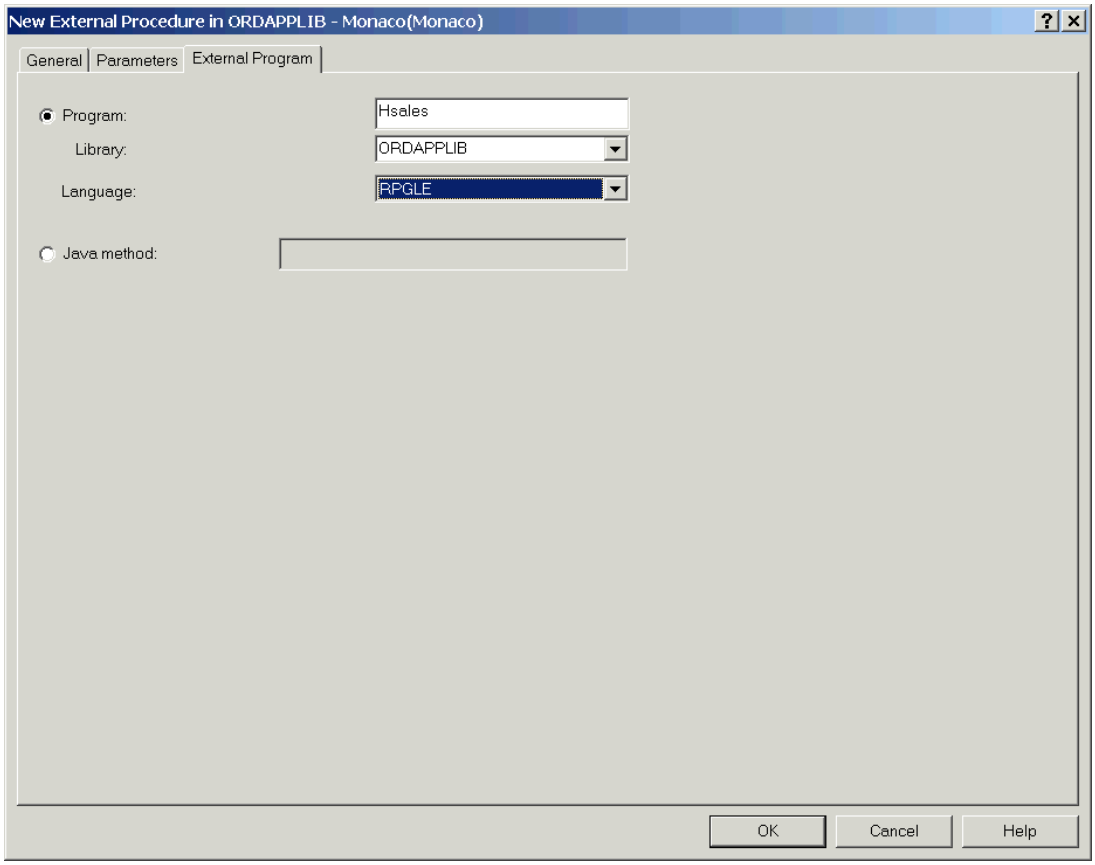


Figure 4-4 External program name, library, and language

The corresponding SQL CREATE PROCEDURE statement is shown:

```
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 the successful completion of the window, refresh the contents of the ORDAPPLIB library by clicking the **Refresh** icon in the toolbar. You now see the HSALES stored procedure icon in the list of the objects, as shown in Figure 4-5.

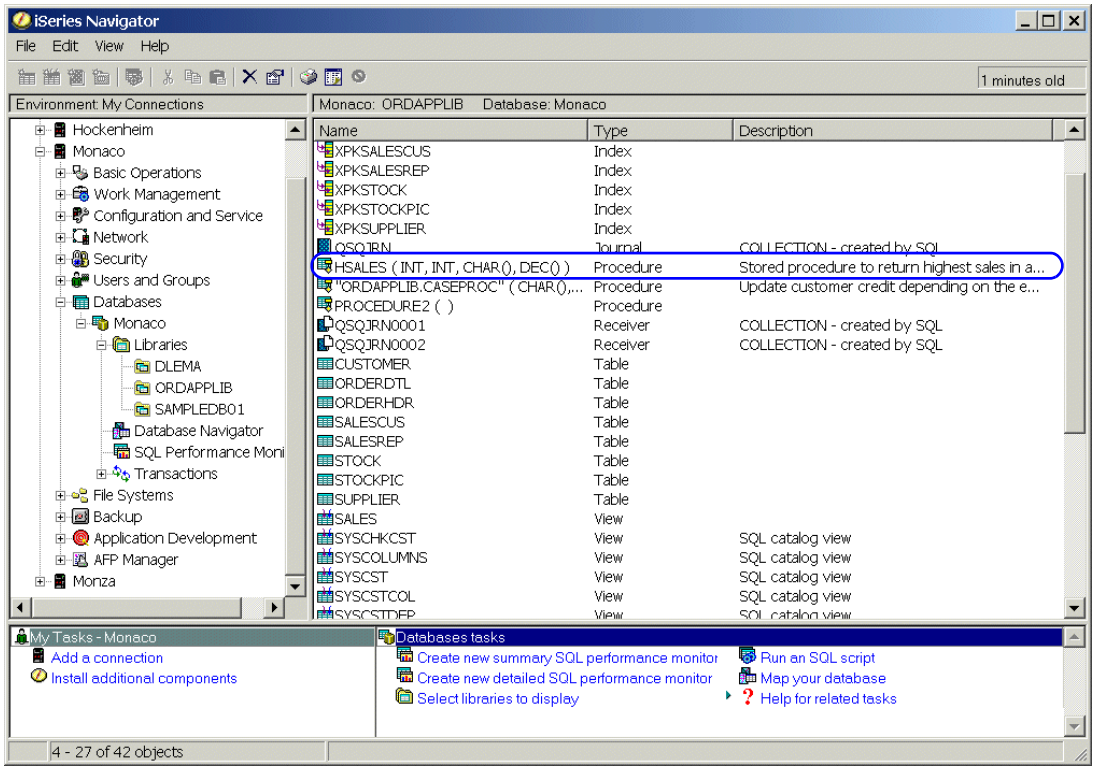


Figure 4-5 A new stored procedure

4.2 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 parameters that are specified on the parameter list. The number and type of additional parameters that are 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

Before 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 explains the number and type of parameters that are passed with each parameter style. Later in this chapter, we provide examples for each of these parameter styles.

4.2.1 SQL parameter style

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

```
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 that are 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 you use 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. Every output parameter has a corresponding null indicator that is passed back to the calling program. In the calling program, you can check whether a null value was returned in the output parameter.
- SQLSTATE: The output parameter, which is 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 that are 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.
  - `01H`*xx*: Warning. The trailing *xx* value is any two digits or uppercase letters. It results in SQLCODE 462 from SQL.
  - `38`*yxx*: Error condition *y* can be any letter or number. The next two characters *xx* are any two digits or uppercase letters to indicate the error. It results in SQLCODE -443 from SQL.
- Procedure name: A fully qualified procedure name. This parameter is an input parameter, which is defined as VARCHAR(517).
- Specific name: The specific name of the function. This parameter is an input parameter, which is 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 or a referential constraint violation, it is set to the first 70 characters of the system message. This parameter is an output parameter that is 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 format changed and the message is a null terminated string.

4.2.2 DB2SQL parameter style

The DB2SQL style is a superset 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 that contains the following fields:

- Relational database name
- Authorization ID
- Coded character set identifier (CCSID)
- Version and 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 that are passed, see the include sqludf in the appropriate source file. For example, for C, sqludf is in `QSYSINC/H`.

4.2.3 GENERAL WITH NULLS parameter style

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

```
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 you use the input parameter in the external stored procedure, check the null indicator. If the corresponding argument contains a NULL value, take corrective action.

4.2.4 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 consist 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 that are allowed in the CREATE PROCEDURE statement is limited by the programming language that is used to implement the stored procedure. For a procedure that was created with the SQL parameter style, the additional implicit parameters are included in the calculation.

4.3 Coding external stored procedures

An external stored procedure does not differ significantly from any other high-level language program that you already write. The difference is that it is registered to the DB2 Universal Database for iSeries, as described in 4.1, “Registering external stored procedures” on page 40, and the way that it receives and returns results both as parameters and as result sets.

A *result set* is an open cursor that is returned by a stored procedure. Result sets are the mechanism that stored procedures use for returning multiple row results. One stored procedure can return none, one, or multiple result sets, but at any time, a calling program can have only 100 procedures with result sets that are waiting to be fetched.

This section explains the differences in stored procedures code in relation to the parameter style that is used and the details about returning result sets and error handling.

4.3.1 Coding for SQL parameter style

This section looks at examples of how to code external stored procedures with the SQL parameter style. It also demonstrates how the parameters that are passed by DB2 Universal Database for iSeries to the external stored procedure can be used within the procedure.

In the Order Entry database that is used throughout this book, we define the ORDERHDR and the CUSTOMER tables. A referential constraint is 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.

Assume that you want to delete a particular customer from the CUSTOMER table. To accomplish this task, you implement an external stored procedure, which is called CusNumDel, by 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 ORDEHDR 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 must be returned to the calling program so that it is aware that the customer record deletion failed.

We examine the CREATE PROCEDURE statement for the CusNumDel external stored procedure in Example 4-1. The numbered sections are explained further in the following list.

Example 4-1 CREATE PROCEDURE statement for the CusNumDel external stored procedure

```
CREATE PROCEDURE PROCLIB/CUSNUMDEL(
   IN CUSNBR CHAR(5))                1
SPECIFIC  CUSNUMDEL                  2
LANGUAGE  C                          3
EXTERNAL  NAME SPROCLIB/CUSNUMDEL    4
MODIFIES  SQL DATA                   5
PARAMETER STYLE SQL                  6
```

CREATE PROCEDURE statement explanation

The following explanation refers to Example 4-1 on page 48:

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 a procedure exists 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 situation, the stored procedure name is *overloaded*.

2 The SPECIFIC NAME clause of the CREATE PROCEDURE statement. Every procedure that was created on the IBM i server needs a specific name. This name must be unique in the specific library. This clause is optional. If you do not specify a certain 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 exists, the system generates a unique name.

3 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 Integrated Language Environment (ILE) C. This information helps the database to pass parameters to the external stored procedure in the format that is required by the programming language. External stored procedures can be written in any of the following languages:

- CL
- COBOL
- COBOLLE
- FORTRAN
- Java
- PL/I
- 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 that is specified in the EXTERNAL NAME clause and set the clause. If the program object does not exist, or if the attribute is not present, the language is defaulted to ILE C.

4 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 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 clause is optional. 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 that it is a valid system name that is not longer than 10 characters. Two different stored procedures can point to the same external program name. An external program must be a *PGM object. It cannot be an ILE service program.

5 The NO/READS/MODIFIES/CONTAINS SQL DATA clause of the CREATE PROCEDURE statement. Here, you specify the kind of SQL statements that the procedure will execute. For a detailed description of the valid SQL statements for a certain clause, see *SQL Reference,* SC41-5612.

6 The PARAMETER STYLE clause of the CREATE PROCEDURE statement. For external stored procedures, this clause 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 that are defined in the CREATE PROCEDURE statement, based on the parameter style that is specified.

Now, we examine the external program CUSNUMDEL that is referred to in the CREATE PROCEDURE statement. We describe the parameters that DB2 Universal Database for iSeries sends to the program and how the program uses 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 by using the SQLSTATE output parameter. The SQLSTATE is set to "38IRC" when a delete rule is violated. When SQLSTATE is set by the external program, the diagnostic message is also returned to the calling program.

Example 4-2 shows how the external stored procedure with the SQL parameter style is coded. The numbered areas are further explained in the following list.

Example 4-2 Coding of an external procedure with the SQL parameter style

```
#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 */  1
  v_custno_ind=*(short int*)argv[2]; /* customer number null indicator */     2
  procname_var=*(struct procname*)argv[4]; /* process name */                  4
```

```
  specname_var=*(struct specname*)argv[5]; /* specific name of the SP. */     5
  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);                                              3
    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));     3
  } 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, which is referred to in the CREATE PROCEDURE statement as shown:

```
EXTERNAL NAME SPROCLIB.CUSNUMDEL
```

The external program that is coded in any host language needs to be complied with the Activation Group parameter as *CALLER. The following explanations refer to the numbers that are shown in Example 4-2 on page 50:

1 The CUSNUMDEL procedure accepts an input parameter of type CHAR(5), which is the customer number to delete 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 that contains the fully qualified name of the procedure.

4 Parameter with the specific name of the procedure that was called. The specific name can be used when the procedure is overloaded. Even if the procedure names are the same, the specific names must 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 "38*yxx*". 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 that we are trying to delete from the CUSTOMER table has a dependent row in the ORDEHDR table, the SQL DELETE will fail and generate SQLCODE=-532 and SQLSTATE=23001. This SQLSTATE cannot be directly returned to the calling program in the SQLSTATE output parameter because it will be treated by the database runtime as an invalid state. The database will 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 the user-defined value that signals the error condition. In the following code snippet, you can see that the SQLSTATE output parameter is set to "38IRC". When the SQLSTATE is set to a value that matches the pattern "38*yxx*", the diagnostic message text is also returned to the calling program 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 *)argv[6],(void *)&outmsgtxt_var,sizeof(outmsgtxt_var.data));
 } else {
   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));
 }
```

For a detailed description of error handling, see Chapter 5, “Java stored procedures” on page 91.

The CUSNUMDEL program was created as a *PGM object. In this case, CUSNUMDEL is a program that was 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 so that we can specify the activation group parameter as *CALLER.

The following control language (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 4-3. 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 "38*yxx*", the corresponding diagnostic message text can be retrieved from the sqlca.errmc field.

Example 4-3 Calling the external stored procedure with the SQL parameter style

```
#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);
  printmsg:
  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 that is shown in bold, we do not specify extra parameters for the SQL parameter style. They are implicitly passed to the stored procedure by the SQL runtime.

4.3.2 Coding the DB2SQL parameter style

This section looks at examples of how to code the external stored procedure with the DB2SQL parameter style when DBINFO is specified. When the DB2SQL parameter style is used in combination with NO DBINFO, it has the same effect as coding with the SQL parameter style.

Following the same example that is shown in 4.3.1, “Coding for SQL parameter style” on page 48, assume that we are required to modify the CusNumDel external stored procedure to record a trace row in table DELCTL with the notification to the user, time stamp, and deleted customer number.

We 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: For 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 that contains the fields that are described in Table 4-1.

Table 4-1 DBINFO fields

Field	Data type	Description
Relational database	VARCHAR(128)	The name of the current server (as it is displayed in WRKRDBDIRE).
Authorization ID	VARCHAR(128)	The runtime authorization ID.
Coded character set identifier (CCSID Information)	INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. For more details, see *SQL Reference*, SC41-5612.
Target column		Not applicable for a call to a procedure.
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server’s platform type.

Now, we examine the external program CUSNUMDEL2 that is referred to in the CREATE PROCEDURE statement (see Example 4-4). This program is similar to the CUSTNUMDEL that was exposed earlier. However, it differs in that it uses the Authorization ID that was received as part of the DBINFO data structure and adds a row in the DELCTL table.

Example 4-4 External program CUSNUMDEL2

```
#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> 1

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;  2
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 procname { short int length;
                    unsigned char data[139];
                  } procname_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 const  "};
  /* 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;  3

  strncpy(inputvar,argv[1],5);
```

```
  inpindicator=*(short int*)argv[2];
  procname_var=*(struct procname*)argv[4];
  specname_var=*(struct specname*)argv[5];

  /* retrieving DBINFO                                          */
  dbinfo = *(struct sqludf_dbinfo*)argv[7];  4

  /* retrieving the user id from DBINFO data structure          */
  user_id.useridlen = dbinfo.authidlen;      5
  strncpy(user_id.useridtxt, dbinfo.authid, dbinfo.authidlen);  5

  EXEC SQL
    DELETE FROM ORDAPPLIB.CUSTOMER WHERE CUSNBR=:inputvar;

  /* inserting control row in DELCTL                            */
  EXEC SQL
    INSERT INTO ORDAPPLIB.DELCTL (CUSNBR, LAST_MOD_USR, LAST_MOD_TS)
    VALUES (:inputvar, :user_id, CURRENT TIMESTAMP);  6

  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 *)argv[6],
           (void *)&outmsgtxt_var,
           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 that is named CUSNUMDEL2 (Example 4-4 on page 55) is similar to CUSNUMDEL, but it uses the DBINFO data structure to get the user ID that invokes this stored procedure and tracks it on the DELCTL table. We highlight the differences:

1. The CUSNUMDEL2 uses the DBINFO data structure that is common to other DB2 Universal Database platforms. This data structure is defined in the `sqludf.h` include file. For other programming languages, corresponding include files are in the `QSYSINC` library. For example, member SQLUDF in QSYSINC/QRPGLESRC can be copied into ILE RPG programs.
2. VARCHAR data structure to store user ID.
3. Here, we defined a variable that is called DBINFO that is based on the structure db2udf_dbinfo. This structure is predefined in `sqludf.h`.
4. DBINFO is passed to the stored procedure program after the SQL parameters that were introduced in 4.3.1, “Coding for SQL parameter style” on page 48.
5. The caller user ID is in DBINFO in VARCHAR style. For details about the information that is available in DBINFO, see the `sqludf.h` include file.
6. This line inserts a new line into the DELCTL table.

Calling an external stored procedure with the DB2SQL parameter style

No difference exists between a DB2SQL and an SQL parameter style external stored procedure from the caller program’s perspective. For details about calling an external stored procedure with DB2SQL parameter style and how to use SQLSTATE to check for possible error conditions that occurred within the stored procedure, see “Calling the external stored procedure with SQL parameter style” on page 53.

4.3.3 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.

Two tables, ORDERHDR and ORDERDTL, are in our Order Entry database. A referential constraint is 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.

Assume that you want to insert the specific order detail values into the ORDERDTL table. To accomplish this task, we implement these values as an external stored procedure ORDDETINS by using the GENERAL WITH NULLS parameter style. This external stored procedure accepts four input parameters to insert 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 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 must be returned to the calling program, indicating the failure of the order detail insertion.

We examine the CREATE PROCEDURE statement for the ORDDETINS external stored procedure. The numbered sections are explained in the following list:

```
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)  ) 1
SPECIFIC                        ORDDETINS
LANGUAGE                        RPGLE
EXTERNAL NAME                   SPROCLIB.ORDDETINS   2
MODIFIES                        SQL DATA
PARAMETER STYLE                 GENERAL WITH NULLS   3
```

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 was defined: SQLST of type CHAR(5).
2. The EXTERNAL NAME clause of the CREATE PROCEDURE statement. The external program is a *PGM object.
3. The PARAMETER STYLE clause. DB2 Universal Database for iSeries passes indicators as additional parameters, apart from the input and output arguments that are defined in the CREATE PROCEDURE statement. When the stored procedure is registered by using System i Navigator, the parameter style **Simple, allow null values** must be selected.

Now, we examine the external program ORDDETINS that is referred to in the previous CREATE PROCEDURE statement. 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 the ordered product as the input arguments. The SQL INSERT statement is executed. If the order number that is 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 by using the sqlstate output parameter.

The code sample in Example 4-5 illustrates how the external stored procedure with the GENERAL WITH NULLS parameter style is coded. The numbered areas are further explained in the following list.

Example 4-5 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          1
     c                   parm                    prdnbr          5          1
     c                   parm                    ordqty          5 0        1
     c                   parm                    ordtot          9 2        1
```

```
c                   parm                    state           5         2
c                   parm                    indds                     3
c                   parm                    outdd                     3
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'
c                   goto      END
c     ERROR         TAG
c                   if        SQLCOD=-530
c                   eval      state='38IRC'
c                   else
c                   eval      state='38999'
c                   endif
c     END           TAG
c                   eval      *inlr=*ON
```

Code sample notes

The following notes refer to Example 4-5 on page 58:

1. The ORDDETINS program accepts the first four input parameters, which will 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 that we are trying to insert into the ORDERDTL table might not match the 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. Because four input parameters are used, four indicators are associated with these parameters.

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 4-6. In the calling program or the client program, you can use the output parameter to check whether an error occurred within the stored procedure.

Example 4-6 Call to the external stored procedure

```
#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:\n\n");
  .....
  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((strncmp(state,"37999",5) > 0) &&
     (strncmp(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 that is shown in bold illustrates how to use the user-defined SQL state that is set by the external stored procedure that is registered with the GENERAL WITH NULLS parameter style.

4.4 Returning result sets from external procedures

Until now, we described 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. Two ways are available to return a result set:

- Cursor result set
- Array result set

The following section describes 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 by using the System i Navigator New Procedure window, you must specify the number of result sets that must be returned to the calling program. An invoker must use JDBC, ODBC, or the command-line interface (CLI) when they call a procedure that returns result sets. The System i server has no embedded SQL support for handling result sets.

4.4.1 Coding external stored procedures that return cursor result sets

In this section, we return to our task of identifying the best and the worst suppliers in the Order Entry database.

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, one or more rows are in the ORDERDTL table, for the same order number. The ORDERDTL table contains the order number (ORHNBR) of the order that is 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 supplier number. The product details are in the STOCK table. Every supplier has a supplier name, supplier number, and supplier address. The supplier details are in the SUPPLIER table. For details about the database layout, see 2.2, “Order Entry database overview” on page 11. Now, assume that you want to identify *n* suppliers with the highest sales in a certain year and month. At the same time, you also want to retrieve *n* suppliers with the lowest sales in a certain year and month. You can implement this business logic by coding a 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 10 suppliers with the highest sales in a certain month and year as the first result set. You also see a list of 10 suppliers with the lowest sales in a certain 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 available rows in the result sets (might be fewer 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 certain month in a year.

To implement this scenario, we used three views: SALES, TOTALSALE, and YEARSALE that are created on the ORDERHDR, SUPPLIER, and the STOCK tables.

We 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 following list:

```
CREATE PROCEDURE                SPROCLIB.Get_Supplier_Rs(     1
                                      IN year INTEGER,        1
                                      IN month INTEGER,       1
                                      INOUT rank INTEGER )    1
RESULT SETS                     2                             2
SPECIFIC                        Get_Supplier_Rs
LANGUAGE                        RPGLE
EXTERNAL NAME                   SPROCLIB.SELPGMRESR
MODIFIES                        SQL DATA
PARAMETER STYLE                 SQL                           3
```

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. The number of result sets that are returned from the procedure.
3. The parameter style is SQL.

Now, we examine the external program SELPGMRESR, which is referred to in the previous CREATE PROCEDURE statement. 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 4-7.

Example 4-7 SELPGMRESR source code

```
     dCounter1         s             10i 0 inz(0)
     dCounter2         s             10i 0 inz(0)
     dtemp             s             10i 0 inz(0)
     dfsqlcod          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 1
     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  5
     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 2
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 7
c+ select  supplier_name,totalsales
c+ from suparna1/totalsale
c+ 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
c+ select  totalsales
c+ from suparna1/yearsale
c+ where year=:year
c+ order by totalsales
c/end-exec
c/exec sql
c+ declare alt7 cursor for
c+ select  supplier_name,totalsales
c+ from suparna1/yearsale
c+ 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
c* 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 alt6
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  alt6 into :hsales
c/end-exec
c                   if        (fsqlcod=1)
c                   eval      rank=temp
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 9
c/end-exec
c
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
c* in this loop -fetch all the rows in the resultant set into var:array
c                   dow       ((sqlcod<>100) and (Counter1<rank)) 3
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
c                   eval      rank=Counter1
c                   endif
c/exec sql
c+ close alt0
c/end-exec
c/exec sql
c+ open alt0
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  alt0 into :hsales 4
c/end-exec
c                   if        (fsqlcod=1)
c                   eval      rank=temp
c                   endif
c/exec sql
c+ close alt0
c/end-exec
c/exec sql
c+ open  alt1
c/end-exec
c/exec sql
c+ open alt4
c/end-exec
c* 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  alt4
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 alt4
c/end-exec
c/exec sql
c+ open alt4
```

```
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  alt4 into :hsales
c/end-exec
c                   if        (fsqlcod=1)
c                   eval      rank=temp
c                   endif
c/exec sql
c+ close alt4
c/end-exec
c/exec sql
c+ open  alt5
c/end-exec
c/exec sql
c+   set result sets cursor alt1,cursor alt5 6 8
c/end-exec
c                   if        Counter1>Counter2
c                   eval      rank=Counter1
c                   else
c                   eval      rank=Counter2
c                   endif
c                   endif
c                   return
```

Code sample notes

The external program name SELPGMRESR (Example 4-7 on page 62) is a *PGM object, which is referred to in the CREATE PROCEDURE statement in the following way:

```
external name      SPROCLIB/SELPGMRESR
```

The following numbers refer to the numbers in Example 4-7 on page 62:

1 To find the total number of suppliers in a certain 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, the highest total sales value is 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, the lowest total sales value is in the first row of the resultant table when the cursor is opened.

3 The cursor might return fewer 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 might retrieve any number of rows, but the procedure returns only *n* rows, where *n* is the value of the third parameter RANK. The FETCH RELATIVE statement is used to retrieve the “cutoff” sales value into the *hsales* variable. Because we use the FETCH RELATIVE statement, the cursor must be declared as a scrollable cursor.

5 The *nth* total sales value that was fetched from the ALT0 cursor is used in the ALT1 cursor to create a result set that contains records for the *n* highest total sales.

6 This cursor ALT1 is opened and returned as a result set.

7 Similarly, the *nth* total sales value that is fetched from the ALT4 cursor is used in the ALT5 cursor to create a result set that contains records for the *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 specific 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.

The SELPGMRESR program was created as a *PGM object. In this case, if SELPGMRESR is a program that is written in C with embedded SQL statements, it is compiled into a *MODULE object. Then, the *MODULE object is bound into the *PGM object so that we can specify the activation group parameter as *CALLER.

Calling stored procedures that return multiple result sets

To invoke the external stored procedure, use the SQL CALL statement from the client program. The CLI client programs that are 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 4.4, “Returning result sets from external procedures” on page 60.

4.4.2 Coding external stored procedures that return array result sets

This section shows how to code an array result set. The external stored procedure must be registered by using either the CREATE PROCEDURE statement or the System i Navigator New Procedure window. The number of result sets that are returned must be set to 1. An external stored procedure that returns an array result set can return only one result set.

Example 4-8 shows the correct CREATE PROCEDURE statement.

Example 4-8 CREATE PROCEDURE statement

```
CREATE PROCEDURE                SPROCLIB.SELPGMARR(
                                      IN orhnbr CHAR(5) )
RESULT SETS                     1
LANGUAGE                        RPGLE
EXTERNAL NAME                   SPROCLIB.SELPGMARR
READS                           SQL DATA
PARAMETER STYLE                 GENERAL
```

The ORDERDTL table contains the order detail entries for every order number in the ORDERHDR table. For every order number in the ORDERHDR table, a number of rows are in the ORDERDTL table with the same order number, but with different product numbers. Assume 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 that is passed as an input parameter. The records are retrieved until end of file (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 4-9.

Example 4-9 ORDERDTL that is processed as a file and returns rows as an array result set

```
forderdtl  if   e           K disk    rename(orderdtl:norderdtl)
di                s              3  0
dproduct          ds                   occurs(5)                  1
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                  2
c/end-exec
c                   return
```

Sample code notes

The following notes refer to Example 4-9 on page 67:

1 The data structure declaration to hold the rows of the array result set while the rows 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:

```
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 that is returned from the procedure is shown in Figure 4-6.

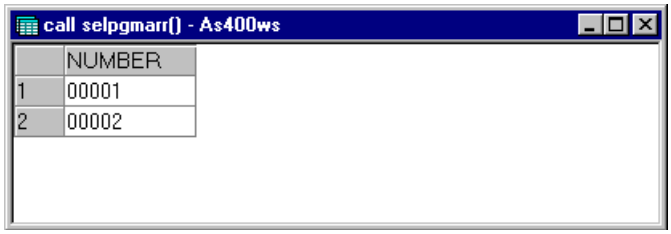


Figure 4-6 An array result set that is returned from SELPGMARR

4.5 CLI client program that calls a procedure that returns multiple result sets

When your client application calls a local or remote stored procedure, your client application must 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 by using the compiler parameter called *Relational Database* (RDB). When you run an application program that is compiled with a Relational Database name, it is implicitly connected to the database that is specified in the parameter. This database can be either local or remote.

If you will 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 4.4, “Returning result sets from external procedures” on page 60, we implemented an example that returns multiple result sets. In this section, we examine the CLI program that is written in ILE C, which calls the Get_Supplier_Rs stored procedure and displays the multiple result sets that are 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 4-10. The client program is created as a *PGM object.

Example 4-10 CLI C client program

```
#include <stdio.h>
/****variable declaration***************/
..................
/****program *****/
     printf( "Please enter the name of the server to connect :\n" );
     gets( Chr_ServerName );
     printf( "Please enter th User Id :\n" );
     gets( Chr_UserId );
     printf( "Please enter the Pass Word :\n" );
     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  " );                          1
     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,  2
                                    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,              3
                                  ( 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 );                     4
           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 );                     5
     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_CleanUpOO4( &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,              5
                                  ( 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_CleanUpOO4( &Hnd_Henv,                 6
                                          &Hnd_Hdbc,
                                          &Hnd_Hstmt );


}
```

Code sample notes

The following notes refer to Example 4-10 on page 69:

1. The CALL statement is prepared with parameter markers. All of the parameters, the name of the procedure to execute, and the library name are taken as input from the user. Therefore, this CLI program can be used to execute any stored procedure that is written in C, RPGLE, SQL, or Java that implements the business logic that is explained in 4.4, “Returning result sets from external procedures” on page 60.
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 correct SQLBindCol function must be executed.
4. After the SQLBindCol, the SQLFetch function retrieves the column values from the returned result set. SQLFetch executes until no more rows are in the result set.
5. We check for more result sets with the SQLMoreResults function. If more than one result set is identified, the sequence of SQLBindCol and SQLFetch calls must 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 that are returned by the stored procedure. Finally, the program releases all of the handlers that it acquired and disconnects from the server.

The results of executing the GETSUPPRS program are shown in Figure 4-7.

```
   The No of rows returned 4:
The List of Suppliers with 4 Best Totalsales:
  The value of Supplier Name is Black
  The value of Total Sales   is 8800.00


  The value of Supplier Name is Red
  The value of Total Sales   is 7345.00


  The value of Supplier Name is Blue
  The value of Total Sales   is 3150.00


  The value of Supplier Name is Yellow
  The value of Total Sales   is 1200.00


  The No of rows returned 4
  The List of Suppliers with 4 Worst TotalSales:

===>
The value of Supplier Name is Yellow
The value of Total Sales   is 1200.00


The value of Supplier Name is Blue
The value of Total Sales   is 3150.00


The value of Supplier Name is Red
The value of Total Sales   is 7345.00


The value of Supplier Name is Black
The value of Total Sales   is 8800.00


There are no more result result sets
Press ENTER to end terminal session.
```

Figure 4-7 The CLI C client output

Debugging a stored procedure

The external stored procedures that are called from the client program can be debugged as any other native ILE C program. To enable debugging, ensure that you compile the stored procedure with 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:

```
STRDBG PGM(SPROCLIB/SELPGMRESR)
```

The stored procedure source is now 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.

4.6 Moving into production (save and restore)

While you deploy 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, additional actions might be required to make the stored procedure available on the target system.

When CREATE PROCEDURE is used to create an external stored procedure that is 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 system 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 original program model (OPM) or an ILE with no embedded SQL statements, you need to register the program as a procedure with the CREATE PROCEDURE statement. The SQL Development Kit does not need to be installed on the target system. You can also use one of the following SQL interfaces:

- System i Navigator
- WRKQMQRY
- **`RUNSQLSTM`** CL command
- A program that uses an ODBC, a JDBC, or a CLI interface

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 correct stored procedure. However, several topics require your attention:

- The external procedure is registered in the target library that is specified by the **`RESTORE`** command, regardless of the CREATE PROCEDURE source.
- If the matching external procedure is not in the catalogs, DB2 Universal Database for iSeries automatically registers the program as a stored procedure.
- If the matching external procedure is in the catalogs with the same signature (same number of parameters), the program is restored, but the catalog information is not updated. This method works fine if 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 in the catalogs and the signature is different (which means different parameters), the program object is restored and the catalog information is updated.

In general, be careful when you restore a stored procedure with a modified number or type of parameters because the restore operation might overlay the program object for an existing procedure that shares the signature. If the procedure exists in the target library or system and the number of parameters or the data type of the parameters changed, a DROP PROCEDURE statement must be performed before the restore operation.

4.7 The Order Entry application: Stored procedure examples

In our application scenario, we use a stored procedure to perform 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 that 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 because the search for an alternative item can involve many 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 identified, the product description (PARM3) is returned as NULL. The stored procedure was defined in the CREATE PROCEDURE statement as SIMPLE CALL WITH NULLS. The SIMPLE CALL WITH NULLS clause is synonymous with GENERAL WITH NULLS and it 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) must 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 2-digit binary subfields: IND1, IND2, IND3, and IND4. However, we use only IND1 and IND3 in our program.

If the stored procedure identifies 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 0.

If no record is identified, IND1 is set to -1, which means that PARM1 has a NULL value. When returning to the calling program, this indicator must be tested. If it has a value of -1, an error message is displayed.

If the item is identified 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 several alternatives might exist with the required quantity in stock. The alternative product description and price are passed back to the calling program.

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 by using two-phase commit.

For details about our application scenario, see Chapter 2, “Stored procedures, triggers, and user-defined functions for an Order Entry application” on page 9.

4.7.1 Calling a stored procedure

The listing in Example 4-11 is an RPG version of the Insert Detail (INSDET) client program with embedded SQL statements.

Example 4-11 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 CF  E                    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   2OIND1
I                                    B   3   4OIND2
I                                    B   5   6OIND3
I                                    B   7   8OIND4
 *
 *****************************************************************
 *  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'0'
C           *IN64     ANDEQ'0'
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
 *
 ****************************************************************
 *  -- 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 reconnections SET CONNECTION is used:   *
 ****************************************************************
C*
C/EXEC SQL
C+ SET CONNECTION ROCHESTER                          -- After 1.conn
C/END-EXEC                                              remote
 *
 ****************************************************************
 *  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     IND3             -1=no alternat.
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/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
C                     END
 *
C           SQLCOD    IFEQ -803                       Duplicate key
C                     SETON                     63
C                     EXSR SUBTR
 *
 ***************************************************************
 *  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
C                     GOTO BEGIN
C                     END
C*
C           SQLCOD    IFEQ -530                       RI Constraint
C                     MOVE '1'       RTNCDE
C                     SETON                     64
C                     EXSR SUBTR
 *
 ***************************************************************
 *  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                     GOTO BEGIN
C                     END
C                     SETON                     22
 *
C                     EXFMTINSDT1                     2nd display
 *
C                     SETOF                     2267
 *
C           *IN03     IFEQ '1'                        End
C                     GOTO SETLR
C                     END
C                     MOVE '0'       RTNCDE
 *
C           *IN11     IFEQ '1'                        Cancel Item
C                     EXSR SUBTR
C                     MOVE '1'       RTNCDE
 *
```

```
 ****************************************************************
 *  If customer does not agree with alternative item, the order *
 *  item can be canceled 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 canceled 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:  *
 ****************************************************************
 *
C/EXEC SQL
C+ COMMIT
C/END-EXEC
```

```
C                   RETRN
 *
 * ----------------------------------------------------
C           SUBTR     BEGSR
 ****************************************************************
 *  The accumul. order total on the display has to be adjusted: *
 ****************************************************************
C                   SUB  DITTOT    DODCUM
C                   ENDSR
```

4.7.2 Sample stored procedure: SQL RPG version

The code listing in Example 4-12 shows the SQL RPG version of the stored procedure.

Example 4-12 SQL RPG version of the stored procedure

```
F*****************************************************************
F* This is the stored procedure called by T4249RADT which processes
F* order detail records, and runs on the remote system where
F* the inventory file (STOCK) locates.
F* This program updates the available quantity of the product
F* which customer orders, if there is enough quantity to meet
F* the ordered quantity.
F* If there is not enough quantity, an alternative within
F* same category will be searched and if found, the alternative
F* record will be updated and returned to the calling program
F* instead the customer-ordered product.
F* The change to the file is committed or rolled back by
F* 2-phase commitment control.
F*****************************************************************
I*
I*   The CALL type of this stored procedure was defined as
I*   "SIMPLE CALL WITH NULLS" on the DECLARE PROCEDURE statement
I*   in the calling program.  Therefore an additional indicator
I*   variable parameter is passed to this stored procedure.
I*   Since the calling program defines four parameters on the
I*   DECLARE PROCEDURE statement, the indicator parameter has four
I*   2-digit Binary variables.
I*
I*   This data structure defines an additional indicator parameter.
I*
IINDDS       DS
I                                     B   1   2OIND1
I                                     B   3   4OIND2
I                                     B   5   6OIND3
I                                     B   7   8OIND4
I*
C*   Five parameters should be defined.
C*
C           *ENTRY    PLIST
C                     PARM           PARM1   5        Product Number
C                     PARM           PARM2   5O       Ordered Qty.
C                     PARM           PARM3  2O        Product Desc.
C                     PARM           PARM4   72       Product Price
C                     PARM           INDDS            Indicator Parm.
C*
C                     MOVE *ZERO     WQTA    5O
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
C+     WHERE PRDCAT = :WCAT AND
C+           PRDQTA >= :PARM2
C+     FOR UPDATE OF PRDQTA
```

```
C/END-EXEC
C*
C/EXEC SQL
C+   OPEN ALT1
C/END-EXEC
C*
C/EXEC SQL
C+  FETCH ALT1 INTO :PARM1, :PARM3, :PARM4, :WQTA
C/END-EXEC
C*
C           SQLCOD    IFEQ 100
C                     MOVE -1        IND3
C                     MOVE *ZERO     IND1
C                     MOVE *BLANK    PARM3
C                     MOVE *ZERO     PARM4
C                     GOTO DONE1
C                     END
C*
C           WQTA      SUB  PARM2     LEFTQ
C/EXEC SQL
C+ UPDATE ORDENTR/STOCK SET PRODUCT_AVAIL_QTY = :LEFTQ
C+ WHERE CURRENT OF ALT1
C/END-EXEC
C*
C                     MOVE *ZERO     IND1
C                     MOVE *ZERO     IND3
C*
C           DONE1     TAG
C/EXEC SQL
C+  CLOSE ALT1
C/END-EXEC
C*
C                     RETRN
C*
C           DONE0     TAG
C/EXEC SQL
C+  CLOSE ALT0
C/END-EXEC
C*
C                     RETRN
```

4.8 External stored procedure that uses a service program

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 provide an example of how to use this support. We use example program codes (Example 4-13) from 3.5.4, “Using an ILE Service Program,” of *Who Knew You Could Do That with RPG IV? A Sorcerer’s Guide to System Access and More*, SG24-5402.

Example 4-13 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                                        4
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)    4A

 * 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                                        1
D  WorkDate                      D   Datfmt(*ISO)          2
D  DayName                      9A                         5

C                   EVAL      DayName = Name(DayOfWeek(Workdate)) 3
 *                  Return    Name(DayOfWeek(Workdate))
 *                  Return    DayName

P                   E

 * 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                       3A
D  WorkDate                      D   Datfmt(*ISO)          3A

 * Stand Alone Fields
D AnySunday       S               D   Inz(D'1995-04-02')
D WorkNum         S               7  0
D WorkDay         S               1  0

C     WorkDate      Subdur    AnySunday     WorkNum:*D
C     WorkNum       Div       7             WorkNum
C                   Mvr                     WorkDay
C                   If        WorkDay < 1
C                   Return    WorkDay + 7
C                   Else
C                   Return    WorkDay
C                   Endif

P                   E
```

The following explanations refer to the numbers in Example 4-13 on page 83.

These example program codes are of two subprocedures that are named NameOfDay (1) and DayOfWeek (3A). The subprocedure NameOfDay is the *entry point* of the external stored procedure that you will create. Therefore, 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 that is 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 that is named SQLLIB, you use the following CL commands to create the service program object:

```
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:

```
CREATE PROCEDURE SQLLIB.NAMEOFDAY (
IN WorkDate DATE, OUT Dayname CHARACTER(9) )   B
LANGUAGE RPGLE
SPECIFIC SQLLIB.NAMEOFDAY
NOT DETERMINISTIC
NO SQL
CALLED ON NULL INPUT
EXTERNAL NAME 'SQLLIB/NAMEOFDAY(NAMEOFDAY)'    A
PARAMETER STYLE GENERAL WITH NULLS;
```

At line A, 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 the Run SQL Scripts utility to try the procedure call with the output parameter that is displayed in the Messages tab of the window. 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 4-8 on page 86:

```
CALL SQLLIB.NAMEOFDAY('2005-11-25', ?) ;
```

Figure 4-8 shows the call for the external stored procedure NAMEOFDAY and its results.

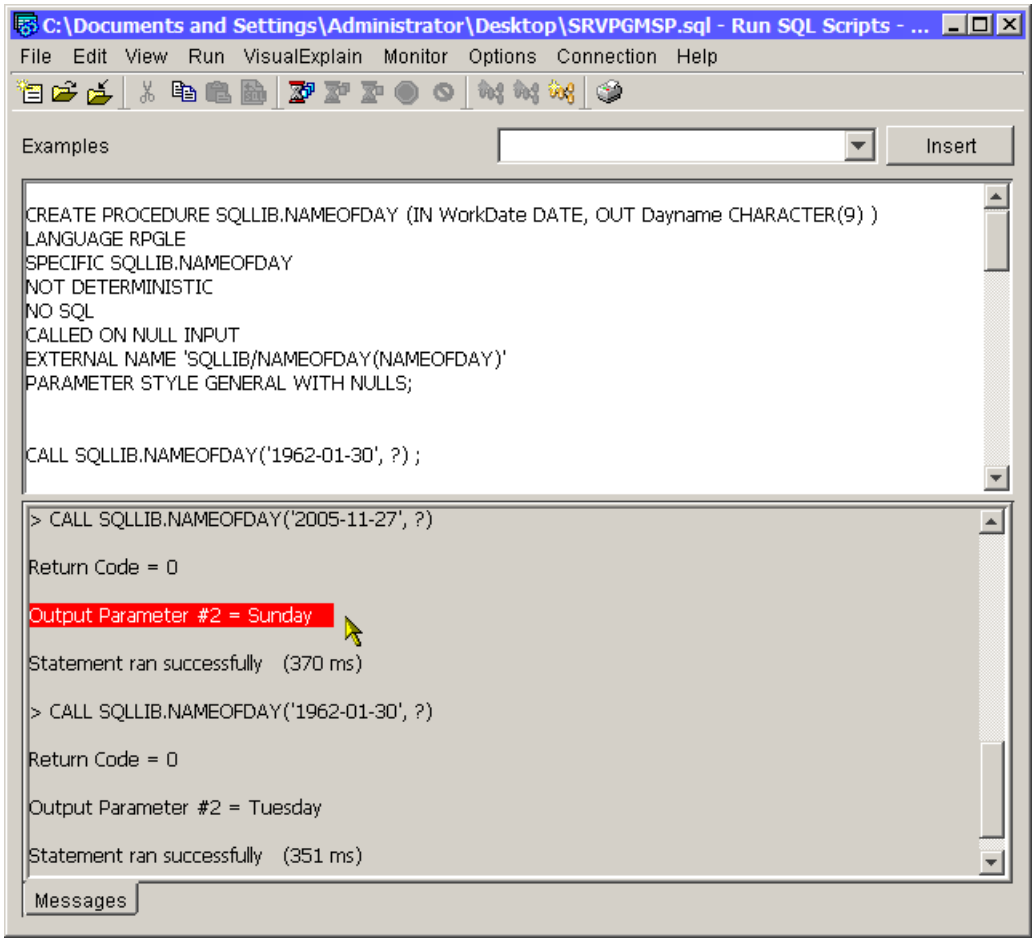


Figure 4-8 Calling the external stored procedure NAMEOFDAY and its results

For more information about the RPG IV subprocedure, see Chapter 3, “Subprocedures,” in *Who Knew You Could Do That with RPG IV? A Sorcerer’s Guide to System Access and More*, SG24-5402.

4.9 RPG IV example for an external stored procedure

We provide examples of RPG IV programs that write to and read from a data queue, which you can register as external stored procedures. These RPG programs use the OS/400 application programming interfaces (APIs) SNDDTAQ and RCVDTAQ. For a detailed description of these APIs, see 5.2.2, “List of data queue APIs,” of *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 book.

Assume that all program objects and the data queue are created in a library that is named `SQLLIB` and that the data queue name is DTAQFIFO for this example.

First, you create a data queue (with a 40-byte maximum length) with the following control language (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.

4.9.1 External stored procedure that writes to a data queue

You create a bound RPG program object from the program code that is shown in Example 4-14.

Example 4-14 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         1
D  DQlib                        10A         2
D  DataSnd                      40A         3
 *-------------------------------------------------------------------------
 * Write data to data queue names DTAQFIFO in library SQLLIB
C                   CallP     SndDtaQ(DQname : DQlib
C                                     : %Len(%Trim(DataSnd)) : DataSnd)
 *
C                   Eval      *InLR = *On
```

The program in Example 4-14 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, which is named WRTDTAQ, by 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 reads from the data queue.

4.9.2 External stored procedure that reads from a data queue

You create a bound RPG program object from the program code that is shown in Example 4-15.

Example 4-15 Program example RDDTAQ to read a text string from a data queue

```
 ********************************************************************
 * Prototype for API QRCVDTAQ - Received From a Data Queue
D RcvDtaQ         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         A
D  DQlib                         10A         B
D  Output                        40A         C2
D  DataRcv        S              40A         D1
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 )  D2
C                   eval      Output = DataRcv                C1
C                   Eval      *InLR = *On
```

The program in Example 4-15 takes two *input* parameters: Data queue name (DQname at A) and Data queue library (DQlib at B). The text string that is 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, which is named RDDTAQ, by 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.

4.9.3 Calling external stored procedures from the 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 System i Navigator:

```
CALL SQLLIB.WRTDTAQ('DTAQFIFO', 'SQLLIB', 'Hello data queue 1') ;

CALL SQLLIB.RDDTAQ('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 4-9.

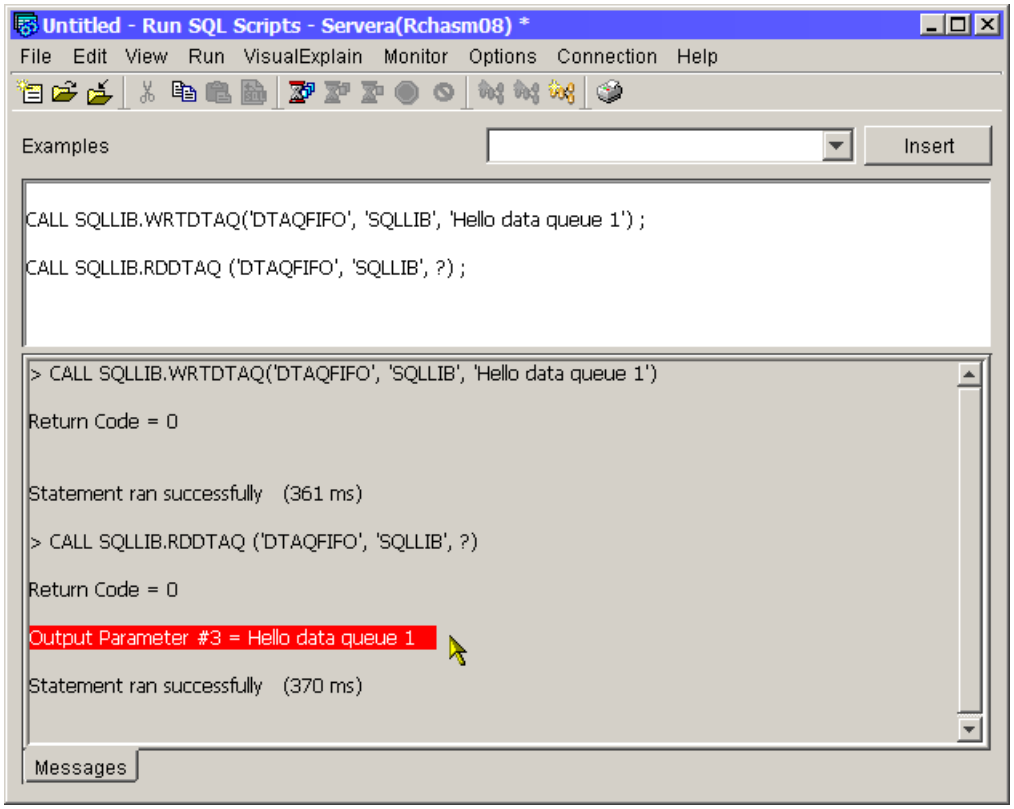


Figure 4-9 Calling external stored procedures WRTDTAQ and RDDTAQ and their results

Consider the following points:

- The data queue name and its library name must be entered in uppercase.
- The output parameter of the RDDTAQ procedure must be specified with the NULL value when it is called. You can type the question mark (?) to represent a NULL value.

> **Note:** This technique is a useful way to use an external procedure to access an OS/400 object (that is not SQL) from an SQL stored procedure.

5

Java stored procedures

This chapter describes the Java stored procedures implementation on the IBM i server. It also compares the support that is provided by the IBM i server with the implementations on other DB2 platforms. The combination of Java and a powerful database server, such as IBM DB2 for i, 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 because this book provides a detailed description of the Java stored procedure implementation. For a more detailed description of how to use Java on the IBM i server, see *Building Java Applications for the iSeries Server with VisualAge for Java*, SG24-6245.

The current implementation of Java stored procedures on the IBM i server aims to provide the same level of support as other DB2s. For more information about Java stored procedures on the DB2 Universal Database platform, see 5.10, “GetSupplierRS example: Implementation with result sets” on page 132.

This chapter focuses on the four fundamental tasks that are involved in the creation of and use of 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 describes problem determination tools. This chapter also outlines the major considerations for the current implementation on the IBM i server. Finally, it provides a more elaborate example of a stored procedure that is called *GetSupplier*.

The following topics are covered in this chapter:

- Prerequisites
- Coding DB2 for i Java stored procedures
- Coding examples
- Registering Java stored procedures
- Calling Java stored procedures
- Using SQL NULL

- SQLJ procedures to manipulate JAR files
- Additional considerations
- GetSuppliers example: Implementation with no result sets
- GetSupplierRS example: Implementation with result sets
- Problem determination

5.1 Prerequisites

Two required products are needed for Java stored procedure support on the IBM i server:

- 5769-SS1 V4R5M0 Operating System/400 or higher
- 5769-JV1 V4R5M0 Developer Kit for Java or higher

Important: We strongly recommend that you install the latest database fix pack (SF99105) that is available for your system.

5.2 Coding DB2 for i Java stored procedures

A Java stored procedure corresponds to a method in a Java class that meets several requirements that are explained in this section.

Before we start with a detailed explanation of how to code this method, we need to describe briefly two different parameter styles that are supported on the DB2 for i. The parameter style that is 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.

5.2.1 Parameter styles

DB2 for i supports two parameter styles for Java stored procedures:

- JAVA (conforms to 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:

```
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 adopted convention is that output (and inout) parameters are represented in the Java method by an array of size one.

This convention and the way that result sets are returned are the main characteristics of the JAVA parameter style stored procedures. The following example illustrates how to define the methods that correspond to stored procedures with input, output, and inout parameters. For illustration purposes, we placed the various types of parameters in distinct methods. In an actual scenario, a stored procedure and its corresponding method are likely to use a mix of several types of parameters (IN, INOUT, and OUT):

```
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
  }
}
```

Important: The type of parameters that you choose for your procedure is important because you must choose a Java data type that can be mapped to an SQL data type. Otherwise, your Java method is simply ignored by the database. For more information about the compatibility of data types, see Table 5-1 on page 96.

The SQL NULL value can be passed to a stored procedure only when the corresponding Java data type can have a null reference value, which works for String, byte[], BigDecimal, Date, Time, TimeStamp, Double, Float, Integer, and Long, but it does not work 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 impossible to test whether, for example, an SQL INTEGER parameter that is passed to the Java stored procedure by using the JAVA parameter style is NULL. The Structured Query Language for Java (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 whether their value is NULL.

For portability reasons, consider the use of the JAVA parameter style because it is supported by most database platforms.

Note: The binary large objects (character large object (CLOB), binary large object (BLOB), and double byte large object (DBCLOB)) cannot be passed as parameters of the Java stored procedure in this implementation of the JAVA parameter style on the IBM i server.

DB2GENERAL parameter style

This parameter style is specific to the DB2 platform and it is not a part of the Java Database Connectivity (JDBC) or SQLJ standard. It was first introduced in Version 5 of DB2 Universal Database.

Note: The stored procedure interface that is provided by System i 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 in the integrated file system (IFS) in the `/QIBM/ProdData/OS400/Java400/ext` directory. The method that corresponds to the Java stored procedure must be defined. A `public void` method as illustrated:

```
public class SomeStoredProcs extends StoredProc {
  public void myStoredProc(...) {
    // SP implementation
  }
  public void anotherStoredProc(...) {
    // SP implementation
  }
}
```

The output parameters are declared similar to 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:

```
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 that are used for the parameters must be compatible with SQL data types. For more information about data type compatibility, see Table 5-1 on page 96.

Use the DB2GENERAL parameter style to check whether the value of any parameter that is received by the stored procedure is SQL NULL with the `isNull()` method that is provided in the `StoredProc` class.

The large object (LOB), binary large object (BLOB), and character large object (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 that are contained in the same `db2routines_classes.jar` file as the `StoredProc` class.

5.2.2 Data type compatibility

Table 5-1 summarizes the compatibilities between the SQL and Java data types, depending on the parameter style convention that is used.

Table 5-1 Data type compatibilities

SQL data type	Java type (JAVA parameter style)	Java type (DB2GENERAL parameter style)
SMALLINT	short (Not nullable)	short
INTEGER	int (Not nullable)	int
BIGINT	long (Not nullable)	long
DECIMAL(p, s)	BigDecimal (Nullable)	BigDecimal
NUMERIC(p, s)	BigDecimal (Nullable)	BigDecimal
REAL or FLOAT(p)	float (Not nullable)	float
DOUBLE PRECISION or FLOAT or FLOAT(p)	double (Not nullable)	double
CHARACTER(n)	String (Nullable)	String
VARCHAR(n)	String (Nullable)	String
VARCHAR(n) FOR BIT DATA	N/A	com.ibm.db2.app.Blob
GRAPHIC(n)	String (Nullable)	String
VARGRAPHIC(n)	String (Nullable)	String
DATE	Date (Nullable)	String
TIME	Time (Nullable)	String
TIMESTAMP	Timestamp (Nullable)	String
CLOB	N/A	com.ibm.db2.app.Clob
BLOB	N/A	com.ibm.db2.app.Blob
DBCLOB	N/A	com.ibm.db2.app.Clob

5.2.3 Database connection in a Java stored procedure

The Java code can interact with the database as any other Java program, but the database connection can be established with the local database by using the current user ID only. Both the JDBC and SQLJ interfaces can be used to access the database.

JDBC

The method to connect to the database depends on the parameter style convention that is 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:

  ```
  Connection con = DriverManager.getConnection("jdbc:default:connection");
  ```

- DB2GENERAL: JDBC creates the connection object to the DB2 for i database with the `getConnection()` method that is provided by `StoredProc`. The following code example illustrates the syntax:

  ```
  Connection con = getConnection();
  ```

SQLJ

For SQLJ, the idea of a connection is replaced by the concept of a *context*. It is also necessary to distinguish between the two parameter styles:

- JAVA: Use the `ExecutionContext` class as shown in the following example:

  ```
  ExecutionContext ec = DefaultContext.getDefaultContext().getExecutionContext();
  ```

- DB2GENERAL: You must explicitly create the DefaultContext object with the `getDefaultContext` method, as shown in the following code example:

  ```
  DefaultContext ctx = DefaultContext.getDefaultContext();
   if (ctx == null)
      { Connection con = getConnection ();
        ctx = new DefaultContext(con);
        DefaultContext.setDefaultContext(ctx);
      }
  ```

5.2.4 Returning result sets in Java stored procedures

Starting in V5R1, Java stored procedures in DB2 for i supported returning result sets. How result sets are declared and coded in the stored procedure differs significantly depending on the parameter style that is used.

Returning result sets in JAVA parameter style stored procedures

In stored procedures with the JAVA parameter style, the 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 for i in the following manner:

```
CREATE PROCEDURE SPWITHTWORESULTSETS (IN INTEGER, OUT VARCHAR(50)) 1
EXTERNAL NAME MyClass!myJavaStoredProcedure
PARAMETER STYLE JAVA 2
RESULT SETS 2 3
LANGUAGE JAVA
```

The Java method to implement this stored procedure looks like this example:

```
import java.sql.*;
public class SomeStoredProcs {
   public static void myJavaStoredProcedure(
      int myInputInteger,
```

```
      String[] myOutputString,
      ResultSet[] myFirstResultSet, 4
      ResultSet[] mySecondResultSet) 4{
      // SP implementation
      ...
      myFirstResultSet[0] = stmt1.executeQuery(qry1); 5
      ...
      mySecondResultSet[0] = stmt2.executeQuery(qry2); 5
   }
}
```

Notes: The following numbered notes refer to the numbers in 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 differ significantly from result sets in stored procedures with the DB2GENERAL parameter style.
3. The number of result sets must be defined explicitly in the 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 as any other output parameter, assigning a result set object to the correct parameter vector.

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 1
RESULT SETS 2 2
LANGUAGE JAVA
```

Example 5-1 shows the Java method that implements this stored procedure.

Example 5-1 Java method that implements the CREATE PROCEDURE

```
import java.sql.*;
import com.ibm.db2.app.*; 3     // StoredProc and associated classes

public class sample2 extends StoredProc 4
{

  /**
   * Java Stored procedure with DB2GENERAL style Parameters
   * that execute TWO predefined statements
   *
   */
  public void returnTwoResultSets () throws Exception 5
  {
    // get caller's connection to the database; inherited from StoredProc
    Connection con = getConnection(); 6
    Statement stmt1 = con.createStatement();
```

```
    String sql1 = "SELECT EMPNO FROM SAMPLEDB02.EMPLOYEE WHERE HIREDATE < '1980-01-01'";
    stmt1.execute(sql1); 7
    Statement stmt2 = con.createStatement();
    String sql2 = "SELECT EMPNO FROM SAMPLEDB02.EMPLOYEE WHERE SALARY > 50000";
    stmt2.execute(sql2); 7
  }
}
```

Notes: The following notes refer to the numbers in the stored procedure and in Example 5-1 on page 98:

1. The DB2GENERAL parameter style is defined for the Java stored procedure. This parameter style can be used with Java stored procedures only.
2. The defined maximum number of result sets.
3. This import file contains `StoredProc` and `DB2GENERAL` classes and interfaces that are required for coding DB2GENERAL stored procedures and user-defined functions (UDFs).
4. Classes that contain the DB2GENERAL stored procedures must expand the `com.ibm.db2.app.StoredProc` class.
5. In contrast with Java classes, DB2GENERAL stored procedures do not declare result sets as parameters.
6. Standard connection for DB2GENERAL parameter style stored procedure.
7. In the implementation, result sets are the cursors that remain open when the method returns.

5.3 Coding examples

The simple example that is described in this section refers to the CUSTOMER table, which is part of our test database.

We present several versions of two Java stored procedures, depending on how the stored procedure interacts with the database, either by 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 inserted rows. We provide an explanation of two versions of this example:
  - *JavaInsertCus* that uses JDBC to connect and work with the database
  - *JavaSQLJInsertCus* that uses SQLJ to interact with the database
- Java stored procedure with DB2GENERAL parameter style: It counts the number of customers in a specific 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* that uses JDBC
  - *DB2SQLJCusInCity* that uses SQLJ and an SQLJ iterator for internal processing
  - *DB2SQLJCusInCity2* that uses SQLJ and where the SQLJ iterator is replaced by the SELECT INTO statement

Because the stored procedures that are used in these examples have output parameters, we cannot call them directly from an interactive interface, such as the 5250 Interactive SQL or the Run SQL Scripts utility, both of which are available in System i Navigator. We created a Java application that is 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 described in this section. Example 5-2 shows the JavaInsertCus stored procedure.

Example 5-2 JavaInsertCus stored procedure

```
import java.sql.*;                                                          1
public class JavaInsertCus                                                  2
{
   public static void JavaInsertCus (                                       3
      String s1, String s2, String s3, String s4, String s5,
      String s6, String s7, java.math.BigDecimal bd1,
      java.math.BigDecimal bd2, int[] insertCount )                         4
   throws SQLException, Exception
   {
      Connection con =
         DriverManager.getConnection("jdbc:default:connection");            5
      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();                                6
      try
      {
         if (stmt != null) stmt.close();
         if (con != null) con.close();
      } catch (SQLException e) { /* ignore */ };                           7
   }
}
```

Notes: The following numbered notes refer to Example 5-2 on page 100:

1. Import of the standard JDBC application programming interface (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`. More than one method can be in the class. The method can have any name. It does not need to match the name of the class.
4. Input parameters are defined just as 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 we receive 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 that is declared in the parameters. It is returned as an output parameter. This array cannot be used to return multiple values. Only the first element will be assigned.
7. Error handling is described in Chapter 5, "Java stored procedures" on page 91.

We can now compare the JDBC version with its much more compact SQLJ counterpart, JavaSQLJInsertCus (Example 5-3).

Example 5-3 JavaSQLJInsertCus

```
import java.sql.*;
import sqlj.runtime.*;                                                    1
import sqlj.runtime.ref.*;                                                1

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();        2
      #sql {
         INSERT INTO CUSTOMER VALUES(:s1,:s2,:s3,:s4,:s5,:s6,:s7,:bd1,:bd2)
      };                                                                  3
      insertCount[0] = ec.getUpdateCount();                               4
   }
}
```

Notes: In the following numbered notes, we describe the features in Example 5-3 on page 101 that are specific only to SQLJ. However, comments 1 through 4 that were 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, such as environment variables that are set after the execution of the SQLJ statement. With SQLJ and the JAVA parameter style, the connection exist implicitly. The JDBC version requires the use of the `getConnection` method.

3 The SQLJ statement is simply an SQL statement that is 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 were inserted by the SQL statement. It corresponds to the return value that is passed by the JDBC `executeUpdate` method that was shown in the previous version of this example.

DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples

Example 5-4 illustrates the DB2GENERAL parameter style. The first version of this stored procedure uses a JDBC connection to the database. The numbers are explained in the following notes.

Example 5-4 DB2GENERAL parameter style

```
import java.sql.*;
import com.ibm.db2.app.*;                                                    1

class DB2CusInCity extends StoredProc                                       2
{
   public void DB2CusInCity (String s, int i) throws Exception              3
   {
      Connection con = getConnection();                                      4
      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));                                                  5
      if (rs != null) rs.close();
      if (ps != null) ps.close();
      if (con != null) con.close();
   }
}
```

Notes: The following notes refer to Example 5-4 on page 102:

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 we define 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 that is 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 that were made for the previous version also apply to these versions. We emphasize only the differences.

DB2SQLJCusInCity is an SQLJ version of DB2CusInCity. It uses a result set to obtain the number of customers in a specific city.

Example 5-5 DB2SQLJCusInCity

```
import java.sql.*;
import com.ibm.db2.app.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator DB2SQLJCusInCity_C (int);                                    1

class DB2SQLJCusInCity extends StoredProc
{
   public void DB2SQLJCusInCity ( String s, int i )
   throws SQLException, Exception
   {
      DB2SQLJCusInCity_C c;                                                2
      DefaultContext ctx = DefaultContext.getDefaultContext();             3
      if (ctx == null)
      {
         Connection con = getConnection ();
         ctx = new DefaultContext(con);
         DefaultContext.setDefaultContext(ctx);
      }
      #sql c = { SELECT Count(*) FROM CUSTOMER WHERE (CUSTOMER_CITY = :s) }; 4
      #sql { Fetch :c into :i};                                            4
      c.close();                                                           4
      set(2, i);
   }
}
```

Notes: The following notes refer to Example 5-5 on page 103:

1. SQLJ defines a cursor as an iterator. It must be declared before the class and it is transformed at compilation 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 to obtain the number of customers. The result set is replaced by a SELECT INTO statement. The corresponding code of DB2SQLJCusInCity2 is shown in Example 5-6.

Example 5-6 Corresponding code of DB2SQLJCusInCity2

```
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, which is not possible with JDBC. 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.

5.3.1 Compilation of Java code

CLASSPATH is the key environment variable to compile a Java program. Depending on the imports that are made in the Java code, you need to modify **CLASSPATH** so that it points at the `.jar` files or `.zip` files that are required by these imports.

Table 5-2 summarizes the files that are needed for different programming styles.

Table 5-2 Required files in the classpath

Parameter style	Database connection	File that is required in the classpath
JAVA	JDBC	db2_classes.jar
JAVA	SQLJ	db2_classes.jar, translator.zip, and runtime.zip
DB2GENERAL	JDBC	db2_classes.jar and db2routines_classes.jar
DB2GENERAL	SQLJ	db2_classes.jar, db2routines_classes.jar, translator.zip, and runtime.zip

All of these files are in the IFS in the /QIBM/ProdData/OS400/Java400/ext directory. If you decide to compile on the IBM i server, the db2_classes.jar file that contains the DB2 native driver and the JDBC API are automatically added to the classpath. If you prefer to compile on your personal computer, you must specify all of these files in your classpath.

You compile your Java code by using the **javac** command when the stored procedure uses only JDBC to connect to the database. This approach results in one .class file. When SQLJ is used, the compilation command is **sqlj**. This approach creates one .ser file, two .class files, and one additional .class file for each iterator.

JavaInsertCus and JavaSQLJInsertCus examples

Before you compile your Java code for these examples, you need to set up your classpath so that it points to the JDBC and SQLJ classes. To facilitate our tests, we decided to compile on the personal computer and copied the IBM i server db2_classes.jar, db2routines_classes.jar, translator.zip, and runtime.zip files on the local disk of our personal computer in the D:\as400_classes directory. Our classpath is set with the command that is shown:

```
set CLASSPATH= %CLASSPATH%;D:\as400_classes\db2_classes.jar;                    1
               D:\as400_classes\translator.zip;D:\as400_classes\runtime.zip     2
```

Notes: The following notes refer to numbers in the previous example:

1. The db2_classes.jar file contains the JDBC driver and the JDBC classes. Your original classpath (%CLASSPATH%) must already point to the Java Developer's Kit (JDK) classes.
2. The translator.zip file provides the SQLJ translator (or precompilator). The runtime.zip file contains the Java runtime support for SQLJ.

You can now compile your .java and .sqlj files that contain the stored procedures with the commands that are listed in Table 5-3.

Table 5-3 Compilation commands and their output files

Compilation commands	Result files
javac JavaInsertCus.java	JavaInsertCus.class
sqlj JavaSQLJInsertCus.sqlj	JavaSQLJInsertCus.java, JavaSQLJInsertCus.class, JavaSQLJInsertCus_SJProfile0.ser, and JavaSQLJInsertCus_SJProfileKeys.class

Note: We assume that a version of the sqlj precompiler is installed on your workstation in advance. In our case, it was a part of our test DB2 Universal Database for the Microsoft Windows NT installation and was in the `\SQLLIB\bin` directory.

DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples

The classpath and compilation considerations are similar to those considerations that were made for the JAVA parameter style example. The difference is the additional `.jar` file that contains 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, which contain the stored procedures, produces the files that are shown in Table 5-4.

Table 5-4 Compilation commands and their results

Compilation commands	Result files
`javac DB2CusInCity.java`	`DB2CusInCity.class`

`sqlj DB2SQLJCusInCity.sqlj`	`DB2SQLJCusInCity.java`
	`DB2SQLJCusInCity.class`
	`DB2SQLJCusInCity_C.class`
	`DB2SQLJCusInCity_SJProfile0.ser`
	`DB2SQLJCusInCity_SJProfileKeys.class`

`sqlj DB2SQLJCusInCity2.sqlj`	`DB2SQLJCusInCity2.java`[1]
	`DB2SQLJCusInCity2.class`
	`DB2SQLJCusInCity2_SJProfile0.ser`
	`DB2SQLJCusInCity2_SJProfileKeys.class`

[1] The `.java` file that was produced by the sqlj precompilation confirms that a result set and the `next()` method are used for the SELECT INTO implementation, as described in "DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples" on page 102.

5.3.2 Where to place Java classes

The compiled code must be loaded into the `/QIBM/UserData/OS400/SQLLib/Function` function directory of the IBM i 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 IBM i 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 IBM i server (`/QIBM/UserData/OS400/SQLLib/Function`), as shown in Figure 5-1.

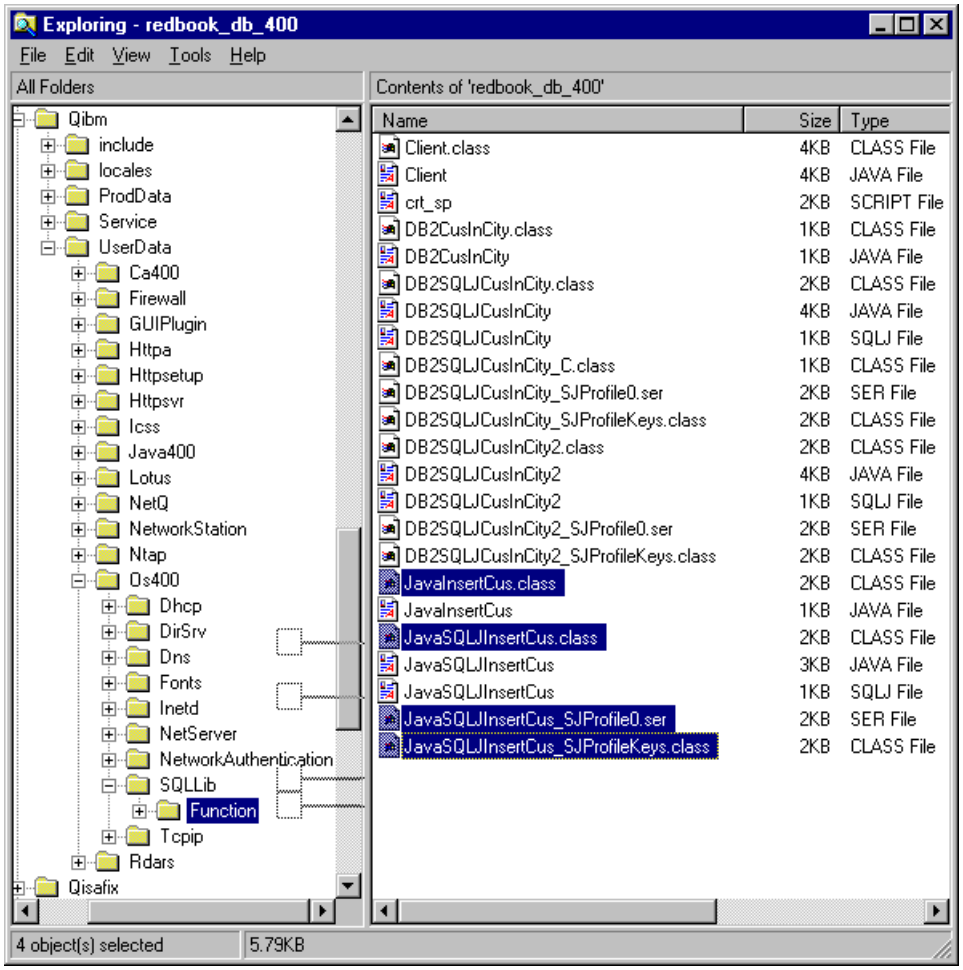


Figure 5-1 Copying the Java code into the IBM i server Function directory

DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples

In this example, apart from the `.java` file that was produced by the SQLJ precompilation, we copy all other files that result from the compilation into the `Function` directory of the IBM i server.

5.3.3 Creating Java programs

After you compile the Java stored procedure and deploy it on the IBM i server, you might 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 IBM i server and associates the Java program object with the class file. To create the optimized Java program for the `DB2CusInCity` class, use the following control language (CL) command:

```
CRTJVAPGM CLSF(Db2CusInCity.class) OPTIMIZE(40)
```

5.4 Registering Java stored procedures

The last step before you complete the Java stored procedure is to identify the code that you loaded in the `Function` directory as a Java stored procedure and to register it in the system catalog (SYSROUTINES and SYSPARMS system tables). Use the CREATE PROCEDURE SQL statement. The syntax of this statement was extended to allow the new type of stored procedures. Figure 5-2 details the new syntax of the CREATE PROCEDURE statement.

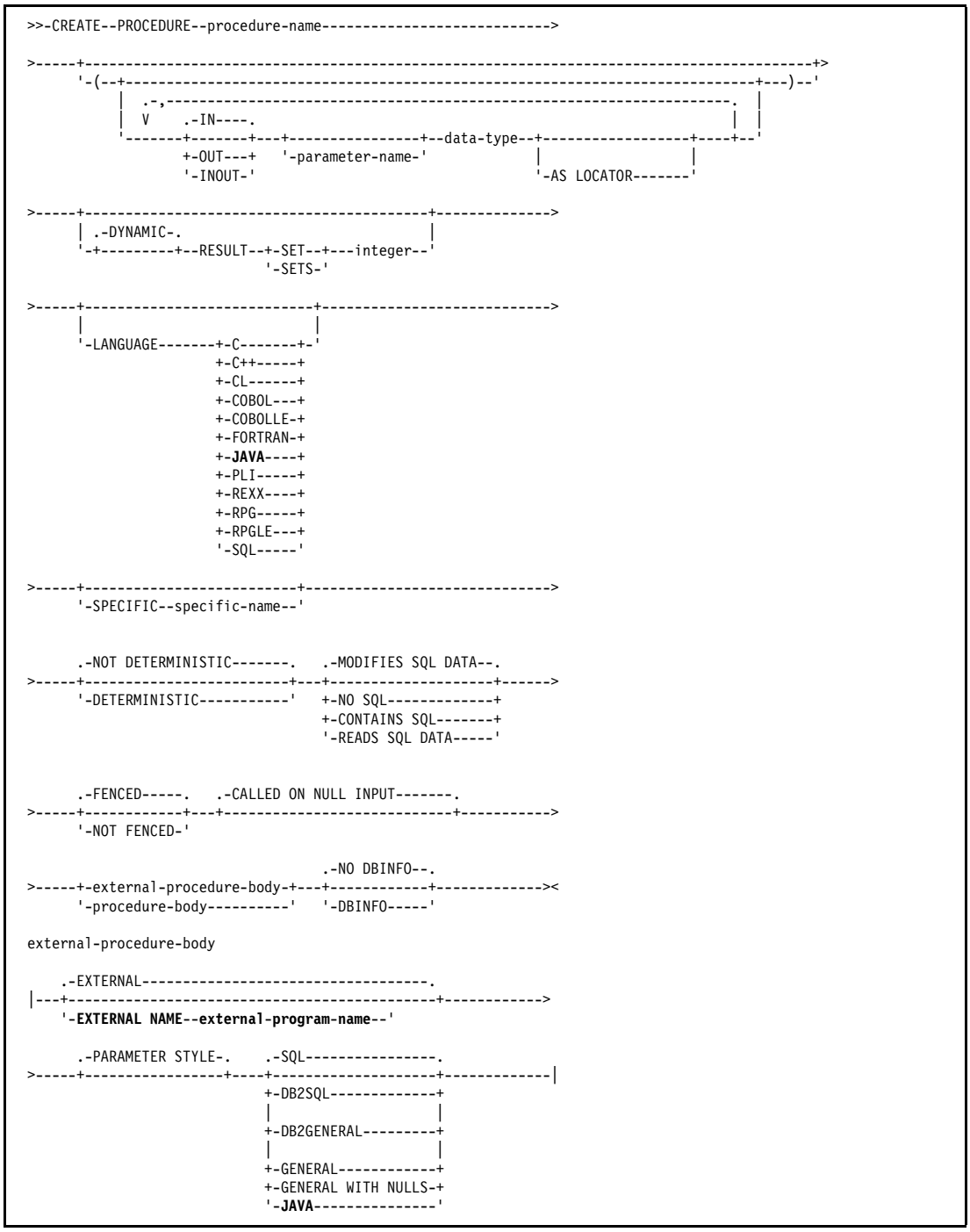


Figure 5-2 CREATE PROCEDURE syntax

The most important options in the syntax in Figure 5-2 on page 108 are described:

- 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 be used only when the Java language is specified.
- EXTERNAL NAME: A Java class and a method can now be specified. They are separated by an exclamation point (!) or by a period (.). The use of the exclamation point (!) was introduced by the DB2GENERAL parameter style. The use of the period (.) is part of the SQLJ standard. For example, you can specify either `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 because all Java stored procedure classes must reside in the `/QIBM/UserData/OS400/SQLLib/Function` directory.

5.4.1 Registering Java stored procedures with System i Navigator

For the JAVA parameter style, the Java stored procedure can be defined with the System i Navigator New External Procedure window. The required steps are listed:

1. In System i 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 5-3.

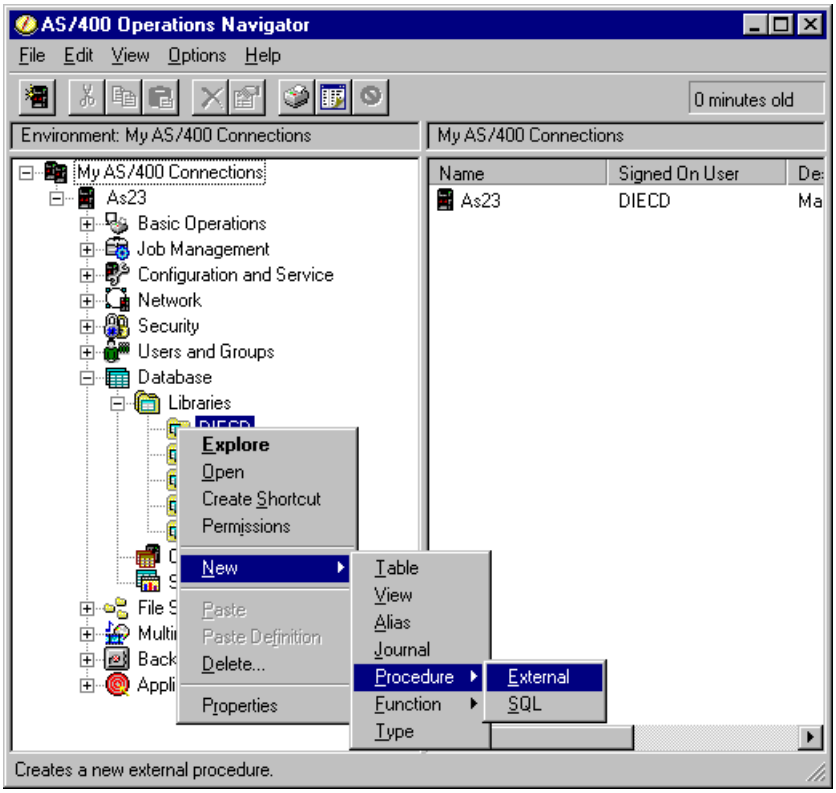


Figure 5-3 Creating a Java stored procedure

2. The New External Procedure window (Figure 5-4) opens. On the General tab, enter the name for the new stored procedure. You can also enter a description and a value for a specific name. Click **OK**.

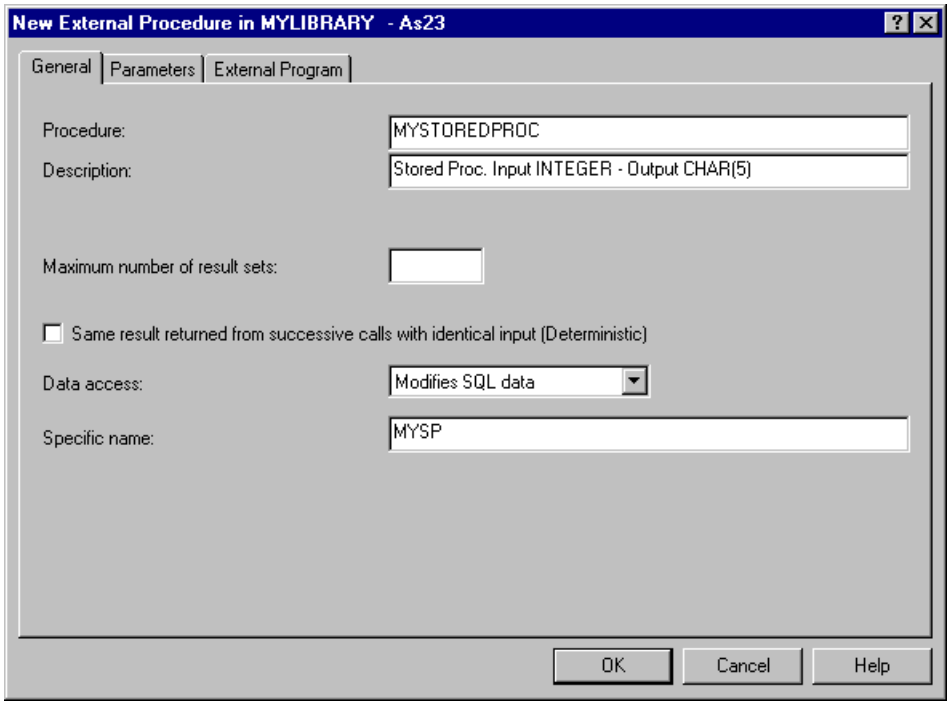


Figure 5-4 Creating a Java stored procedure: The General tab

3. Click the **Parameters** tab. Choose the **Java** parameter style, and click **Insert** to add the parameters, as shown in Figure 5-5.

> **Note:** The DB2GENERAL parameter style Java stored procedures cannot be registered by using System i Navigator.

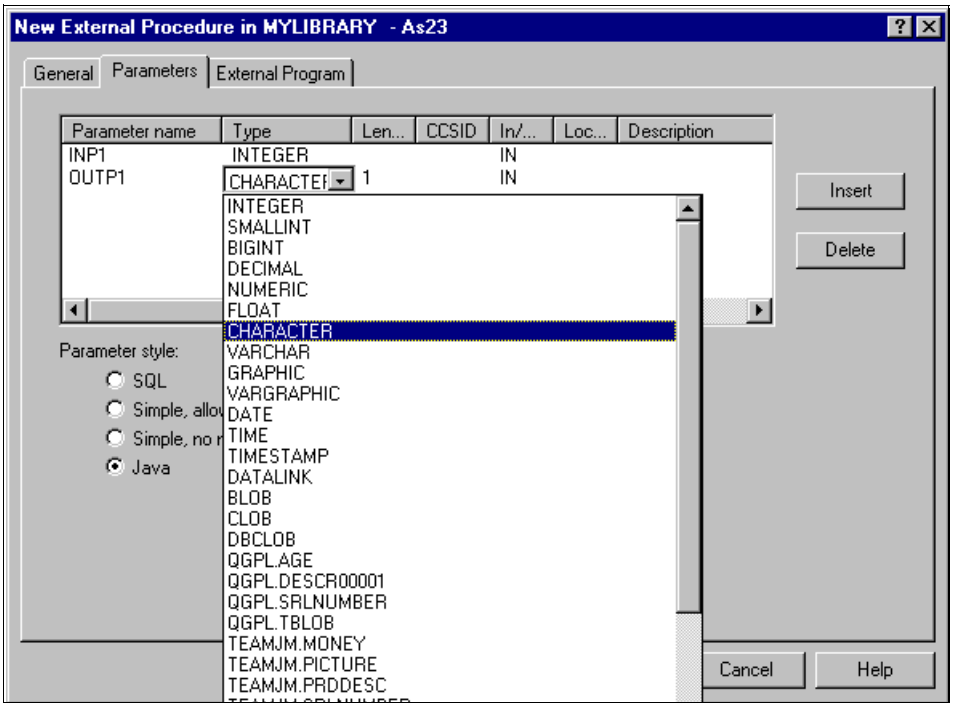


Figure 5-5 Creating a Java stored procedure: The Parameters tab

4. 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 point (!) or a period (.) as indicated in Figure 5-6. Click **OK**.

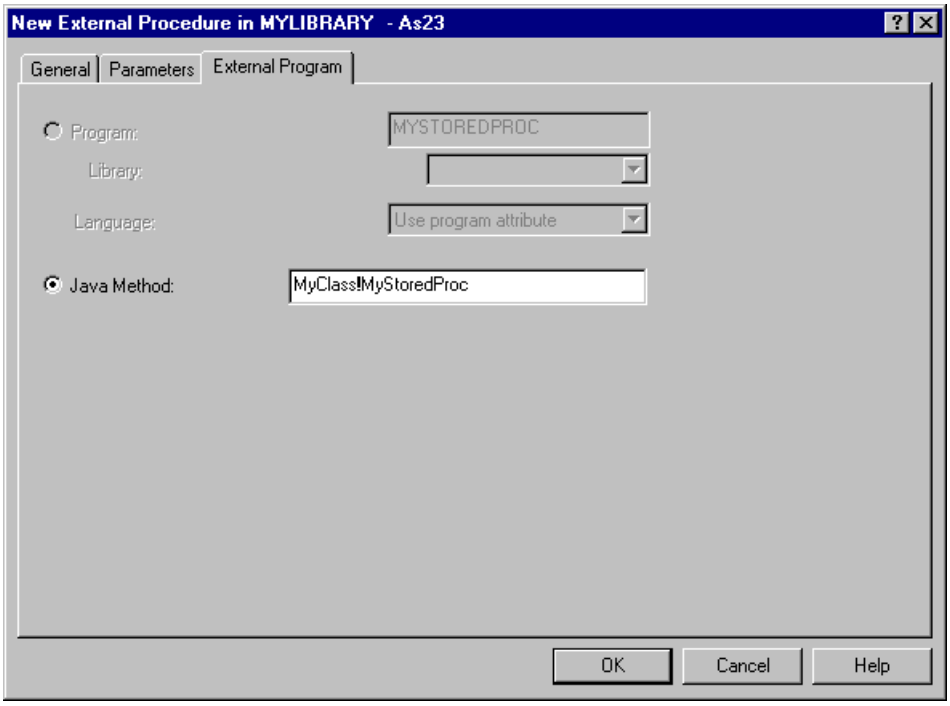


Figure 5-6 Creating a Java stored procedure: The External Program tab

The Java procedure registration is now complete.

5.4.2 Using the Run SQL Scripts utility

Loading the classes into the `Function` directory does not mean that DB2 for i is ready to use the new stored procedures. First, we must register our methods in the Java classes as stored procedures with the CREATE PROCEDURE SQL statement.

In the example in Figure 5-7, we execute these statements in the Run SQL Scripts utility. To access the utility from the main System i Navigator window, right-click **Database**, and select **Run SQL Scripts**.

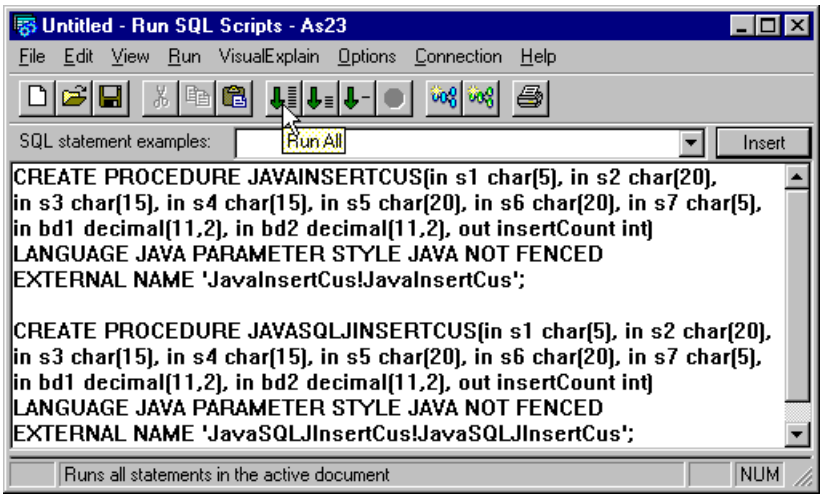


Figure 5-7 Creating the stored procedures in the Run SQL Scripts utility

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.

5.4.3 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:

```
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';
```

To execute the CREATE PROCEDURE statement that is contained in the source file, you can use the following **RUNSQLSTM** CL command:

```
RUNSQLSTM SRCFILE(MYLIBRARY/QSQLSRC) SRCMBR(CRT_SP) COMMIT(*NONE)
```

5.5 Calling Java stored procedures

A Java stored procedure is similar to any other stored procedure, and it 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 affects only the Java method code. From the calling process point of view, no difference exists between an output parameter that is returned by a Java stored procedure and an output parameter that is returned by a stored procedure that is written in other programming languages.

JavaInsertCus and JavaSQLJInsertCus examples

To test our Java stored procedures, we use the client application in Example 5-7 that calls our JDBC and SQLJ Java stored procedures.

Example 5-7 Client application to call JDBC and SQLJ Java stored procedures

```
import java.math.*;                                                    1
import java.sql.*;
import com.ibm.as400.access.*;                                         2

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...\n");
   e.printStackTrace ();
  }
 }
 public static void main (String argv[])
  { Connection con = null;
    CallableStatement ps;
    String sql;
    try
    {   String url = "jdbc:as400://AS400WS                             3
        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 JAVAINSERTCUS (?,?,?,?,?,?,?,?,?,?)";
        ps = con.prepareCall(sql);                                     4
        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");
        ps.setString(6, "Calentina");
        ps.setString(7, "22457");
        ps.setBigDecimal(8, new BigDecimal("2000000"));
        ps.setBigDecimal(9, new BigDecimal("500.5"));
        ps.registerOutParameter(10, Types.INTEGER);                    5
        ps.execute();
```

```
         System.out.println("There was " + ps.getInt(10) + " record inserted.");

        sql = "Call JAVASQLJINSERTCUS (?,?,?,?,?,?,?,?,?,?)";            6
        ps = con.prepareCall(sql);
        ps.setString(1, "00004");
        ps.setString(2, "David Rosenborrow");
        ps.setString(3, "xx-00-73-124578");
        ps.setString(4, "xx-00-73-123456");
        ps.setString(5, "Main Street");
        ps.setString(6, "Calentina");
        ps.setString(7, "22457");
        ps.setBigDecimal(8, new BigDecimal("123456789.11"));
        ps.setBigDecimal(9, new BigDecimal("987654321.99"));
        ps.registerOutParameter(10, Types.INTEGER);
        ps.execute();
         System.out.println("There was " + ps.getInt(10) + " record inserted.");

        if (ps != null) ps.close();
        if (con != null) con.close();
    }
    catch (Exception e)
      { e.printStackTrace ();
      }
  }
}
```

Notes: The following notes refer to the numbers in Example 5-7 on page 114:

1. The `java.math.*` classes are needed for the BigDecimal Java type that corresponds to the SQL DECIMAL type.
2. These classes contain the toolbox JDBC driver.
3. The URL to connect to the IBM i server where the stored procedures were defined.
4. The Java stored procedure is called with a callable statement like any other stored procedure. The caller is not aware that the stored procedure is written in Java.
5. The internal definition as an array for the output parameter is transparent to the caller.
6. A similar call is used for the SQLJ type Java stored procedure.

The output of the client is shown:

```
  Java Stored Procedure Sample
There was 1 record inserted.
There was 1 record inserted.
```

The output indicates that the stored procedures were successfully executed.

DB2CusInCity, DB2SQLJCusInCity, and DB2SQLJCusInCity2 examples

Our same client application calls the new stored procedures that illustrate the DB2GENERAL parameter style, as shown in Example 5-8.

Example 5-8 New stored procedures that illustrate the DB2GENERAL parameter style

```
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...\n");
   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.
```

5.6 Using SQL NULL

The theory that concerns SQL NULL values was presented in 5.2, “Coding DB2 for i Java stored procedures” on page 93. This section compares two stored procedures by 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, which is called T, that is 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 5-9.

Example 5-9 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)                                                    1
      {ps.setNull(1, Types.CHAR);}                                     2
     else {ps.setString(1, s);}
     ps.setInt(2, i);                                                  3
     ps.executeUpdate();
     try {if (ps != null) ps.close();
          if (con != null) con.close();
     } catch (SQLException e) { /* ignore */ };
    }
}
```

Notes: The following notes refer to the numbers in Example 5-9 on page 117:

1. With the JAVA parameter style, we test whether an object instance (“s”) has a null reference to know whether the passed parameter was an SQL NULL value.
2. If an SQL NULL value was passed, we use the `setNull` method to prepare our insert SQL statement.
3. A null integer cannot be passed to a stored procedure by using the JAVA parameter style. If a null integer is passed to a stored procedure, SQL error code -20205 is signaled to the caller.

Example 5-10 shows the code of DB2SPNULL.

Example 5-10 DB2SPNULL code

```
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))                                                    1
      {ps.setNull(1, Types.CHAR);}                                     2
     else {ps.setString(1, s);}
     if (isNull(2))                                                    3
      {ps.setNull(2, Types.INTEGER);}                                  4
     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 the numbers in Example 5-10:

1. With the DB2GENERAL parameter style, we test whether a parameter has an SQL NULL value with the `isNull` method that is 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 described in the JAVA parameter style.
3. We use the `isNull` method to check whether the integer parameter has an SQL NULL value.
4. We can insert a NULL value for the integer parameter, which is impossible with the JAVA parameter style.

After we create the two stored procedures that correspond to the two Java methods, we call them with the SQL statements in the Interactive SQL session, as shown in Figure 5-8. 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

Type SQL statement, press Enter.
   > CREATE TABLE T(S CHAR (5 ), I INTEGER )
     Table T created in DIECD.
   > call javaspnull(NULL, 0)
     CALL statement complete.
   > call db2spnull(NULL, NULL)
     CALL statement complete.
   > call javaspnull(NULL, NULL)
===> select s,i from t
```

Figure 5-8 Passing NULL values to the Java stored procedures

We receive the output that is shown in Figure 5-9 that confirms the previous comments.

```
                                  Display Data
                                              Data width . . . . . . :      21
 Position to line  . . . . .               Shift to column  . . . . . .
 ....+....1....+....2.
 S                  I
 -                  0
 -                  -
 ********  End of data  ********
```

Figure 5-9 NULL values were inserted by the Java stored procedures

5.7 SQLJ procedures to manipulate JAR files

Java stored procedures (and 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 with 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.

We briefly explain the following stored procedures:

- SQLJ.INSTALL_JAR
- SQLJ.REPLACE_JAR
- SQLJ.REMOVE_JAR
- SQLJ.UPDATEJARINFO
- SQLJ.RECOVERJAR

To use these stored procedures, we need INSERT, UPDATE, DELETE, and SELECT privileges for the SYSJAROBJECTS and SYSJARCONTENTS catalog tables and *EXECUTE authority on library QSYS2. We also need the correct authorities to the affected JAR file and the /QIBM/UserData/OS400/SQLLib/Function/jar/*schema* directory, where *schema* is the schema of the *jar-id*. For more information, see the *SQL Programming Guide*, SC41-5611. Adopted authority cannot be used for these authorities.

5.7.1 SQLJ.INSTALL_JAR

The SQLJ.INSTALL_JAR stored procedure installs a JAR file in DB2 for i. This JAR file can be used in subsequent CREATE PROCEDURE statements.

The SQLJ.INSTALL_JAR stored procedure requires three parameters:

- **jar-url**: The URL that contains the JAR file to be installed or replaced. The only URL schema that is supported is “file:”.
- **jar-id**: The JAR identifier in the database to be associated with the file that is specified by the **jar-url**. The **jar-id** will use SQL naming, and the JAR file will be installed in the schema or library that is specified by the implicit or explicit qualifier.
- **deploy**: The value that is used to describe the install_action of the deployment descriptor file. If this integer is a nonzero value, the install_action of a deployment descriptor file must be executed at the end of the install_jar procedure. The current version of DB2 for i supports a value of zero only.

When a JAR file is installed, DB2 for i 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 for i copies the JAR file to a `jar/schema` subdirectory of the `/QIBM/UserData/OS400/SQLLib/Function` directory. DB2 for i gives the new copy of the JAR file the name that is in the **jar-id** clause.

Important: A JAR file that was installed by DB2 for i in a subdirectory of `/QIBM/UserData/OS400/SQLLib/Function` must not be modified. Instead, remove or replace an installed JAR file.

Example

We create a sample JAR file by using the following command in the Qshell interface that is 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 that is in the `/home/dlema/JavaTest` directory is installed in DB2 for i 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 see the MYSAMPLE_JAR file. The `.../jar/DLEMA` directory was automatically created by the SQLJ.INSTALL_JAR stored procedure.

If we examine the contents of QSYS2.SYSJAROBJECTS, we can see our recently installed JAR file, as shown in Figure 5-10.

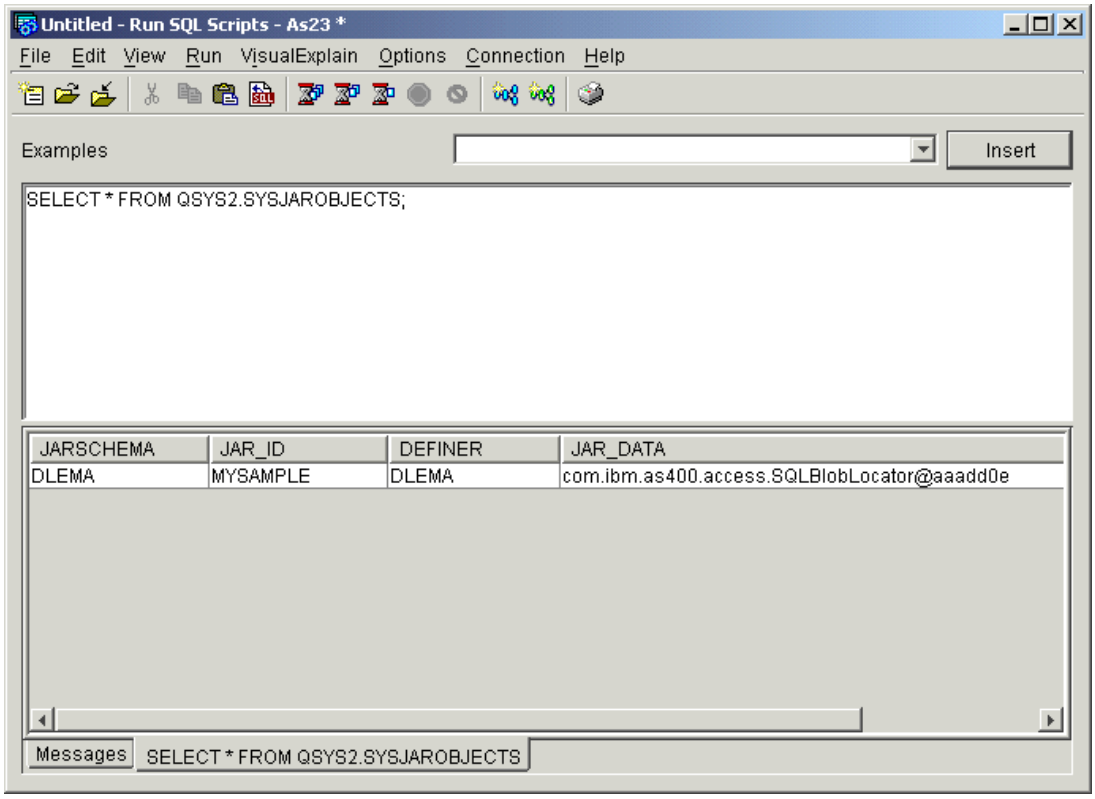


Figure 5-10 QSYS2.SYSJAROBJECTS contents after the installation of the MYSAMPLE JAR file

The QSYS2.SYSJARCONTENTS table has a row for each class inside the installed JAR files, as shown in Figure 5-11.

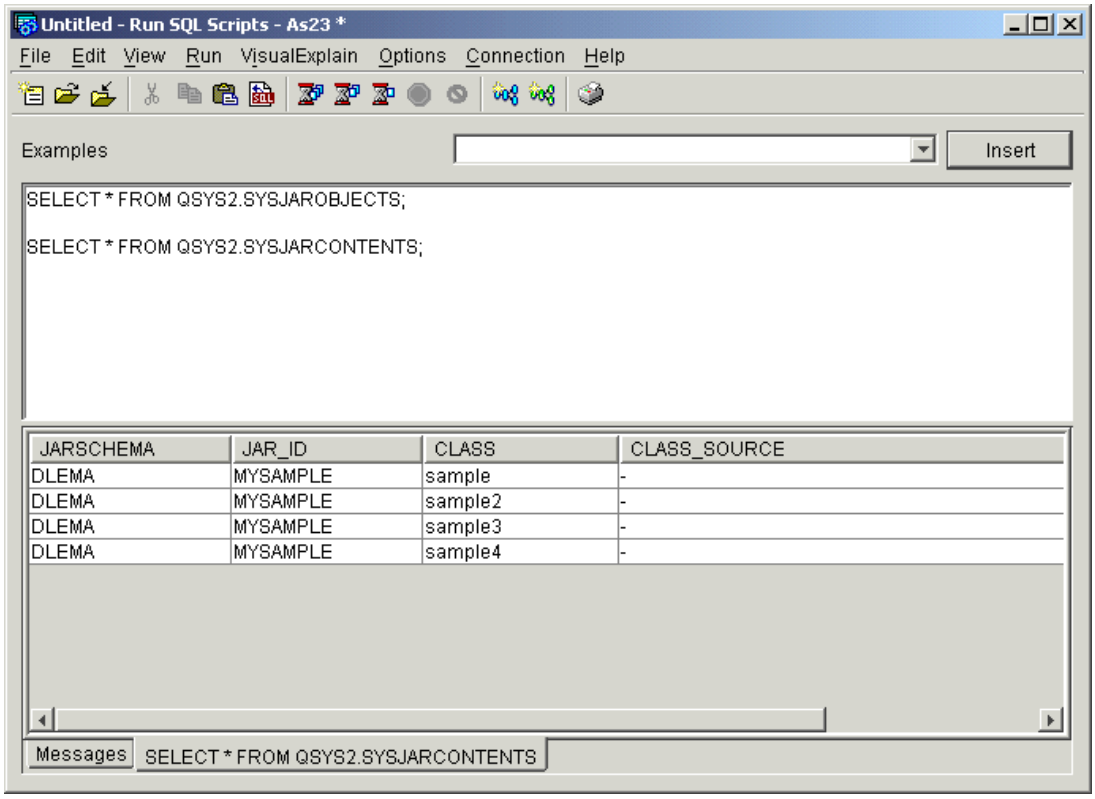


Figure 5-11 QSYS2.SYSJARCONTENTS after the installation of the MYSAMPLE JAR file

5.7.2 SQLJ.REMOVE_JAR

The SQLJ.REMOVE_JAR stored procedure removes a JAR file into DB2 for i. The SQLJ.REMOVE_JAR stored procedure requires two parameters:

- **`jar-id`**: The JAR identifier that is associated with the JAR file to remove from the database.
- **`undeploy`**: The value that is used to describe the install_action of the deployment descriptor file. If this integer is a nonzero value, the remove_actions of a deployment descriptor file must be executed at the end of the install_jar procedure. The current version of DB2 for i supports a value of zero only.

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.

5.7.3 SQLJ.REPLACE_JAR

The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into DB2 for i. The SQLJ.REPLACE_JAR stored procedure requires two parameters:

- **jar-url**: The URL that contains the JAR file to replace. The only URL schema supported in “file:”.
- **jar-id**: The JAR identifier that is associated with the JAR file to remove 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 that is referred to by the **jar-id** `mysample_jar` is replaced with the new `mysample.jar` file in the specified directory.

5.7.4 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 for i stored procedure builder.

The SQLJ.UPDATEJARINFO stored procedure requires three parameters:

- **jar-id**: The JAR identifier to update.
- **class-id**: The package qualified class name of the class to update.
- **jar-url**: The URL that contains the JAR file with which to update the JAR file. The only URL schema that is 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 that is 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`.

5.7.5 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/`*`schema`*`/`*`jar_id.jar`* file.

The SQLJ.RECOVERJAR stored procedure requires the **jar-id** parameter, which is the JAR identifier to recover.

Example

The following command is issued from an SQL interactive session:

```
CALL SQLJ.RECOVERJAR('mysample_jar')
```

The JAR file that is 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.

5.8 Additional considerations

This section focuses on important topics that are specific to the current implementation of Java stored procedures on the IBM i server.

General considerations are listed:

- *Built-in functions*: The built-in functions that are described in the SQLJ routines standard were introduced in V5R1. SQLJ.ALTER_JAR_PATH is not supported yet.
- *Result set*: Result sets are supported as of V5R1.
- *Adopted authority*: The authority of the current user is used. No support exists for the concept of adopted authority.
- *Threads*: Java stored procedures must not create additional threads.
- *Java Developer's Kit (JDK)*: Java stored procedures automatically use the latest version of the Java Developer's Kit that is installed on the IBM i 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. When the character is not stored in Unicode, a character conversion must be performed on any input and output of character data between Java and the database. This conversion affects performance, but the effect can be avoided by storing the character data directly in Unicode coded character set identifier (CCSID) 13488.

Consider two important consequences before you use this approach. If the character data is stored in Unicode, a character conversion is required for the applications that do not manipulate the data in Unicode. The Unicode has double-byte encoding so that two bytes are required per character in place of one byte for a single-byte character set. For large tables, a significant increase in DASD requirements occurs. The performance of a Java application might improve when the character data is stored in Unicode. In this case, consider changing the character data to Unicode only when the data is exclusively or almost exclusively used by JDBC applications and when the additional DASD requirements are not a concern.

5.8.1 Moving into production (save and restore)

While you deploy 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 implement that stored procedure, additional actions might be required to make the stored procedure and external program available on the target system.

For Java stored procedures, you can save the `.class` or `.jar` files that implement Java stored procedures to a save file or any other media and then restore it to the production system in the `/QIBM/UserData/OS400/SQLLib/Function` library. The Java code that is 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 re-create the procedure with the CREATE PROCEDURE statement because the system catalog is not updated automatically. Therefore, a CREATE PROCEDURE must be performed for each restored Java stored procedure.

For performance reasons, you also want to perform a CRTJVAPGM with OPTIMIZE(40).

5.9 GetSuppliers example: Implementation with no result sets

The 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.

Because the result sets in Java stored procedures are supported in V5R1, in 5.10, “GetSupplierRS example: Implementation with result sets” on page 132, we illustrate how the Java stored procedure can return the result sets to the calling process.

5.9.1 Stored procedure: GetSupplier

The business logic of this example is described in 4.4.1, “Coding external stored procedures that return cursor result sets” on page 61. The stored procedure returns two result sets containing the list of *n* best and *n* worst suppliers and their total sales amount for a specific 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 5-11.

Example 5-11 GetSupplier Java class

```
import java.sql.*;

public class GetSupplier
{public static void GetSupplier (int year, int month, int[] rank, String[] suppliers )    1
 throws SQLException, Exception
    {Connection con = DriverManager.getConnection("jdbc:default:connection");
     PreparedStatement ps = null;
     ResultSet rs = null;
     String sql;
     int rowCount;
     suppliers[0] = "";

     // best suppliers                                                                      2
     if (month < 1)                                                                         3
     {
       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;
      while ((rs.next()) && (rowCount<rank[0]))
      {
        suppliers[0] = suppliers[0] + rs.getString(1).trim() + "/" +
            rs.getBigDecimal(2, 0) + "/";                                    4
        rowCount++;                                                          5
      }
      if (rowCount != rank[0]) {rank[0] = rowCount;}
      try {if (rs != null) {rs.close();}
      } catch (SQLException e) { /* ignore */ };

      // worse suppliers                                                     6
      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 the numbers in Example 5-11 on page 125:

1. The *suppliers* is the new output parameter that is used to return the concatenation of all suppliers and their corresponding total sales amount. The *rank* is an inout parameter that passes the requested number of the suppliers. It returns the actual number of suppliers that is 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 of `1 - 12` if we want the data for a specific 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 IBM i server.

Stored procedure creation

We register our stored procedure with the following CREATE PROCEDURE SQL statement:

```
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 5.9.3, “Java GUI client: ClientGetSupplierGUI” on page 132, we describe the Java clients that are used to call this Java stored procedure. ClientGetSupplier is a text version, while ClientGetSupplierSwing is a graphical user interface (GUI) version of the Java client.

5.9.2 Java client: ClientGetSupplier

This client application is intended to call the GetSupplier stored procedures that are written in any language on both DB2 for i and DB2 Universal Database on other platforms. This client application 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 5-12.

Example 5-12 ClientGetSupplier client code

```
import java.math.*;
import java.util.*;
import java.io.*;
import java.sql.*;
import COM.ibm.db2.jdbc.app.*;                                          1
import com.ibm.as400.access.*;                                          2

class ClientGetSupplier
{ public static void main (String argv[])
  {Properties props = new Properties();
   Connection con = null;
   CallableStatement cs;
   int rankCount;
   boolean returnsRS;
```

```
String sql;
String suppliers = "";
try
 {props.load(new BufferedInputStream(new FileInputStream("logon.properties")));        3
 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;}                                                            4
 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               5
 { sql = "Call " + storedProc + " (?,?,?)";
   cs = con.prepareCall(sql);
   cs.setInt(1, year);
   cs.setInt(2, month);
    cs.setInt(3, rank);                                                                6
    cs.registerOutParameter(3, Types.INTEGER);                                         6
   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                   7
   if (brs != null)
   { System.out.println("The best suppliers are :");
     System.out.println("-----------------------");
       cursor = brs.next();                                                            8
     while (cursor)
     { System.out.println(brs.getString(1).trim() + " with a total sale of "
+ brs.getBigDecimal(2, 0));                                                            9
       cursor = brs.next();
     }
     cs.getMoreResults();                                                             10
     ResultSet wrs = cs.getResultSet(); // the worst suppliers result set            11
     if (wrs != null)                                                                12
// or we could test for UpdateCount() = -1 and MoreResult = false
     { System.out.println();
       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));                                                          13
            cursor = wrs.next();
          }
        }
        else {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                         14
    { 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);                                    15
      cs.execute();
      rank = cs.getInt(3);
       suppliers = cs.getString(4);                                                  15
      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]);                                              17
      }
      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]);                                    17
      }
    }

    if (cs != null) cs.close();
    if (con != null) con.close();
   }
   catch (Exception e)
   { e.printStackTrace (); }
  }

  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 numbered notes refer to the numbers in Example 5-12 on page 127:

1 and **2** The client can access both DB2 for i and DB2 Universal Database on other platforms. The classpath must contain the toolbox driver for the IBM i server (`jt400.jar`) and the `db2java.zip` classes for DB2 Universal Database, as presented in 5.10, “GetSupplierRS example: Implementation with result sets” on page 132. If you want to access only one of the two platforms, you can omit the corresponding import statement.

3 The application reads the required information to connect to the DB2 platform and to call the stored procedure from the properties file that is called `logon.properties`. The properties dbDriver, dbUrl, dbUser, and dbPassword are used to connect to the database by 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. The returnsResultSet property is set to No for the Java stored procedure on the IBM i server that returns the suppliers output parameter that contains the results. The last three properties are the input parameters that are 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 differentiate between the stored procedure that returns two result sets and the stored procedure that returns the suppliers parameter instead.

6 When we call 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. 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 that is 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 that is 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. An SQL decimal data type is mapped to the BigDecimal Java type. Also, we do not 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 that was 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 If the `getMoreResults()` 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 result set.

14 In this second part of the application, we handle the IBM i server Java stored procedures that return a parameter that contains the concatenated string that represents 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.

16 Each element of the String is separated from the next element 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.

17 We display the elements from this array.

With our classpath pointing to the driver implementations that are referred to in **1** and **2** and with the `logon.properties` file in our current directory, we compile and execute the `ClientGetSupplier` class. The results are shown:

```
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.

5.9.3 Java GUI client: ClientGetSupplierGUI

The Java GUI client that is presented in this section interacts with the user through a visual interface. We can enter the information that was placed in the `logon.properties` file in the previous section and display the results that are returned by the stored procedure. The GUI interface is similar to that of the Visual Basic client, and it is not presented in detail. It was developed with IBM VisualAge® for Java (JDK 1.1.8).

Figure 5-12 shows how it works.

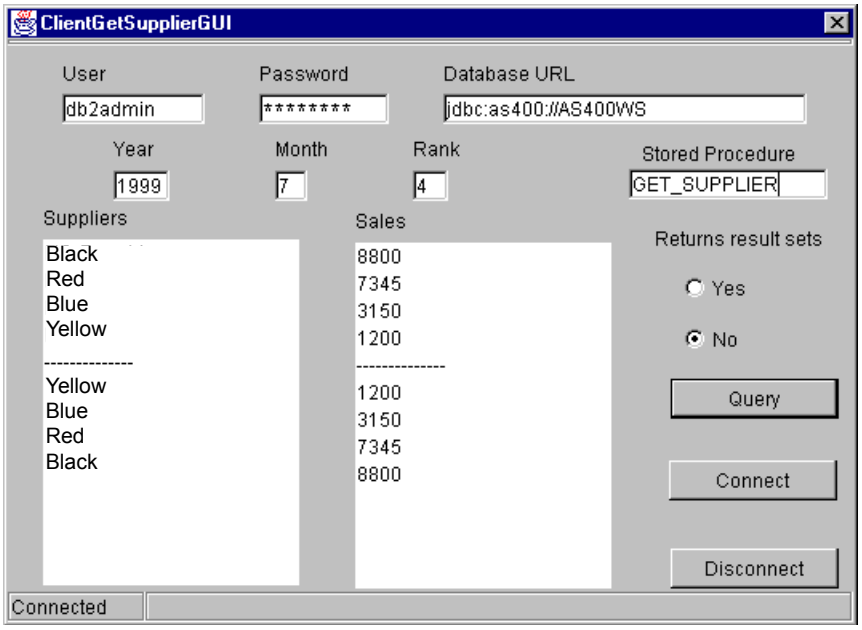


Figure 5-12 ClientGetSupplierGUI displays the IBM i server Java stored procedure results

This same GUI client is used to call the external SQL and Java stored procedures that were created on the IBM i server and the stored procedures that were created on DB2 Universal Database V6.1. We first enter the user, password, and JDBC URL to connect to the database that we want. When we are connected, we enter the stored procedure that we want to execute and its parameters. Then, we call it by clicking **Query**.

5.10 GetSupplierRS example: Implementation with result sets

A Java stored procedure on IBM i server V5R1 or DB2 Universal Database Version 6 and later can return result sets. In this section, we present a GetSupplierRS Java stored procedure that returns two result sets: the best suppliers and the worst suppliers.

5.10.1 GetSupplierRS stored procedure with the JAVA parameter style

Example 5-13 shows the GetSupplierRS stored procedure with the JAVA parameter style. The numbered lines are explained in the following notes.

Example 5-13 GetSupplierRS stored procedure with the JAVA parameter style

```
import java.math.*;
import java.sql.*;

public class GetSupplierRS
{public static void GetSupplierRS (int year, int month, int[] rank,
 ResultSet[] bestSuppliers, ResultSet[] worstSuppliers)                    1
 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                   2
     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                  3
     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)                                                          4
    {
      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();                                   5
    if (month < 1)                                                          6
    {
      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();                                 7
   }
}
```

Notes: The following numbered notes refer to the numbers in Example 5-13 on page 133:

1. In the Java language, a result set is simply an instance of the `java.sql.ResultSet` class. A result set is considered to be similar to another variable. When a result set is returned by a Java stored procedure by 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 that was achieved by the *n*th best and the *n*th worst suppliers. Afterward, we retrieve two result sets. The first result set contains the suppliers with `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 with `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 to get the second result set that returns 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 returns result sets only if FENCED is used in the CREATE PROCEDURE SQL statement. The specification of NOT FENCED prevents the stored procedure from returning any result set, but no error message is issued as a warning. In DB2 for i, this parameter is provided only for compatibility with other platforms, and it has no effect.

5.10.2 GetSupplierRS stored procedure with the DB2GENERAL parameter style

Example 5-14 shows the GetSupplierRS stored procedure with the DB2GENERAL parameter style. The numbered lines are explained in the following notes.

Example 5-14 GetSupplierRS stored procedure with the DB2GENERAL parameter style

```
import java.math.*;
import java.sql.*;
import com.ibm.db2.app.*;                                              1

public class GetSupplierResultSetDB2GENERAL extends StoredProc         2
{
  public void GetSupplierRS (int year, int month, int rank)            3
  throws SQLException, Exception
  {
     Connection con = getConnection();                                 4
```

```
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                    5
   }
}
```

Notes: The main differences from the JAVA parameter style version are listed. The numbered notes refer to the numbers in Example 5-14 on page 135:

1. Import file that contains the `StoredProc` superclass.
2. A stored procedure class that contains DB2GENERAL stored procedures must extend the `com.ibm.db2.app.StoredProc` class.
3. Methods that conform to DB2GENERAL stored procedures are not static. IN, OUT, and INOUT parameters are defined as an input parameter (no array convention used).
4. Connection is established by the `getConnection()` method that is inherited from the `StoredProc` class.
5. OUT and INOUT parameters are returned by the `set` method that is 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 returns result sets only if FENCED is used in the CREATE PROCEDURE SQL statement. The specification of NOT FENCED prevents the stored procedure from returning any result set, but no error message is issued as a warning. In DB2 for i, this parameter is provided only for compatibility with other platforms, and it has no effect.

5.10.3 Java clients: ClientGetSupplier and ClientGetSupplierGUI

We use the Java clients that are presented in 5.9.2, “Java client: ClientGetSupplier” on page 127 and 5.9.3, “Java GUI client: ClientGetSupplierGUI” on page 132 to test whether they work with the Java stored procedure that was created both on DB2 Universal Database V6.1 and on DB2 for i. The result is displayed in Figure 5-13.

As shown in Figure 5-13, the same client can be used to call stored procedures that are written in different languages and on different platforms. The same satisfactory results were also obtained with the text version ClientGetSupplier.

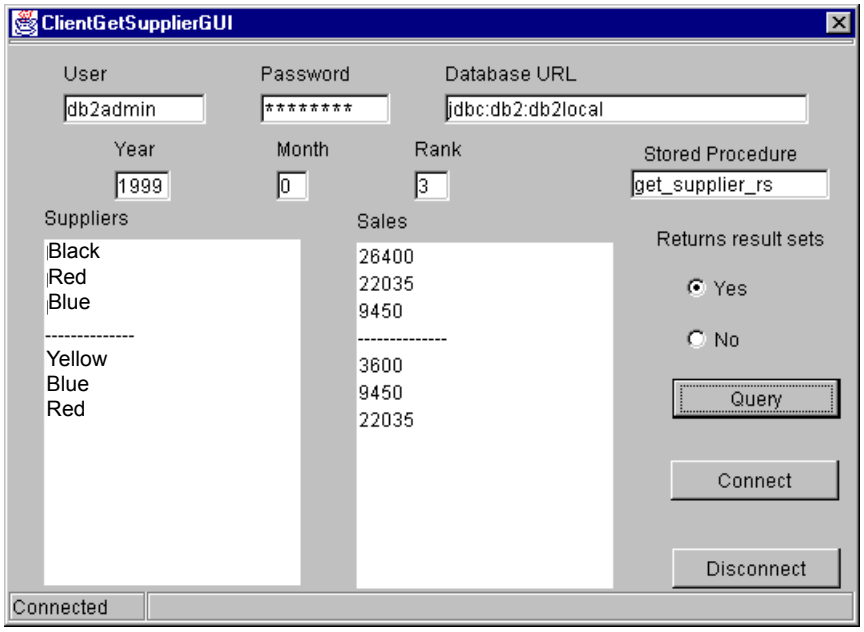


Figure 5-13 Results of the Java stored procedure on DB2 Universal Database V6.1

5.11 Problem determination

This section focus on the debugging and the tracing possibilities of Java stored procedures.

5.11.1 Debugging

At first glance, debugging might seem slightly complicated because it requires the Java virtual machine (JVM) to be started when you try to access the source. We learned that the easiest way to debug was in a client/server environment. The following steps outline the process that we 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 IBM i server Function directory `/Qibm/UserData/OS400/SQLLib/Function`.

3. Identify the full name of the job where the Java stored procedure will be executed. If you use the Java toolbox to connect to the IBM i server, the job is QZDASOINIT. The easiest way to find the correct job is to use the command:

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

   Here, *USERID* is the user ID that was used to connect to the IBM i server. Document the job number and user profile.

4. Start to service the job that you identified in step 3, for example, run this command:

   ```
   STRSRVJOB JOB(076853/QUSER/QZDASOINIT)
   ```

5. *Ensure* that the JVM was started in this job. For example, the client can call a dummy Java stored procedure that does nothing and then wait until you set up the debugging.

6. Debug for the class that you want, for example:

   ```
   STRDBG CLASS(myClass)
   ```

7. If everything is alright, you now see the Java source file on your 5250 emulation. You can step through it and add breakpoints in the same way that add them with any other traditional language.

Figure 5-14 shows the added breakpoint.

```
                          Display Module Source

Class file name:   spbjavasp2b
     1  /**
     2   * JDBC Stored Procedure SPBJAVASP2B
     3   */
     4  import java.sql.*;                   // JDBC classes
     5
     6  public class spbjavasp2b
     7  {public static void spbjavasp2b ( int i ) throws SQLException, Excepti
     8      {// Get connection to the database
     9          Connection con = DriverManager.getConnection("jdbc:default:con
    10          PreparedStatement stmt = null;
    11          int updateCount;
    12          String sql;
    13          sql = "UPDATE T SET I =  ?";
    14          stmt = con.prepareStatement( sql );
    15          stmt.setInt( 1, i );
                                                                        More...
Debug . . .

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
Breakpoint added to line 14.
```

Figure 5-14 Java stored procedure debugging

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 for a key to be pressed before it calls the stored procedure.
9. On the 5250 emulation, the debug panel displays, and you can now debug your Java code.
10. After the debugging is completed, stop your debug and service sessions by using these two CL commands:

    ENDDBG
    ENDSRVJOB

5.11.2 Tracing

The trace level component can be used to trace the actions that are performed by the OS/400 module that is responsible for the Java stored procedures support. However, its interpretation can be difficult and needs to 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 you work 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 the trace and output it to a spooled file before the `QTEMP` library is cleaned. Now, we can easily call the first stored procedure to enable the tracing, then call the Java stored procedure that we want to trace. Then, we call the last stored procedure to stop the tracing and obtain the spooled file.

The code that makes up the SPTRACEON procedure is shown:

```
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:

```
PGM
DMPUSRTRC
CPYF FROMFILE(QTEMP/QAPOZDMP) 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';
```

Note: The **`insertCount`** parameter (highlighted in bold in the preceding code) in the Java method is defined as an integer, but the CREATE PROCEDURE statement executes successfully.

At execution time, the SQL interface first checks whether the parameters that were passed by the SQL CALL match those parameters that were declared by the CREATE PROCEDURE statement. If yes, 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 about 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 it looked for the `ConvertUnicode` method with a specific signature in the `ConvertUnicode` class.

The trace is partially shown in Figure 5-15 and is further explained in the following notes.

```
 70     0000001A:042576 SQLT_SQLEJ.SQLT_SQLEJ_CALLSTP_DLL: code 60 description JNI GetMethodID
failed.  class:    1
 71     0000001A:042624 SQLT_SQLEJ.SQLT_SQLEJ_CALLSTP_DLL: 13 bytes of internal data
 72     0000001A:042688 E733F80762:C343F0 L:000D Buffer Data EBCDIC
 73     0000001A:042800  E733F80762:C343F0  4A617661 496E7365 72744375 73          *¢/./.>.........*
 74     0000001A:042896 E733F80762:C345F0 L:000D Buffer Data ASCII
 75     0000001A:043000  E733F80762:C345F0  D181A581 C995A285 99A3C3A4 A2          *JavaInsertCus...*2
 76     0000001A:043056 SQLT_SQLEJ.SQLT_SQLEJ_CALLSTP_DLL: code 61 description JNI GetMethodID failed.  method:3
 77     0000001A:043096 SQLT_SQLEJ.SQLT_SQLEJ_CALLSTP_DLL: 13 bytes of internal dat
 78     0000001A:043168 E733F80762:C343FE L:000D Buffer Data EBCDIC
 79     0000001A:043240  E733F80762:C343F0                                4A61     *.............¢/*
 80     0000001A:043344  E733F80762:C3440  7661496E 73657274 437573               *./.>...........*
 81     0000001A:043440 E733F80762:C3463 L:000D Buffer Data ASCII
 82     0000001A:043544  E733F80762:C3463  D181A581 C995A285 99A3C3A4 A2          *JavaInsertCus...*4
 83     0000001A:043600 SQLT_SQLEJ.SQLT_SQLEJ_CALLSTP_DLL: code 62 description JNI GetMethodID failed. signature:5
 84     0000001A:043648 SQLT_SQLEJ.SQLT_SQLEJ_CALLSTP_DLL: 192 bytes of internal data
 85     0000001A:043712 E733F80762:C3442 L:00C0 Buffer Data EBCDIC
 86     0000001A:043824  E733F80762:C3442  284C6A61 76612F6C 616E672F 53747269     *.<¦/./.%/>.....*
 87     0000001A:043944  E733F80762:C3443  6E673B4C 6A617661 2F6C616E 672F5374     *>..<¦/./.%/>....*
 88     0000001A:045656  E733F80762:C3444  72696E67 3B4C6A61 76612F6C 616E672F     *..>..<¦/./.%/>..*
 89     0000001A:045768  E733F80762:C3445  53747269 6E673B4C 6A617661 2F6C616E     *....>..<¦/./.%/>*
 90     0000001A:045888  E733F80762:C3446  672F5374 72696E67 3B4C6A61 76612F6C     *......>..<¦/./.%*
 91     0000001A:046000  E733F80762:C3447  616E672F 53747269 6E673B4C 6A617661     */>.....>..<¦/./*
 92     0000001A:046112  E733F80762:C3448  2F6C616E 672F5374 72696E67 3B4C6A61     *.%/>.....>..<¦/*
 93     0000001A:046232  E733F80762:C3449  76612F6C 616E672F 53747269 6E673B4C     *./.%/>.....>..<*
 94     0000001A:046344  E733F80762:C344A0  6A617661 2F6D6174 682F4269 67446563     *¦/./._/.........*
 95     0000001A:046464  E733F80762:C344B0  696D616C 3B4C6A61 76612F6D 6174682F     *._/%.<¦/./._/...*
 96     0000001A:046576  E733F80762:C344C0  42696744 6563696D 616C3B5B 4C6A6176     *......._/%.$<¦/.*
 97     0000001A:046688  E733F80762:C344D0  612F6C61 6E672F53 7472696E 673B2956     */.%/>.....>....*
 98     0000001A:046792 E733F80762:C3467 L:00C0 Buffer Data ASCII
 99     0000001A:046904  E733F80762:C3467  4DD39181 A5816193 81958761 E2A39989     *(Ljava/lang/Stri*6
100     0000001A:047016  E733F80762:C3468  95875ED3 9181A581 61938195 8761E2A3     *ng;Ljava/lang/St*
101     0000001A:047136  E733F80762:C3469  99899587 5ED39181 A5816193 81958761     *ring;Ljava/lang/*
102     0000001A:047248  E733F80762:C346A0  E2A39989 95875ED3 9181A581 61938195     *String;Ljava/lan*
103     0000001A:047368  E733F80762:C346B0  8761E2A3 99899587 5ED39181 A5816193     *g/String;Ljava/l*
104     0000001A:047480  E733F80762:C346C0  81958761 E2A39989 95875ED3 9181A581     *ang/String;Ljava*
105     0000001A:047592  E733F80762:C346D0  61938195 8761E2A3 99899587 5ED39181     */lang/String;Lja*
106     0000001A:047712  E733F80762:C346E0  A5816193 81958761 E2A39989 95875ED3     *va/lang/String;L*
107     0000001A:047824  E733F80762:C346F0  9181A581 619481A3 8861C289 87C48583     *java/math/BigDec*
108     0000001A:047944  E733F80762:C3470  89948193 5ED39181 A5816194 81A38861     *imal;Ljava/math/*
109     0000001A:048056  E733F80762:C3471O  C28987C4 85838994 81935EBA D39181A5     *BigDecimal;[Ljav*7
```

Figure 5-15 Component level trace of a Java stored procedure

Notes: The following notes refer to the numbers in Figure 5-15:

1 and **2** Indicate that a problem occurred in 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 that is described in **6** through **7** and cannot be found.

6

Stored procedure error handling

We can see error handling from two different, but complementary, 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 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 that are shown in this chapter relate specifically to stored procedures on DB2 for i, many of the concepts closely relate to user-defined functions (UDFs).

This chapter covers the following topics:

- Database error reporting strategy
- Error handling in SQL stored procedures
- Error handling in external stored procedures
- Error handling in Java stored procedures
- Retrieving user-defined errors in a client application
- Transaction management in stored procedures
- External stored procedures and commitment control
- Several practical examples

6.1 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 that is 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 it processes 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 that is returned is 0, and the SQLSTATE is '00000'.

6.1.1 User-defined errors and warnings

User-defined errors are certain conditions in an application that are defined as errors by the business logic rather than by the runtime environment. For example, a business rule might exist 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 (UDF)) is 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 represent the nature of the error or warning, which is also called *error class*, and the last three characters represent the detailed error condition, which is 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. It is commonly accepted to code user-defined SQLSTATES in the form 38*yxx*. The *y* can be any letter or number, and the *xx* is any two digits or uppercase letters, taking care not to use one of the predefined SQLSTATES, such as 38502. (The external function is not allowed to execute SQL statements.) For more information about SQLCODEs and SQLSTATEs, see *DB2 Universal Database for iSeries SQL Messages and Codes*, which is available at this website:

https://publib.boulder.ibm.com/iseries/v5r1/ic2924/info/rzala/rzalamst.pdf

You can also define user-defined warnings in a consistent way by using SQLSTATEs 01H*xx*.

6.1.2 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 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.

Note: SQLSTATE uses reserved ranges for user-defined errors and warnings. For a consistent approach to error handling, you must use one of the following values:

- `00000`: Successful execution.
- `01H`*xx*: Warning. The trailing two characters *xx* can be any digits or uppercase letters. It results in SQLCODE +462 from the SQL runtime.
- 38*yxx*: Error condition. The *y* is an uppercase letter between I and Z, and *xx* is any two digits or uppercase letters. It results in SQLCODE -443 from the SQL runtime.

For external stored procedures, sometimes you might be tempted to use a different SQLSTATE to be returned to the calling client. This approach will not work. You can set the SQLSTATE only to the values that are specified here. Otherwise, the calling program receives sqlstate `39001`, which indicates an invalid SQLSTATE.

6.2 Error handling in SQL stored procedures

This section describes different types of error handling statements for the SQL stored procedure:

- Condition and handler declarations
- SIGNAL and RESIGNAL statement
- SQLCODE and SQLSTATE variables
- GET DIAGNOSTICS EXCEPTION

6.2.1 Condition and handler declaration

The following typical situations require explicit error handling:

- In a sequential FETCH, record not found means that the cursor reached the end of the file.
- In an INSERT, duplicate key values or check constraint inconsistencies exist.
- In a DELETE, a referential integrity violation occurred.
- In an UPDATE, a check constraint violation occurred.
- In a SELECT, a row is not found.

You can handle these situations by declaring condition variables and a handler for each condition. Use the condition declaration to declare a meaningful condition name for a corresponding SQLSTATE value. The following example illustrates how to code condition declarations:

```
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 error occurs when an UPDATE or INSERT violates a check constraint that was 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 that are 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.

Three types of condition handler are available:

- CONTINUE: When this condition is specified, after the SQL statement in the handler successfully executes, the control is returned to the SQL statement that follows the SQL statement that raised the exception.
- EXIT: If EXIT is specified, after the SQL statements in the handler are 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. 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.

For 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 that is contained in the DB2 Communications Area (DB2 CA). The DB2 runtime sets this value each time that an SQL statement is executed. SQLSTATEs are consistent across all DB2 platforms. The general conditions are listed:

- SQLEXCEPTION specifies that the handler is invoked when an SQL exception occurs. SQLEXCEPTION 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. SQLWARNING corresponds to SQLSTATE class '01'.
- NOT FOUND specifies that the handler is invoked when a NOT FOUND condition occurs. NOT FOUND corresponds to SQLSTATE class '02'.

In addition, you can provide handlers for a *specific* condition. For example, you might declare a handler for SQLSTATE '02505', which corresponds to the duplicate key exception. A code snippet illustrates the handler:

```
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:

```
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 to handle special conditions:

```
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. Assume that you have the following two declarations:

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION1
SET v_sqlcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR            2
   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 that are 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 was added in V5R2.

The following example defines all key elements of the compound control statement, condition, and handler:

```
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 that follows the SQL statement that raised the condition.

Consider the following example:

```
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 of the transactions that were performed in the compound statement, and errmsg is set to 'ERROR, ROLLBACK WAS ISSUED'. The control is returned to the end of the compound statement.

We are ready to complete our procedure. See Example 6-1.

Example 6-1 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
```

In certain cases, you might need to execute more than one statement on the DECLARE of the handler, as shown in the following code snippet in Example 6-2.

Example 6-2 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 did not support nested compound statements. However, you can circumvent this limitation by coding a “dummy” loop that performs the same function as the previous example by using the following approach in Example 6-3.

Example 6-3 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;
```

6.2.2 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, with the message text. If a handler is defined to handle the exception, the handler is called immediately by the SIGNAL statement, as shown in Example 6-4.

Example 6-4 Raising an error by using the 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 the G10 procedure is called, you can see the entries log in the result table (Figure 6-1).

SELECT * FROM result - As400ws

	PROC	RES
1	exec of G10	START of Proc
2	exec of G10	EXIT handler fired
3	exec of G10	END of Proc

Figure 6-1 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 6-5.

Example 6-5 Raising error by 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 6-2).

SELECT * FROM result - As400ws

	PROC	RES
1	exec of G11	START of Proc

Figure 6-2 Results from the G11 procedure

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, this 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
```

Any valid SQLSTATE value can be used in the SIGNAL statement. You are not limited to the sqlstates of the '38*yxx*' class. However, for consistency reasons, we recommend that you use the '38*yxx*' 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 be coded only as part of the SQL PSM condition handler. The RESIGNAL statement is used to resignal an error or warning condition. It returns SQLSTATE and SQL message text to the invoker.

The use of the RESIGNAL statement *without* an operand causes the identical condition to be passed outward. A RESIGNAL statement *with* an operand causes the original condition to be replaced with the new condition that you specified.

See Example 6-6.

Example 6-6 Raising errors by using SIGNAL and RESIGNAL statements

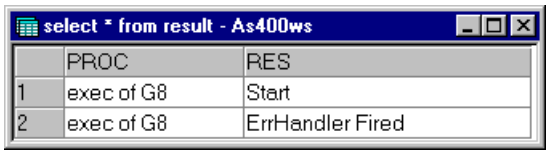
```
CREATE PROCEDURE G8()
LANGUAGE SQL
BEGIN
   DECLARE not_found_text CHAR(70);
   DECLARE CONTINUE HANDLER FOR SQLSTATE '38TNF'
        BEGIN                                                          1
          INSERT INTO result(proc,res) values('exec of G8',
             'ErrHandler Fired');
           RESIGNAL SQLSTATE '38TNF' SET MESSAGE_TEXT=not_found_text; 2
          INSERT INTO result(proc,res) values('exec of G8',
             'Stmt after resignal');                                   3
        END;                                                           1
   SET not_found_text = 'Part number not found!';

   INSERT INTO result(proc,res) values('exec of G8','Start');
   SIGNAL SQLSTATE '38TNF';
   INSERT INTO result(proc,res) values('exec of G8','After signal 38TNF');
   INSERT INTO result(proc,res) values('exec of G8','End');
END;
```

Notes: The following notes refer to Example 6-6 on page 151:

1. In previous versions to V5R2, nested compound statements are not supported and must be replaced by a dummy LOOP.
2. This RESIGNAL statement overrides the system message for SQLSTATE 02000 “Row not found” with “Part number not found!”
3. After the RESIGNAL command is fired, the stored procedure returns the specified signal to the caller application immediately. No statements that follow the RESIGNAL are executed.

After calling the G8 procedure, you see the entries log in the result table (Figure 6-3).



	PROC	RES
1	exec of G8	Start
2	exec of G8	ErrHandler Fired

Figure 6-3 Result from the G8 procedure

The following embedded SQL RPGLE program illustrates how to retrieve the user-defined SQLSTATE and the error message text that are returned from the G8 SQL procedure:

```
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 Java Database Connectivity (JDBC), the toolbox JDBC, and the Open Database Connectivity (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.

We look at several practical examples. We start with an SQL stored procedure that is implemented in SQL Persistent Stored Modules (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 departmental manager's 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 its manager. The routine's logic checks if the rule is not compromised. 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 6-7. The numbered sections are explained further in the following list.

Example 6-7 Stored procedure that uses SIGNAL and RESIGNAL

```
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,
   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 handlers for user-defined error sql states
declare exit handler for sqlstate '38SO1' 2
    resignal sqlstate '38SO1'
          set message_text ='MODSAL: Compensation exceeds the limit.';

declare exit handler for sqlstate '02000' 3
signal sqlstate '38SO2'
           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'; 1
end if;

update employee set salary = v_salary + i_salary where empno = i_empno;

end
```

Code sample notes

The following notes refer to Example 6-7 on page 152:

1 If the business rule is compromised, sqlstate '38S01' is signaled. The control is transferred to the error handler that is defined for this state. The SIGNAL might include the message text and it might be signaled directly to the invoker.

2 This error handler that is 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.

3 The sqlstate '02000' is returned to the SQL SP if no data for the employee number is passed as the first parameter. This condition can be thrown either by the FETCH or searched UPDATE statement. The error handler handles this condition by signaling sqlstate '38S02' to the caller.

6.2.3 SQLCODE and SQLSTATE variables in the SQL procedure

It might be 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. Because the local SQLCODE and SQLSTATE variables are reset after *each* statement, their values must be copied to other local variables. The following example shows the use of condition handlers to assign the values of the SQLSTATE and SQLCODE variables to 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;
```

6.2.4 Returning values by using the RETURN statement

A return value can be used to return a program code to a caller application. For example, you can use the return value to inform the caller application whether the SQL procedure completed successfully. In Example 6-8, if no records were updated, the procedure returns -1. Otherwise, it returns 0 to represent success.

Example 6-8 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 155 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. For more information, see this website:

https://ibm.biz/Bd4c5V

6.2.5 GET DIAGNOSTICS

The GET DIAGNOSTIC statement can be used in several 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 6-9 writes SQLSTATE and the error message text to the errorlog table.

Example 6-9 Usage of GET DIAGNOSTICS in SQL PSL

```
CREATE PROCEDURE GetDiag()
LANGUAGE SQL
BEGIN
   DECLARE msgtxt CHAR(70);
   DECLARE msgtxtlen INTEGER;
   DECLARE PrevSQLState CHAR(5);
   DECLARE SQLSTATE CHAR(5);                                          1
   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);            2
```

```
     END;
    INSERT INTO result(proc,res) values('exec of GetDiag','Start');
    INSERT INTO unknown values('TEST');                              3
END;
```

Notes: The following notes refer to Example 6-9 on page 154:

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). Therefore, GET DIAGNOSTICS EXCEPTION must 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, available since V4R4, of the GET DIAGNOSTICS statement. Use it to retrieve the number of rows that are affected by an INSERT, UPDATE, or DELETE statement. The numrec procedure that is shown in Example 6-10 is used to increase the credit of a customer in the city of Rochester by 5%. GET DIAGNOSTIC is used to assign the number of records that are updated to a variable *num_records*, which might, in turn, be returned to the caller application.

Example 6-10 Usage of ROW_COUNT in SQL PSL

```
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 differs slightly from other database implementations because 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 that is shown in Example 6-11 attempts to increase the credit limit for all customers in Rochester.

Example 6-11 Usage of RETURN_STATUS in SQL PSL

```
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');                                 1
```

```
      GET DIAGNOSTIC retval = RETURN_STATUS;                          2
      IF retval <> 0 THEN
        GOTO BadNews;                                                 3
      END IF;
   END IF;
   ...
   BadNews:
   RETURN -1;
END
```

Notes: The following notes refer to Example 6-11 on page 155:

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.

6.2.6 Error handling in nested compound statements

When nested compound statements are used, each compound statement has its own scope for variable definitions and for its condition definitions and error handlers. Example 6-12 illustrates the scope of different error handlers.

Example 6-12 Error handlers in nested compound statements

```
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';
      ELSE
         SET I = 1; /*Don't do anything */
      END CASE;
END;
```

The expected behavior of error handlers in nested compound statements is described in Table 6-1.

Table 6-1 Expected behavior of error handlers in nested compound statements

PARAM value	Expected behavior
1	Error 38H01 is fired from the internal compound statement. That error is not handled by any error handler and it will be passed back to the caller program.
2	Error 38H02 is fired from the internal compound statement. That error is not managed by any error handler in the internal compound statement, but it is handled by an error handler in the external error handler, which will fire error 38HE0 that will be passed back to the caller.
3	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.
4	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.
5	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.
6	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.
7	The stored procedure will terminate without errors.

In the following snippet, when statement (stmt) 2 causes a NOT FOUND condition, the defined handler captures the exception. After the defined handler finishes its operation, it exits the nested compound exception, not the whole program, and continues 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 the behavior of condition handlers in nested compound statements further, we examine one more stored procedure that is named p_nested_test(). This procedure manipulates data in a sample table called COFFEES. First, the routine calculates the average price of coffee brands that are contained in the table. Then, it inserts a new row.

Example 6-13 shows the source code listing of the procedure. The numbered lines are explained after the example.

Example 6-13 p_nested_test() procedure

```
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  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_COFFEES ;
    IF V_AVG_PRICE IS NULL THEN                                    --[1.1]
      SIGNAL COFFEES_UNKNOWN_AVG_PRICE
             SET MESSAGE_TEXT = 'Unknown avg price of coffee.' ;
    END IF ;
    INSERT INTO JM_DEBUG ( SQLTEXT )                               --[1.3]
    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 ;                        --[3.1]
      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 is defined in the Level_1 compound statement:

  The scope of this handler is the entire stored procedure body.
- An exit and a continue handler are defined in the Level_2_1 compound statement.

  The scope of these handlers is limited to the compound statement in which they were declared.
- An exit handler is defined in the 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 that are 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 method is a common debugging technique that is used by SQL Procedural Language (SQL PL) developers.

Consider the following test scenarios for the stored procedure execution to see how various programming constructs interact.

Test scenario 1

In this scenario, the COFFEE table initially contains no rows. We call the procedure with the following parameters:

```
CALL P_NESTED_TEST('COFFEES', ' ');
```

The first parameter is an input value that contains the name of the table to manipulate. 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 that are shown in Figure 6-4.

SQLTEXT
Level_1-Main Procedure Body: V_CUR_STMT prepared
Level_2_1-Handler for SQLSTATE 70019: Message: Unknown avg price of coffee.
Level_2_1 - Body: v_avg_price = 0E0
Level_1-Resuming processing after end of Level_2_1 compound statement.
Level_2-2-Main Body: 1 row(s) inserted in COFFEES.
Level_1-Resuming processing after end of Level_2_2 compound statement.

Figure 6-4 Message file from Test scenario 1

The execution proceeds successfully until the IF statement at [**1.1**] in Example 6-13 on page 158 is reached in compound statement Level_2_1. Because the COFFEES table is empty now, the V_AVG_PRICE variable is set to NULL (unknown). It causes the COFFEES_UNKNOWN_AVG_PRICE signal to fire. 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. 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 that are generated by DB2 for i or OS/400. In our case, the business rule is that the unknown value of the average coffee price is not allowed.

Test scenario 2

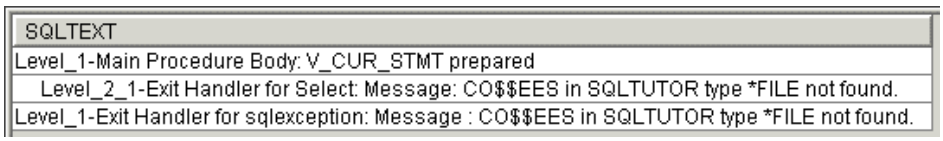
This time, we call the stored procedure with an intentionally corrupted first parameter:

```
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 that are shown in Figure 6-5.



SQLTEXT
Level_1-Main Procedure Body: V_CUR_STMT prepared
Level_2_1-Exit Handler for Select: Message: CO$$EES in SQLTUTOR type *FILE not found.
Level_1-Exit Handler for sqlexception: Message : CO$$EES in SQLTUTOR type *FILE not found.

Figure 6-5 Message file from Test scenario 2

After invocation, the execution proceeds until the OPEN cursor statement is reached at [**2.1**] in Example 6-13 on page 158. Because the table name is corrupted, the DB2 runtime fails to find the corresponding table object, and it throws an SQL exception with SQLSTATE '42704', which indicates that the table was not found. Because 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 that is mapped to custom SQLSTATE '70010'. The control is transferred to the end of the compound statement Level_2_1 and returns to the main procedure body. The error condition that is signaled in Level_2_1 is pending, so now the runtime tries to locate an appropriate handler in the Level_1 compound statement (main procedure body). No handler exists for a specific condition COFFEES_QUERY_FAILED, but the runtime finds the general exception handler at [**2.3**]. After the handler writes a debug message into jm_debug, the handler resignals the error condition that is returned to the caller and concludes in the error messages that were shown.

Test scenario 3

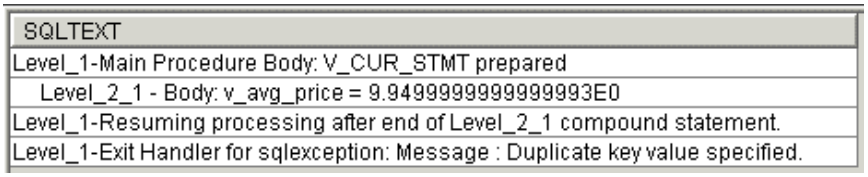
We call the stored procedure again with the following parameter:

```
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 that are shown in Figure 6-6 on page 162.



SQLTEXT
Level_1-Main Procedure Body: V_CUR_STMT prepared
Level_2_1 - Body: v_avg_price = 9.9499999999999993E0
Level_1-Resuming processing after end of Level_2_1 compound statement.
Level_1-Exit Handler for sqlexception: Message : Duplicate key value specified.

Figure 6-6 Message file from Test scenario 3

After the invocation, the execution proceeds until the EXECUTE IMMEDIATE statement is reached at [**3.1**] in Example 6-13 on page 158. The first column of the COFFEES table is defined as the primary key. The row with the key value 10 in the first column exists because it was inserted when Test scenario 1 was run. Therefore, the 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 pending error condition. Now, the runtime locates the general handler for SQL exceptions and invokes it at [**3.2**]. After the handler writes a debug message, the handler resignals the original error that was thrown in the inner Level_2_2 compound statement.

Note: The following required program temporary fixes (PTFs) deliver support for condition handlers in the complex nested compound statements that we describe in this section:

- V5R2: SI17232 and SI17233
- V5R3: SI18929

Using a nested error handler to avoid locks

We use a nested error handler to avoid locks that are caused by cursors that are left open. The open cursor operation results in a read lock that is 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. Under certain circumstances, the described behavior can result in locks that persist while a specific job (activation group) is active.

Under the described circumstances, locks can persist while a certain job (activation group) is active. We illustrate this situation with an example.

We run the procedure that is shown in Example 6-14 from the Run SQL Scripts utility of System i Navigator.

Example 6-14 Example with cursor left open

```
create procedure testopencurs()
language sql
p1: begin
declare v_cof_name varchar(32);
declare c1 cursor for
   SELECT cof_name FROM COFFEES;
declare exit handler for sqlexception
   resignal ;                                    4

OPEN c1;                                         1
FETCH c1 into v_cof_name;
COMMIT HOLD;                                     2
SIGNAL SQLSTATE '70000'
```

```
set message_text='Exit before cursor closed';   3
CLOSE c1;
END p1;
```

The following notes refer to Example 6-14 on page 162:

- 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.

You cannot use the Run SQL Script utility 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 6-15 illustrates how to explicitly close the cursor in the stored procedure with the nested error handler.

Example 6-15 Avoid leaving the cursor open

```
create procedure testopencurs()
language sql
p1: begin
declare v_cof_name varchar(32);
declare c1 cursor for
   SELECT cof_name FROM COFFEES;
declare exit handler for sqlexception               1
begin
   declare i int;
   declare continue handler for sqlstate '24501'    3
       set i =0;
   close c1;                                        2
   resignal;
end;
OPEN c1;
FETCH c1 into v_cof_name;
COMMIT HOLD;
CLOSE c1;
SIGNAL SQLSTATE '70000'
set message_text='Exit after cursor closed';
END p1;
```

The following notes refer to Example 6-15:

- 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 the SQL state of '24501', which is intercepted by the inner continue handler at 3.
- After the exception is ignored, 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.

6.2.7 Use nested compound statements for better performance

You need to use nested compound statements to localize exception handling and cursors. If certain handlers are specified, code is generated to check to see whether 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 situations 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:

```
...
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.

6.3 Error handling in external stored procedures

External stored procedures must also manage error conditions. Ideally, all errors are trapped and corrective action is taken when it is required in the stored procedure. However, situations occur when it is not possible to handle the error condition within the stored procedure. In these cases, the stored procedure *propagates* the error condition to the calling program.

Important: If the external stored procedure completes with an error, the states of the output parameters are undefined. No guarantee exists 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 for i passes to the routine several parameters in addition to those parameters that are specified on the parameter list, as explained in 4.2.1, "SQL parameter style" on page 46. The DB2SQL parameter style is an extension to the SQL parameter style because the SQL extra parameters are passed to the external procedure in addition to the DBINFO data structure, as explained in 4.2.2, "DB2SQL parameter style" on page 47.

The SQL parameter style has the following structure:

```
IN | OUT | INOUT argument [repeated],
INOUT Argument indicator [repeated - one for each argument],
OUT Sqlstate,
IN Procedure name,
IN Specific name,
OUT Diagnostic message
```

Two parameters are especially interesting 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, see 6.1, “Database error reporting strategy” on page 144.
- Diagnostic message: This output parameter can be set to a customized error message.

Important: Sometimes, you might be tempted to set sqlstate to the value that is returned to the stored procedure by the SQL runtime to pass it on to the calling client. This approach will not work. You can set sqlstate only to the values that are previously specified. Otherwise, the calling program receives sqlstate 39001, which indicates an invalid sqlstate.

6.3.1 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 correct SQLCODE, SQLSTATE, error message length, and the error message text.

You can use the SQL parameter style for more flexibility. SQLSTATE can be set within the external stored procedure with the message text.

SQL and DB2SQL parameter style

This section shows how error handling is easier with the SQL parameter style. All errors that occur within the stored procedure can be handled within the stored procedure, depending on the business logic that is implemented by the stored procedure. When a stored procedure is called by a program, the calling program checks the completion status. As described in 4.3.1, “Coding for SQL parameter style” on page 48, a stored procedure can return an output parameter that contains SQLSTATE, 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 binary large object (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 integrated file system (IFS) file name that contains the picture of the product. Possible errors that can occur during the stored procedure execution are listed:

- 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 '38*yxx*' if a severe error is encountered, but it cannot be corrected. The severe error conditions, 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 6-2.

Table 6-2 Errors

Error condition	SQLSTATE	Message text
IFS file not found	38TNT	IFS file not found
STOCKPIC table not found	01HTC	N/A
Any other errors	38999	Unhandled error condition within the stored procedure

Examine the CREATE PROCEDURE statement for the INSPIC procedure in Example 6-16. The numbered sections are explained in the following list.

Example 6-16 INSERTPIC external stored procedure creation

```
CREATE PROCEDURE                SPROCLIB/INSPIC(              1
                                      IN prdnbr CHAR(5),      1
                                      IN filename CHAR(50) )  1
SPECIFIC                        INSPIC
LANGUAGE                        C
EXTERNAL         NAME           SPROCLIB/insertpic
MODIFIES                        SQL DATA
PARAMETER STYLE                 SQL                           2
```

CREATE PROCEDURE notes

The following notes refer to the numbers in Example 6-16:

1 This section is the external stored procedure that is called INSPIC with two input parameters.

2 We use the SQL parameter style.

Examine the source of INSERTPIC, which is referred to in the CREATE PROCEDURE statement that is shown in Example 6-16. 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 6-17 illustrates how error handling can be effectively implemented with the SQL parameter style.

Example 6-17 INSERTPIC external stored procedure C source

```
#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[255];
      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))                                              5
  {
    strncpy(statevar,"00000",5);
    strncpy(argv[5],statevar,5);                                        1
    exit(0);
   }
ErrorHandler:
  if ((sqlca.sqlcode ==-204))                                           3
  {
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)                 4
     VALUES(:prdnum, :pict_file);
   strncpy(statevar,"01HTC",5);
   strncpy(argv[5],statevar,5);
   exit(0);
  }
  if ((sqlca.sqlcode ==-452))
  {
    strncpy(statevar,"38FNT",5);                                       2
    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);                                        1
    strncpy(errmc,"unhandled exception",21);
    strncpy(outmsgtxt_var.data,errmc,21);                               1
    outmsgtxt_var.length = 21;
    *(struct outmsgtxt *) argv[8] = outmsgtxt_var;
   }
    exit(1);
}
```

Code sample notes

The following notes refer to the numbers in Example 6-17 on page 166:

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 fails. The program logic checks for SQLCODE= -452, which corresponds to SQLSTATE='428IA'. This SQLSTATE cannot be directly returned to the calling program. If you return SQLSTATE='428IA', it is treated as an invalid SQLSTATE, and the calling program SQLCA.SQLSTATE contains '39001'. We set the SQLSTATE output parameter to a user-defined value. Because the severity of the error is high, we set the SQLSTATE to '38FNT'. The message text explains the error condition.
3. If the PRODPIC table does not exist in the library, the INSERT statement fails. The program logic checks for SQLCODE= -204, which corresponds to SQLSTATE='42704'. This error condition is handled within the stored procedure, and error correction occurs. If the insertion fails because the table did not exist, the PRODPIC table is created within the stored procedure. Any errors that occur on the execution of the CREATE TABLE statement are returned to the calling program with an SQLSTATE of '38999' and the related message text.
4. After the table is created, the values are inserted into the table.
5. On the successful execution, the stored procedure returns the SQLSTATE that is set to “00000” to the calling program.

The calling program checks for the different error conditions after the SQL CALL statement by using SQLWHENEVER or a conditional structure, such as IF-THEN_ELSE or CASE statements. The code in Example 6-18 shows the errors that were handled in the calling program.

Example 6-18 Client C program that calls a stored procedure and recovers error information

```
#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);
}
```

6.3.2 GENERAL and GENERAL WITH NULLS parameter styles

External stored procedures that are defined with the GENERAL and the GENERAL WITH NULLS parameter styles have no specific way to return error conditions. The GENERAL and GENERAL WITH NULLS parameter styles were provided as a mechanism of reusing an existing procedure, and they were not necessarily designed as database stored procedures. Most of the procedures report their error conditions through an output parameter, such as the example in 4.3.3, “Coding the GENERAL WITH NULLS parameter style” on page 57.

6.4 Error handling in Java stored procedures

The Java language allows great 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 that is used for any other errors in Java by using the 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 that is passed back to the caller depends on the type of exception that is thrown in the Java method, and 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, the DB2 for i runtime handles only certain return codes. For consistency reasons, we recommend that you adopt a similar approach to the approach that is presented in 6.2, “Error handling in SQL stored procedures” on page 145. While you throw the `SQLException` for a user-defined error condition, you can set the sqlcode to -438 and the sqlstate to a state of class '38*yxx*'. This way, the error condition is correctly recognized by the database runtime and passed back to the calling process.

You also can use the `SQLWarning` class, which is an extension of the `SQLException` class, to manage warnings.

Table 6-3 is a guideline for setting the SQL return codes to values that can be handled correctly by client code that is written in any programming language. It summarizes the mapping possibilities mapping between a Java exception that is returned to the caller and its corresponding SQLException that is returned to the caller.

Table 6-3 Mapping between Java exceptions and SQLExceptions

Java exception	SQLCODE of the Java SQLException	Result in a returned SQLException	SQLCODE of the returned SQLException	SQLSTATE of the returned SQLException
java.sql.SQLException	> 0 or = 0	No	No SQLException	No SQLException
java.sql.SQLException	< 0	Yes	SQLCODE of the Java SQLException	SQLSTATE of the Java SQLException
Any other exception	N/A	Yes	-443	'38501'

When the SQLCODE of the Java SQLException is 0, the client code assumes that the Java stored procedure executed without any errors or warnings. The only trace that a Java SQLException was thrown is message MCH74A0, which is 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 that corresponds 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. Therefore, you can create your own Java stored procedure with, for example, an SQL code of +438 (stored procedure that signaled an error). And, you can create an SQLSTATE that is defined at your convenience, for example, '`01ABC`'. Then, you can use the `getWarnings()` method in the Java client to retrieve your user-defined SQLSTATE and resolve the issue.

Important: The message of the SQLEXCEPTION that is returned to the caller is always the message that is associated with the IBM i server SQLCODE. It does not refer to the original message of the Java exception.

If the SQLCODE of the SQLException is negative and the SQLCODE is a valid DB2 Universal Database for iSeries value, the related 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

If you work 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 to handle errors.

Assume 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.

Note: We refer to the open source version of the JDBC driver that is available for download:

https://ibm.biz/Bd4c5V

For the ODBC functionality, you need to ensure that you loaded the latest version of the iSeries (IBM i) Access Express Service Pack on your workstation. We also recommend that you install the latest database fix pack on your IBM i server.

The equivalent Java SQLException can be defined as shown:

```
if (e instanceof FileNotFoundException) throw
     new SQLException("IFS file not found.", "38TNT", -438);
```

For the Java stored procedure, the user-defined error message is overlaid with the system error message that is associated with the -438 error code. If no system message for this error code is on your IBM i server, you can define the message with the following control language (CL) command:

```
ADDMSGD MSGID(SQL0438) MSGF(QSQLMSG) MSG('Stored procedure signaled an error')
```

Error handling example

In 4.2.1, "SQL parameter style" on page 46, we explain that Java stored procedures on the IBM i server can pass large object (LOB) parameters only when you use the DB2GENERAL parameter style. However, the Java stored procedure can use character large object (CLOB) or binary large object (BLOB) classes internally if the JDBC 2.0 driver is used. It is also possible to handle BLOB without using the BLOB class, but by using an InputStream, as shown in the following example.

This illustration features a Java stored procedure, LoadPicture, which is 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 that indicates the special condition. For the C-embedded SQL version of this stored procedure, see 6.3, "Error handling in external stored procedures" on page 164. The code of this stored procedure is presented in Example 6-19.

Example 6-19 Java stored procedure that reports error and warning conditions

```
import java.sql.*;
import java.io.*;                                                    1
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(?, ?)");      2
    }
```

```
    catch(SQLException ex) {                                              3
      if (ex.getErrorCode() == -204)
      // or (e.getSQLState().equals("42704"))                             4
      {  // the table doesn't exist, we create it
        s = con.createStatement();
        s.executeUpdate("CREATE TABLE PRODPIC " +                         5
                        "(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;                                                         6
      }
    }
    f = new File(imageFile.trim());                                       7
    int filelength = (int) f.length();
    try {
      is = new FileInputStream(f);                                        7
    }
    catch(IOException e) {
      if (e instanceof FileNotFoundException)
        throw new SQLException("IFS file not found.", "38FNT", -438);     8
    }
    ps.setString(1, imageId);
    ps.setBinaryStream(2, is, filelength);                                9
    ps.executeUpdate();
    if (tableCreated)
      throw new SQLException("Table created and insert successful.",
                             "01HTC", 438);                               10
  }
}
```

Notes: The following notes refer to Example 6-19 on page 171:

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 might differ 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 SQLCODE -204 on the IBM i server.
5. Table was created that contains the pictures, if it was not found.
6. If a non-handled SQLException is thrown, it is thrown to the caller.
7. A FileInputStream, which is based on the file path that is received as a parameter, is created in the IFS.
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 SQLEXCEPTION with SQLCODE -443 that is returned to the caller. This exception is a generic exception that does not provide the caller process with the 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, which indicates that a serious error condition occurred in the stored procedure. Also, we create our own SQLSTATE '`38`*`FNT`*', where *FNT* is a user-defined portion of the SQLSTATE. Now, the caller application can test for the SQLSTATE that was 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 was created, the stored procedure signals a warning (SQLException with a positive SQLCODE). We choose the SQLCODE +438, which indicates that the stored procedure signaled an error. We create the SQLSTATE '`01H`*`CT`*', where *CT* stands for create table.

The SQL statement to create the stored procedure is shown:

```
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 6.5.1, “Retrieving error conditions in a JDBC client” on page 174.

6.5 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. If the sqlcode field contains either -443 or -438, the SQLSTATE and the diagnostic message are retrieved from the SQLCA. Then, the client application can access these values by using standard error handling code.

Note: From the client application point of view, no difference exists between the native DB2 for i runtime errors and the user-defined error conditions. The user-defined errors are displayed as another SQL runtime error. This capability is useful in client/server environments, where the retrieval of native IBM i errors can be cumbersome or impractical.

6.5.1 Retrieving error conditions in a JDBC client

A Java client, whether it is a JDBC client or a Structured Query Language for Java (SQLJ) client, uses the TRY and CATCH blocks to manage errors, including database error conditions. Two predefined classes are available to manage 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 useful methods for retrieving related information, such as `getErrorCode()` and `getSQLState()`. The `getErrorCode()` method retrieves the vendor-specific error code, which in the DB2 Universal Database family is the SQLCODE. The `getSQLState()` method retrieves the SQL92 standardized SQLSTATE.

`SQLException` is thrown by stored procedures when an error condition is reached. `SQLWarning` does not throw an exception. As shown in the example, `SQLWarning` can be retrieved with the `getWarnings()` method that is associated with the SQL statement. The following example shows how to manage both error and warning conditions.

Example 6-20 shows a JDBC client that retrieves error conditions.

Example 6-20 Java client program that recovers error or warning information

```
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");                    1
      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();                                                          2
        SQLWarning w = cs.getWarnings();                                       3
        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());             4
          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)                                                   5
      {
        System.out.println("Here's the e.toString() : " + e.toString());     6
        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);
    }
  }
}
```

Notes: The following notes refer to Example 6-20 on page 174:

1. The application reads the information that it needs to connect to the DB2 platform and to call the stored procedure from a properties file that is called `"piclogon.properties"`. The contents of this file are shown:

   ```
   # 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 that are passed to the stored procedure.

2. The stored procedure is executed after its two parameters are established.
3. The first warning or SQLException with a positive SQLCODE that can be returned by the stored procedure is retrieved.
4. All of the information that is contained in the warning is retrieved and displayed.
5. Any SQLException (with a negative SQLCODE) that occurred during the execution of the stored procedure is retrieved.
6. The information that is contained in this exception is retrieved and displayed.

6.5.2 Retrieving error conditions from an ODBC or CLI client

Whenever any ODBC or DB2 command-line interface (CLI) function is called, a return code is returned to the application to indicate the success or failure of the attempted operation. 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:

- Native error code
- SQLSTATE
- Error message text

Based on the values that are returned, the application determines the course of action. Table 6-4 provides the list of possible codes that can be returned by the function.

Table 6-4 DB2 CLI function return codes

Return codes	Explanation
SQL_SUCCESS	The function completed successfully. No additional SQLSTATE information is available.
SQL_SUCCESS_WITH_INFO	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".
SQL_NO_DATA_FOUND	This return code is mostly associated with query results. The function completed successfully but it cannot find any relevant data.
SQL_ERROR	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, native error code, and error message text.
SQL_INVALID_HANDLE	The function failed because an invalid environment handle, connection handle, or statement handle was sent as an argument to a function.

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, such as HY*xxx*, correspond to errors that occurred in the DB2 ODBC or CLI driver. The SQLCODE for an error that is generated by DB2 ODBC or CLI is -9999.

Retrieving error codes and messages

Whenever an error condition occurs, the application must 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 handle 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 handle 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 handle 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 size is ideally 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 6-21.

Example 6-21 CLI client program that recovers an error condition

```
SQLHENV         Hnd_Henv;
SQLHDBC         Hnd_Hdbc;
SQLHSTMT        Hnd_Hstmt
SQLRETURN       Nmi_ReturnCode;

int PrintError( SQLHENV Hnd_Henv, SQLHDBC Hnd_Hdbc,
                SQLHSTMT Hnd_Hstmt );

void main() {
     Nmi_ReturnCode = SQLAllocEnv( &Hnd_Henv );
     if ( Nmi_ReturnCode != SQL_SUCCESS ) {
        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 );
```

```
}

/* --- Dsiplay 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 that retrieves error conditions that occurred in the stored procedure, see 6.8.4, “C++ client code that uses ODBC” on page 196.

6.6 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 IBM i server) and rows 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 IBM i clients underutilize. With the original transaction model of the IBM i, programmers used a mechanism that allowed 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, must stay independent from caller or firing processes. For this reason, we, as designers and programmers, must carefully plan for an error-handling strategy that does not ruin the caller or firing process work.

In this section, we describe the transaction management strategies that are available in stored procedures. We start with transaction management terminology.

6.6.1 Transaction management terminology

A *transaction* is a set of operations to complete 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 action 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 so that you can 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 must appear as a single atomic change to the user. Users and application programmers might think of an LUW as a transaction. Commitment control ensures that either the entire group of individual changes occur on all systems that participate in the LUW or that none of the changes occurs.

The *savepoint* is a marker or milestone within a transaction to which data and schema changes can be undone. The *nested savepoint* is a model where a savepoint can be defined within an existing savepoint versus a linear model where savepoints are not nested. The *savepoint level* is the atomic context for which a rollback or release outside of the level is not allowed by the user application. Savepoints were first introduced in DB2 Universal Database for iSeries in V5R2.

The *isolation level* is the level of reinforcement of the transactional behavior of the database for a particular activation group. *NONE is 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). Be careful when you read IBM i literature because several of these terms have different meanings for authors that are familiar with other platforms. For more information, see *SQL Reference*, SC41-5612.

6.6.2 Transactional behavior

While the isolation level is set up in a value different than *NONE, we can see the scenarios that are shown in Figure 6-7.

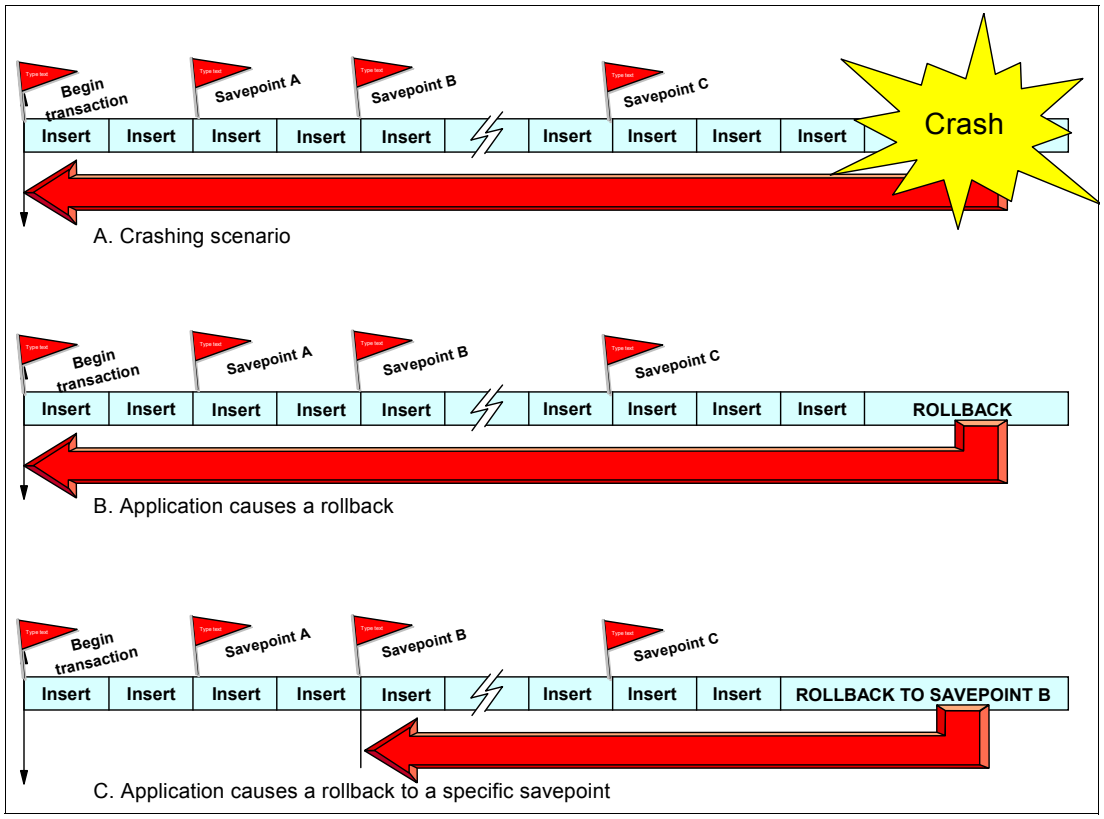


Figure 6-7 Transactional scenarios

In Figure 6-7, you see the following scenarios:

- The procedure reaches a COMMIT statement. All SAVEPOINTs are released and the changes in the database become permanent. It is similar to a confirmation that the completion of the transaction is reached. A new transaction or unit of work is initiated.
- The system crashes before it reaches 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 6-7.

- 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 6-7 on page 180.
- 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 6-7 on page 180. In the figure, savepoint B was not released but savepoint C was released.

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.

Note: The activation group of the Integrated Language Environment (ILE) C program that is generated by the execution of the SQL CREATE PROCEDURE statement is always set to *CALLER. Therefore, an SQL procedure always runs in the same activation group as the program that calls it. If the SQL procedure COMMITs any changes, all changes within this activation group are committed.

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 6-5.

Table 6-5 Events that initiate and terminate savepoint levels

Savepoint level is initiated when	Savepoint level terminates when
A new unit of work is started.	A COMMIT or ROLLBACK is issued.
A trigger is invoked.	The trigger completes.
A user-defined function is invoked.	The user-defined function completes.
A stored procedure is invoked, and the stored procedure was created with the NEW SAVEPOINT LEVEL clause.	The stored procedure returns to the caller.
A BEGIN is part of an ATOMIC compound SQL statement.	An END is part of an ATOMIC compound SQL statement.

When a savepoint level ends, all active savepoints that are 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 that are issued within the parent savepoint level.

In scenario D in Figure 6-8, only statements in stored procedure A are rolled back. All statements in the 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 that are performed in procedures B and C.

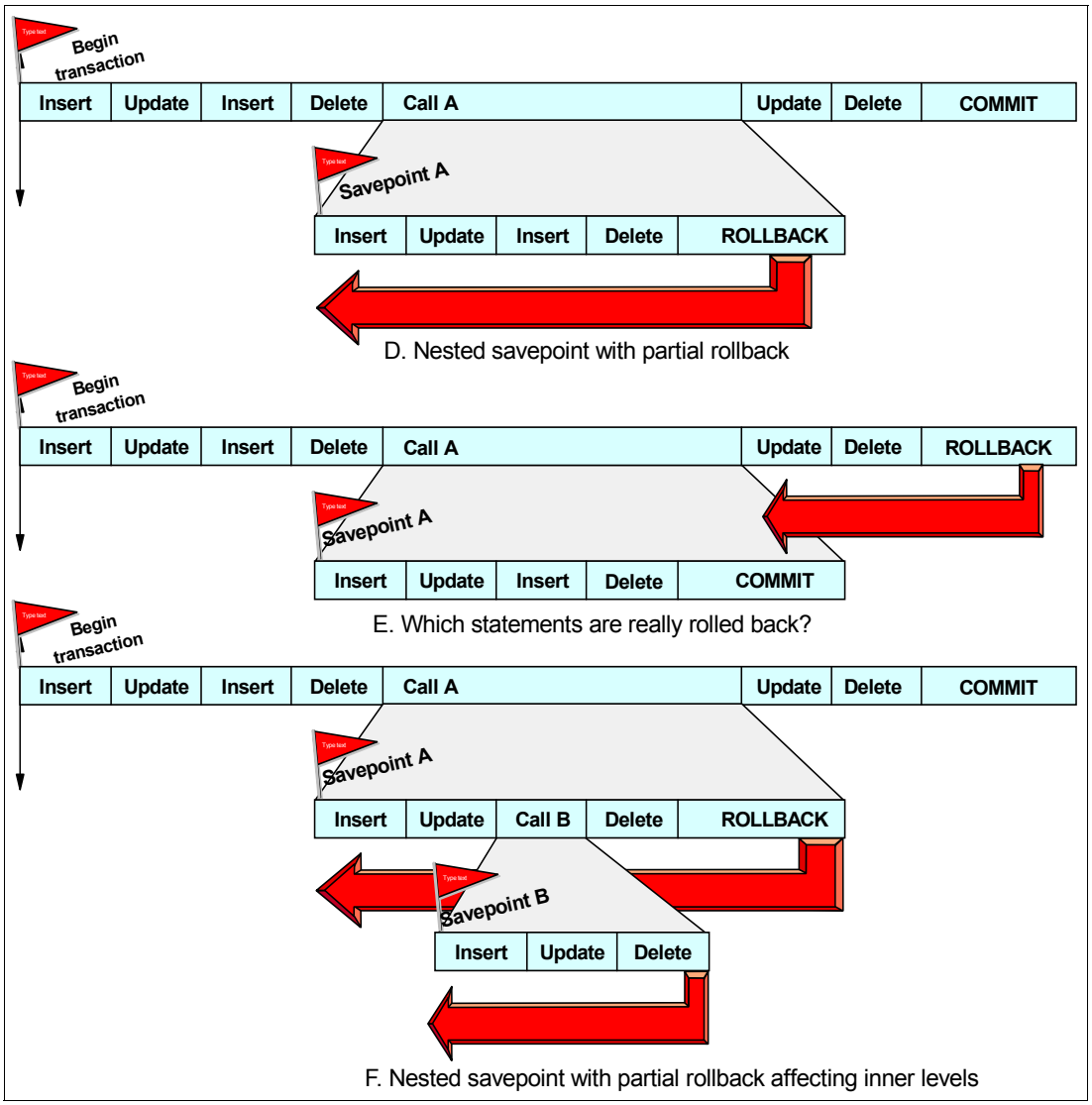


Figure 6-8 Nested savepoints

6.6.3 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.

SAVEPOINT

The SAVEPOINT statement is used to establish a milestone into the current LUW. A name for the savepoint must be supplied. You can optionally specify whether that savepoint name must be unique. If yes, the savepoint cannot be reused with the stored procedure and procedures, triggers, and UDFs that it is 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 needing 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 can be specified as in the following example:

```
...
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. We start by defining what the isolation level is. The isolation level that is 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 it 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 **CRTSQL*xxx***, **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 its value 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 by using an isolation clause. The isolation level is in effect only for the execution of the statement that contains the isolation clause. An example of the SET TRANSACTION statement is shown:

```
SET TRANSACTION ISOLATION LEVEL UR
```

This statement sets the isolation level to READ UNCOMMITTED, which is the equivalent to *CHG. For more information, see *SQL Reference*, SC41-5612.

6.6.4 Transaction management in compound statements

You can use compound statements 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**. **ATOMIC** 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 must *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, ATOMIC *must* be specified in the BEGIN clause.

Example 6-22 illustrates the use of commitment control statements within a compound SQL.

Example 6-22 Commitment control statements within a compound 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;
   SET o_numrec = numrec;
   CLOSE c1;
   COMMIT;  3
END
```

Notes: The following notes refer to Example 6-22 on page 184:

1 The BEGIN clause does not have the **ATOMIC** keyword.

2 ROLLBACK is issued if an SQL EXCEPTION exists. In this case, all of the updates to the CUSTOMER file are reversed.

3 After the cursor is closed, the procedure COMMITs all of the changes to the file.

Example 6-23 presents equivalent code, but the code uses BEGIN ATOMIC instead (1).The error handler that causes a ROLLBACK and the commit statement are not present in this example.

Example 6-23 Commitment control statements that use BEGIN ATOMIC

```
CREATE PROCEDURE CREDITP_ATOMIC
     (IN    i_perinc DECIMAL(3,2),
      INOUT o_numrec DECIMAL(5,0))
      LANGUAGE SQL
 BEGIN ATOMIC    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;
   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 changed slightly. In V5R1, a BEGIN ATOMIC compound statement caused an implicit commit when the procedure reached the END statement. Therefore, a stored procedure that contains an atomic compound statement will automatically commit the database changes that the caller program performed before the caller program called 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 that were performed by the caller before the caller called the stored procedure.

With the new savepoint support, the atomic compound statement 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 approach leaves the commitment responsibility to the caller, and it still allows 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 that are 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:

```
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 you execute the stored procedure CALLER in V5R2, the TEST table is empty, if it was empty before execution. In V5R1, the TEST table is not empty.

6.7 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.

6.7.1 Activation group

The scope for the transaction or UOW is the activation group. Therefore, if both the caller and stored procedure programs are running on the same activation group, changes that are performed by both programs belong to the same transaction. A commit that is performed in the stored procedure will also commit changes that were previously performed by the caller program, which usually we want to avoid.

This situation is illustrated in Figure 6-9.

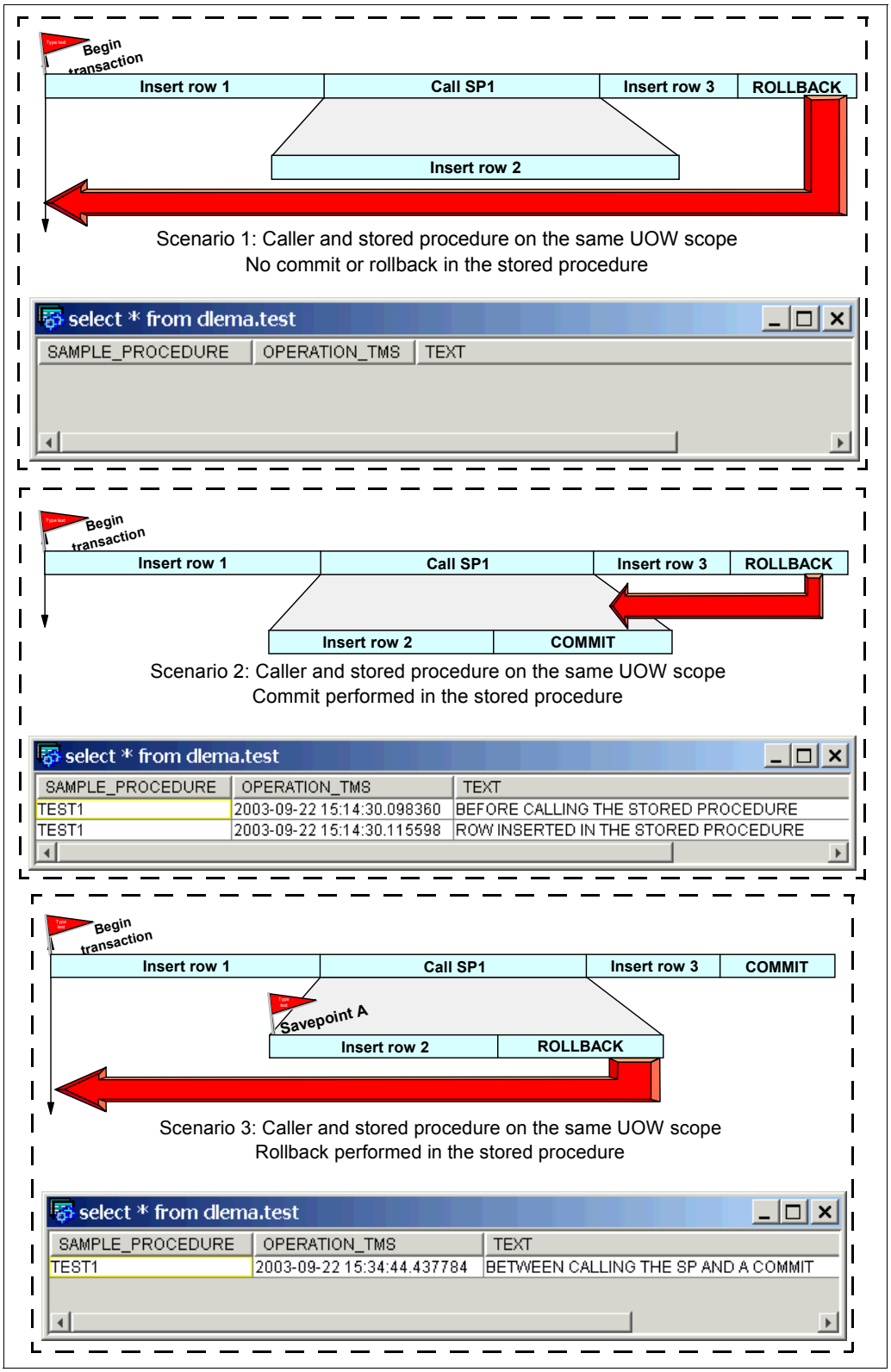


Figure 6-9 Caller and stored procedure in the same activation group

Although scenario 1 is acceptable, scenarios 2 and 3 are 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 that are performed by the other, as shown in Figure 6-10.

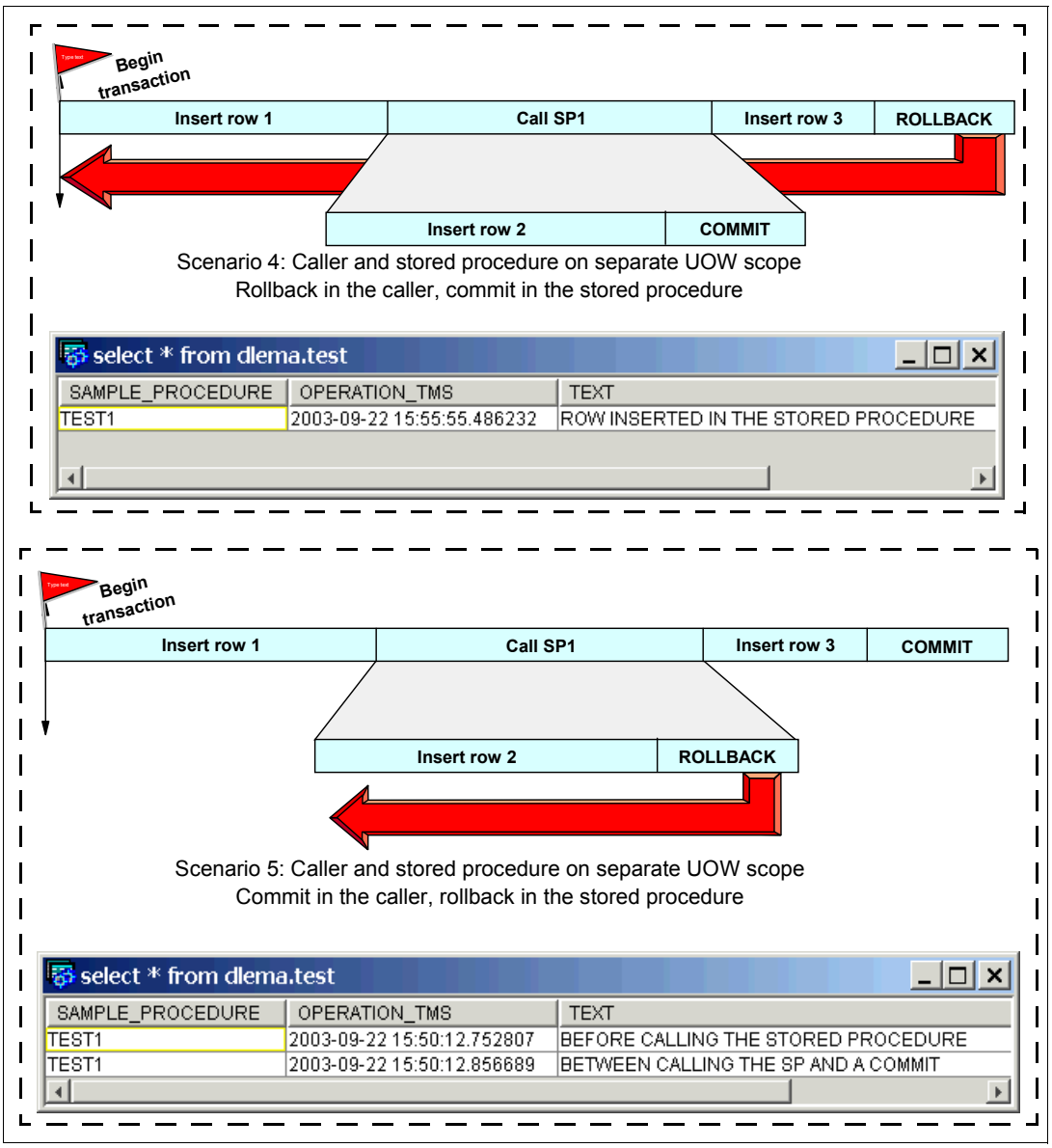


Figure 6-10 External stored procedure that is running in a different activation group

If your stored procedure is an original program model (OPM) program that is running at the remote system, it will always run in the default activation group and in the same commitment definition as the calling application.

Figure 6-11 shows a local stored procedure that shares the commitment definition.

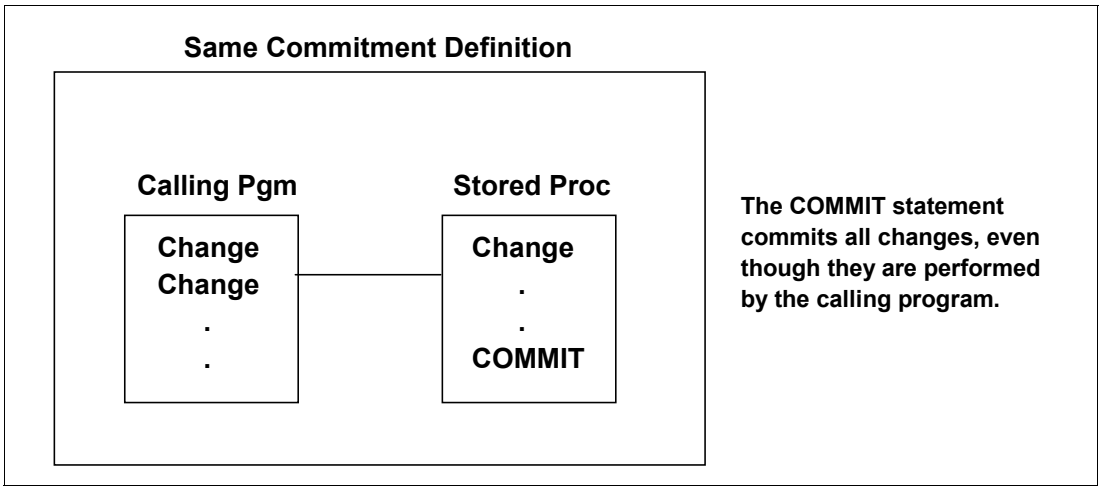


Figure 6-11 Local stored procedure that shares the commitment definition

Figure 6-12 shows a local stored procedure in a separate commitment definition.

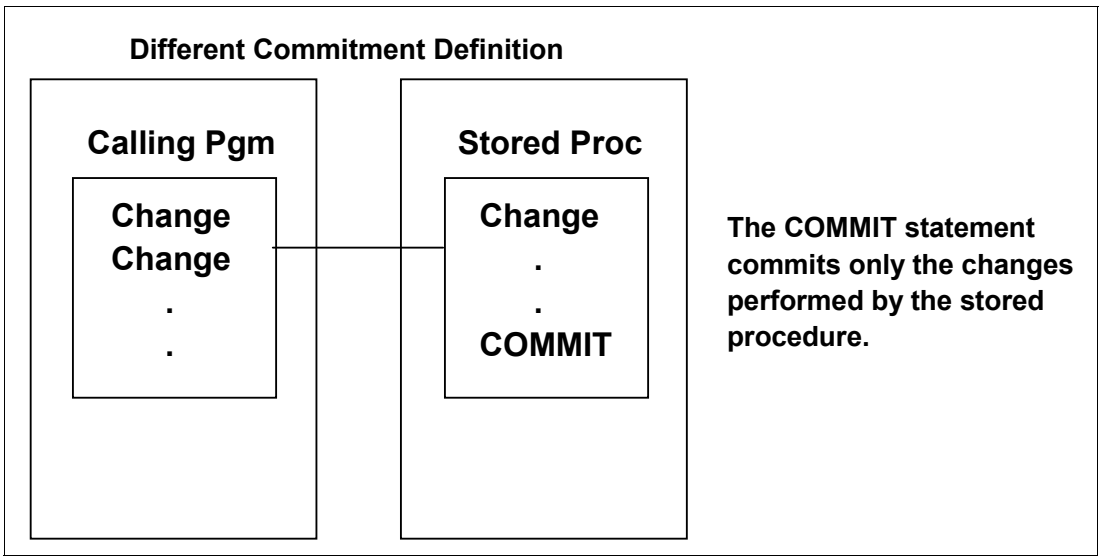


Figure 6-12 Local stored procedure in a separate commitment definition

Note: If you create an ILE program with the default compilation attributes of option 14 on the WRKMBRPDM display, your ILE program might end up running as a separate activation group. Consult the specific ILE language programmer's guide. Use a two-step compilation process, and create the module first with option 15. Then, create the program by using the `CRTPGM ACTGRP(*CALLER)` parameter.

6.7.2 Savepoints

Savepoints were 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 involve stored procedures is greatly improved, as shown in 6.6.2, "Transactional behavior" on page 180.

Important: The only supported interface to work with savepoints is through SQL statements.

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 that are 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 statements that are issued within the stored procedure affect only the changes that are performed by the stored procedure itself. The application cannot commit and roll back those changes.

6.8 Several practical examples

We present three similar versions of stored procedures:

- SQL stored procedure
- External stored procedure
- Java stored procedure

We also present two versions of client code: C++ client code and Java client code.

Because these 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.

6.8.1 SQL stored procedure example

We 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 manager's 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 whether the rule is compromised. If it is, an error condition is signaled to the calling process.

The code snippet in Example 6-24 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 following list.

Example 6-24 shows how to set user-defined errors in an external stored procedure.

Example 6-24 Setting user-defined errors in an external stored procedure

```
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);

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'                                    2
    resignal sqlstate '38S01'
          set message_text ='MODSAL: Compensation exceeds the limit.';
declare exit handler for sqlstate '02000'                                    3
    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';                                            1
end if;

update employee set salary = v_salary + i_salary where empno = i_empno;

end
```

Code sample notes

The following notes refer to Example 6-24 on page 191:

1 If the business rule is compromised, sqlstate '38S01' is signaled. The control is transferred to the error handler that is defined for this state.

2 The error handler that is 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. On 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 is truncated in SQL stored procedures.

3 The sqlstate '02000' is returned to the SQL SP if no data for the employee number is passed as the first parameter. This condition can be thrown either by the FETCH or searched UPDATE statement. The error handler handles this condition by signaling sqlstate '38S02' to the caller.

6.8.2 External stored procedure example

Example 6-25 presents an external stored procedure that is implemented in C-embedded SQL. The routine is called MODSALC. It is functionally equivalent to the SQL stored procedure MODSAL that was presented in 6.8.1, “SQL stored procedure example” on page 190.

Example 6-25 External stored procedure that is implemented in C-embedded SQL

```
#pragma  nosigtrunc
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqludf.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
    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 */                                    3
    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 */                                    1
    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 */               2
  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 6-25 on page 192:

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 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 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 that is returned to the client to determine (for example, through a table lookup) the error message to present to the user. In this case, the error message mapping is performed by the client application.

2. This error handler is a "catch-all" error handler for native SQL errors. The SQL exceptions that are not handled by the external procedure's logic are *not* passed back to the calling process. On return, the sqlstate is set to '00000', which means successful completion. Therefore, the external procedure needs to take the correct action. In our case, we set the sqlstate to '38S00', so we make sure that the procedure call fails with the -443 sqlcode. The diagnostic message is used to return native sqlcode and sqlstate.
3. The procedure checks to see whether the employee serial number that was passed as the first parameter is valid. If no data is identified for a specific employee number, sqlstate is set to '38S02', and the correct diagnostic message is returned to the caller.

6.8.3 Java stored procedure example

The code sample in Example 6-26 shows the Java implementation of the MODSAL SQL stored procedure.

Example 6-26 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 slq state 38SO2
        throw new
          SQLException("Invalid employee number.", "38SO2", -438);           2
       }
       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 38SO1
         throw new
           SQLException("Compensation exceeds the limit.", "38SO1", -438);    1
        }
       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;                                                             3
      }
  }
}
```

Code sample notes

The following notes refer to Example 6-26 on page 194:

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 that is 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. Because both codes are correctly handled by the database runtime, we opted for the solution that provides an entire diagnostic message buffer.
2. If the employee number that was passed as a parameter is invalid, an SQLException is thrown to signal that an error condition occurred in the stored procedure.
3. All potential SQL runtime errors, with the user-defined error conditions, are caught by this CATCH block and thrown again so that they are returned to the caller.

6.8.4 C++ client code that uses ODBC

Example 6-27 shows how the SQL errors that are returned from a stored procedure can be retrieved and presented in a C++ client that connects to the IBM i database server through an Open Database Connectivity (ODBC) connection.

Example 6-27 SQL errors that are presented in a C++ client that connects to the IBM i

```
#include <windows.h>
#include <SQL.h>
#include <sqlext.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)              1
   {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,                    2
       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=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)            1
   {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)           1
   {modifySalary.printError(modifySalary.ConHandle,
                               modifySalary.SpStmtHandle);}
    }
```

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

Code sample notes

The following notes apply to Example 6-27 on page 196:

1. If an error condition exists, the `printError` method is invoked to manage that error condition.
2. While SQL errors exist, retrieve them and print the relevant information.

6.8.5 Java example client code

Example 6-28 shows the Java version of the client code. It illustrates how to handle errors that are returned from the stored procedure call.

Example 6-28 Handing errors that are returned from a stored procedure call

```
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
```

6.8.6 Results for the example programs

The user-defined errors that are returned by a stored procedure are mostly consistent across all supported stored procedure types. The test results that are displayed by the sample client for different implementations of the MODSAL stored procedure are presented next. In the test scenario, we try to set the salary for the employee '000220' so that the total compensation violates the company regulation, which results in an SQL error.

Results for the MODALC (C-embedded SQL)

The results are shown:

```
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
```

In this case, the original message was truncated. The SQL return code is set to -443.

Results for MODSAL (SQL PSM)

The results are shown:

```
SQLSTATE: 38SO1
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)

The results are shown:

```
SQLSTATE: 38SO1
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.

7

Database triggers

Triggers represent one of the most powerful features of IBM DB2 for i. This chapter introduces the concept of triggers, when they can be used, and the types of triggers that are supported by DB2 for i.

This chapter describes the following topics:

- Trigger concepts
- Types of triggers in DB2 for i
- Enabling and disabling a trigger
- Displaying and reviewing trigger information
- System catalog tables
- Authorization and adopted authorities on triggers
- Renaming and copying

7.1 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, after a trigger is in place, application programmers and 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 that you designed. The application waits until the trigger ends and then gains control again.

Triggers can cause other triggers to be called. In the example that is shown in Figure 7-1, when you update TABLE1, a trigger is activated and updates the database table, TABLE2. This operation causes a second trigger to run.

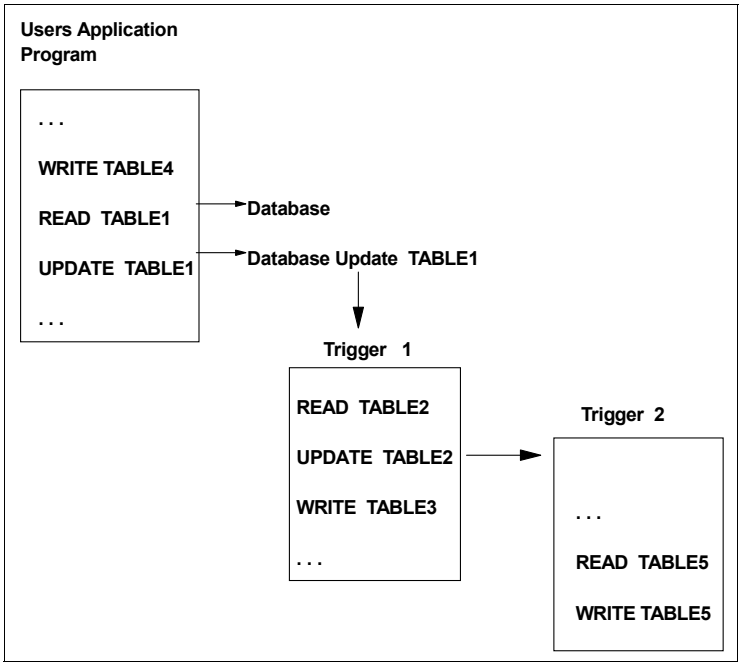


Figure 7-1 Trigger overview

It is important to identify the database tables that must be monitored and the events that need to call the triggers, remembering that a trigger is called every time that the event happens. We recommend that you think of triggers as part of your database design, rather than as a function that relates to a specific application.

You can use a trigger program for the following purposes:

- Enforcing business rules, regardless of how complex they are

  You might want to ensure, for example, that whenever you enter an order in your database, the customer that you are dealing with has no bad credit history. A trigger that is associated with the order table can check customer credit history consistently and take corrective actions.

- Validating data and maintaining an audit trail

  You might need to ensure that, whenever a sales representative enters an order, that the sales representative is assigned to that particular customer. Also, you want to track the violation attempts. Again, a trigger can be activated on the order table to validate the sales representative assignment 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 accept an order from you. Triggers can be the ideal solution to integrate your existing Order Entry application with your facsimile support on the IBM i server. Another example is to send an email to confirm the order from a customer.

- Preserving data consistency across different database tables

  In this case, triggers can complement referential integrity and check constraint support because 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 acts correctly every time that data is changed.

You can benefit from triggers for several reasons:

- Application independence

  DB2 for i activates the trigger program, regardless of the interface that you are using to access the data. Rules that are 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 that are implemented at the database level are automatically available to all applications that use 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 can provide a way to split the application logic between the client and the server system.

In Figure 7-2, you can see how client applications can take advantage of the functions that are 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 can also benefit from this implementation by reducing data traffic across communication lines.

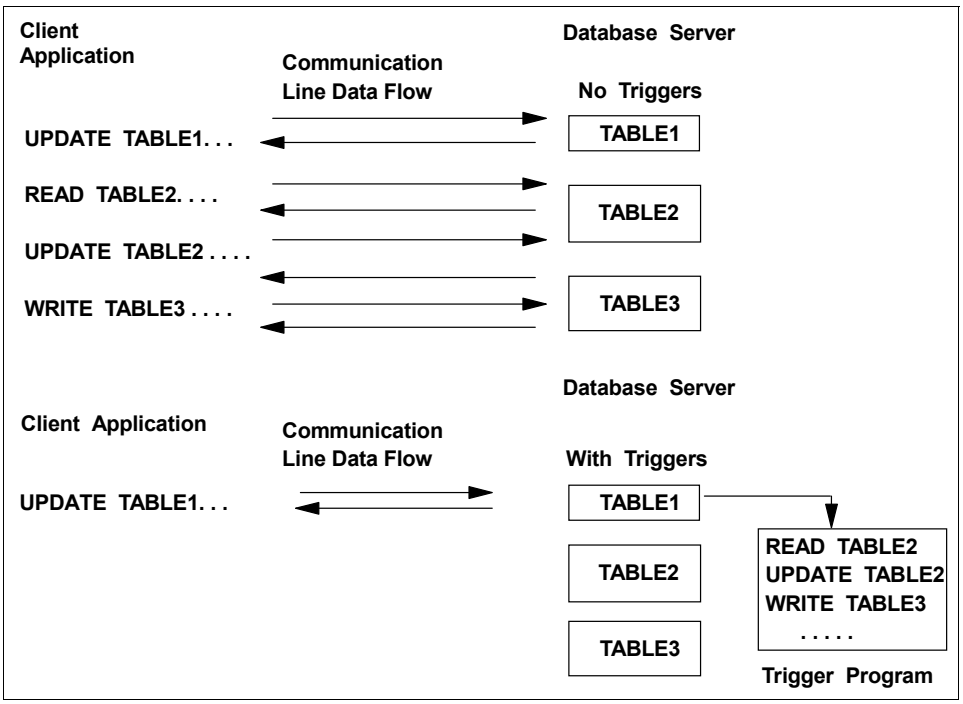


Figure 7-2 Using triggers in client/server application development

7.2 Types of triggers in DB2 for i

Two types of triggers are available in DB2 for i for database tables. Up to 300 triggers can be defined for a single table. Before V5R1, you defined up to six triggers only for a single table. The two types of triggers are shown:

- SQL triggers
- External triggers

7.2.1 SQL triggers

For an SQL trigger, the program that performs the tests and actions is written by 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 that are specified in the SQL trigger are executed each time that an insert, update, or delete operation is performed. An SQL trigger can call stored procedures or user-defined functions (UDFs) 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. For more information about SQL triggers, see *SQL Procedures, Triggers, and Functions on DB2 for i*, SG24-8326.

7.2.2 External triggers

For an external trigger, the program that contains the set of trigger actions can be defined in any supported high-level language that creates a *PGM object. SQL can be embedded in the trigger program. To define an external trigger, you must create a trigger program and add it to a table by using the Add Physical File Trigger (**ADDPFTRG**) control language (CL) command. Or, you can add it by using System i Navigator. To add a trigger to a table, you must follow these steps:

1. Identify the table.
2. Identify the kind of operation.
3. Identify the program that performs the actions that you want.
4. Provide a unique name for the trigger or let the system generate a unique name.

7.3 Enabling and disabling a trigger

Triggers need to be enabled to run. However, you can disable a trigger to work with the table without causing the trigger to run. This capability can be useful if a long batch processing task or a large data load occurs. In these cases, it can be beneficial to disable the triggers that are associated with a table.

To enable or disable a trigger, follow these steps:

1. In System i 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 7-3.

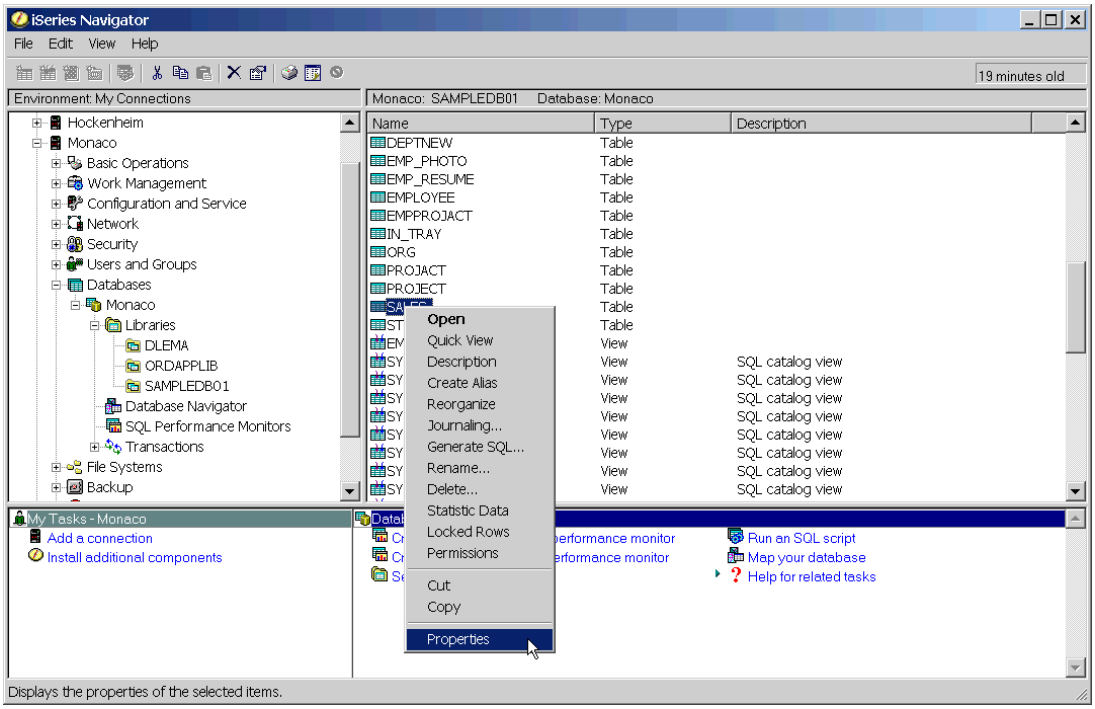


Figure 7-3 Properties table

3. In the Table Properties window, click the **Triggers** tab. Select the trigger that you want to enable or disable. Click **Enable** to enable it or **Disable** to disable the trigger, as shown in Figure 7-4.

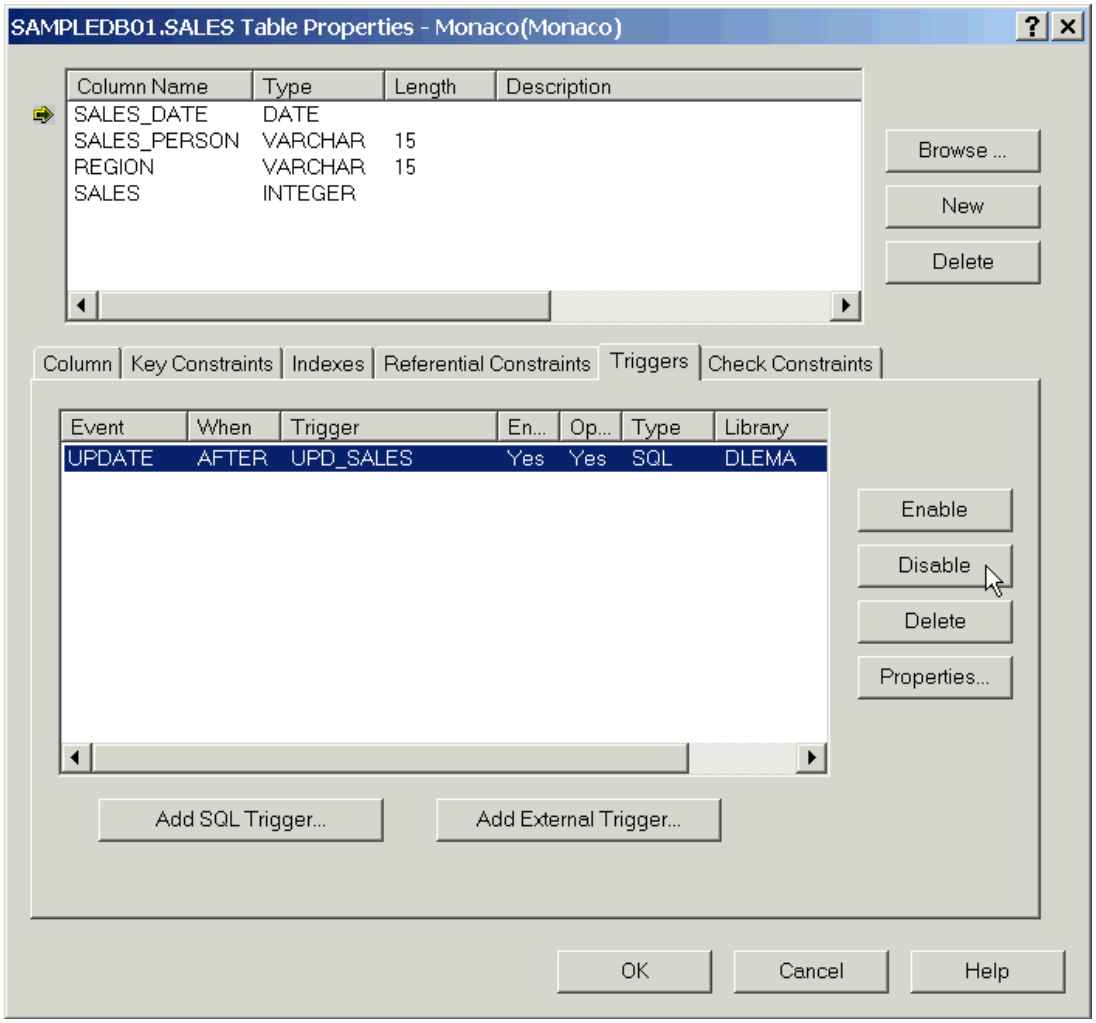


Figure 7-4 Disabling or enabling a trigger

7.4 Displaying and reviewing trigger information

You can use the following methods to assist you in visualizing trigger information:

- System i Navigator to view the properties of a trigger
- The Display File Description (**DSPFD**) command
- The Print Trigger Programs (**PRTTRGPGM**) command

7.4.1 Using System i Navigator to view the properties of a trigger

After you create and define triggers to a table, one way to visualize the properties of a trigger is to use System i Navigator:

1. In System i Navigator, double-click the **SAMPLEDB01** library. The right panel displays all DB2 for i objects in this library.
2. Find and right-click the **SALES** table, and select **Properties**, as shown in Figure 7-3 on page 207.
3. In the Table Properties window (Figure 7-5), click the **Triggers** tab. This window shows all of the triggers that the table defined.

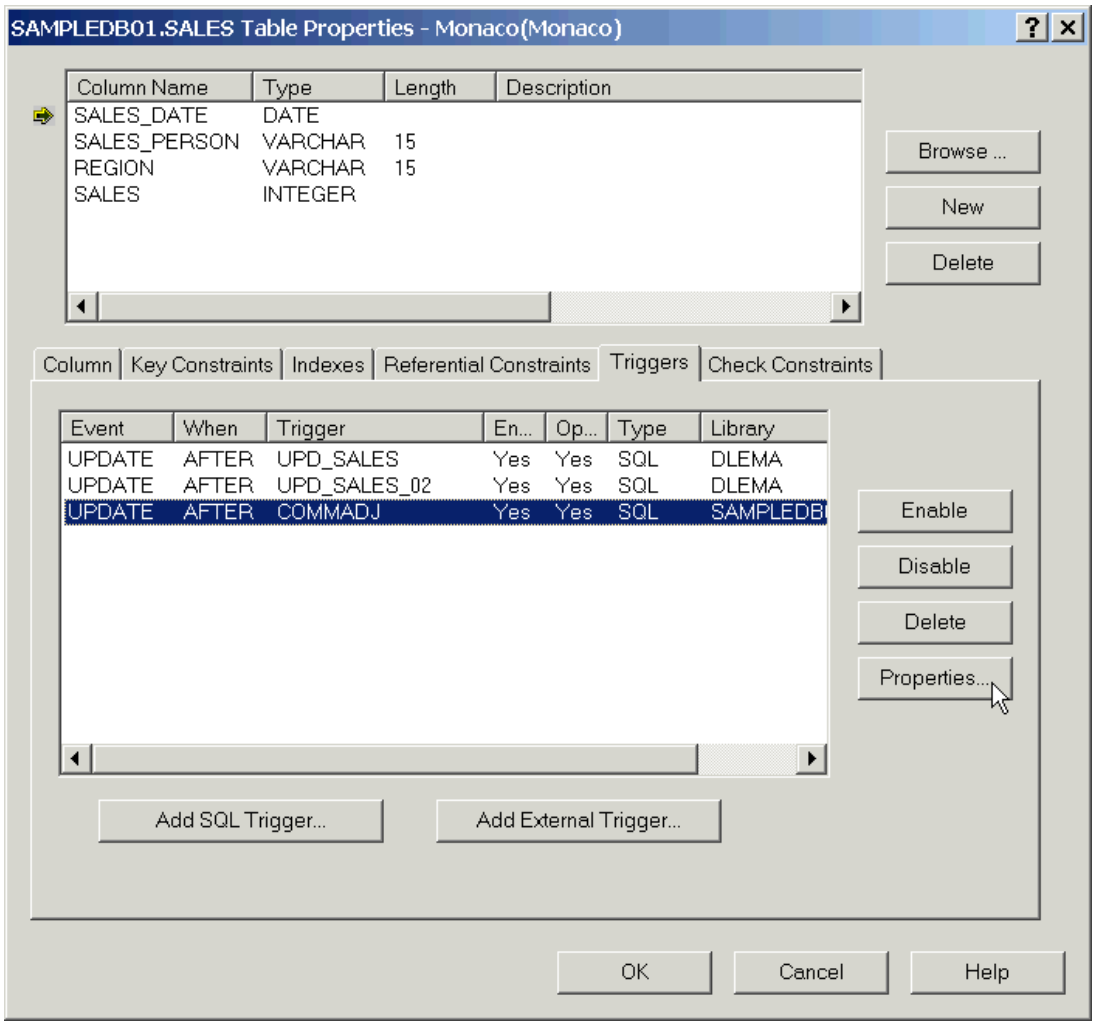


Figure 7-5 Triggers tab

Now, you can select a particular trigger to see its details.

7.4.2 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:

- Number of trigger programs
- Trigger name and library

- Trigger status
- Trigger program names and libraries
- Trigger events
- Trigger times
- Trigger update conditions
- Trigger type
- Trigger mode
- Trigger orientation
- Trigger creation date and time
- Number of trigger update columns
- List of trigger update columns

7.4.3 Printing trigger information

Use the Print Trigger Programs (**PRTTRGPGM**) command to print a report listing of all of 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 7-6 shows how the command is used for library LIB03.

```
                      Print Trigger Programs (PRTTRGPGM)

Type choices, press Enter.

Library  . . . . . . . . . . . . LIB          > LIB03
Changed report only  . . . . . . CHGRPTONLY     *NO

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

Figure 7-6 Print Trigger Programs

Figure 7-7 shows the report that is produced by the **PRTTRGPGM** command for all trigger programs in library LIB03. All of the triggers belong to the SALES table, and each trigger has a unique name.

```
                              Trigger Programs (Full Report)                                    Page     1
5722SS1 V5R1M0  010525                                                                 ASM23  09/09/01  23:17:28
 Specified library  . . . . . . :   LIB03
                                                                                                  Allow
                     Trigger            Trigger Trigger    Trigger    Trigger     Trigger     Trigger         Repeated
 Library    File     Name               Type    Library    Program    Time        Event       Condition       Change
 LIB03      SALES    XTR_COMCALC          *SYS    LIB03      COMCALC    After       Insert                      No
 LIB03      SALES    XTR_COMMADJ          *SYS    LIB03      COMMADJ    After       Update      Change          Yes
  * *  *  Full text of truncated lines * * *
   (There are no objects to list)
```

Figure 7-7 Trigger Programs report

You can use this initial report by using CHGRPTONLY(*NO) as a base to evaluate any trigger programs that exist on your system. Then, you can print the changed report by using CHGRPTONLY(*YES) regularly to see whether new trigger programs were added to your system.

7.5 System catalog tables

Four catalog views that relate to triggers are defined in QSYS2.

SYSTRIGGERS

The SYSTRIGGERS view contains one row for each trigger in an SQL schema, as shown in Figure 7-8.

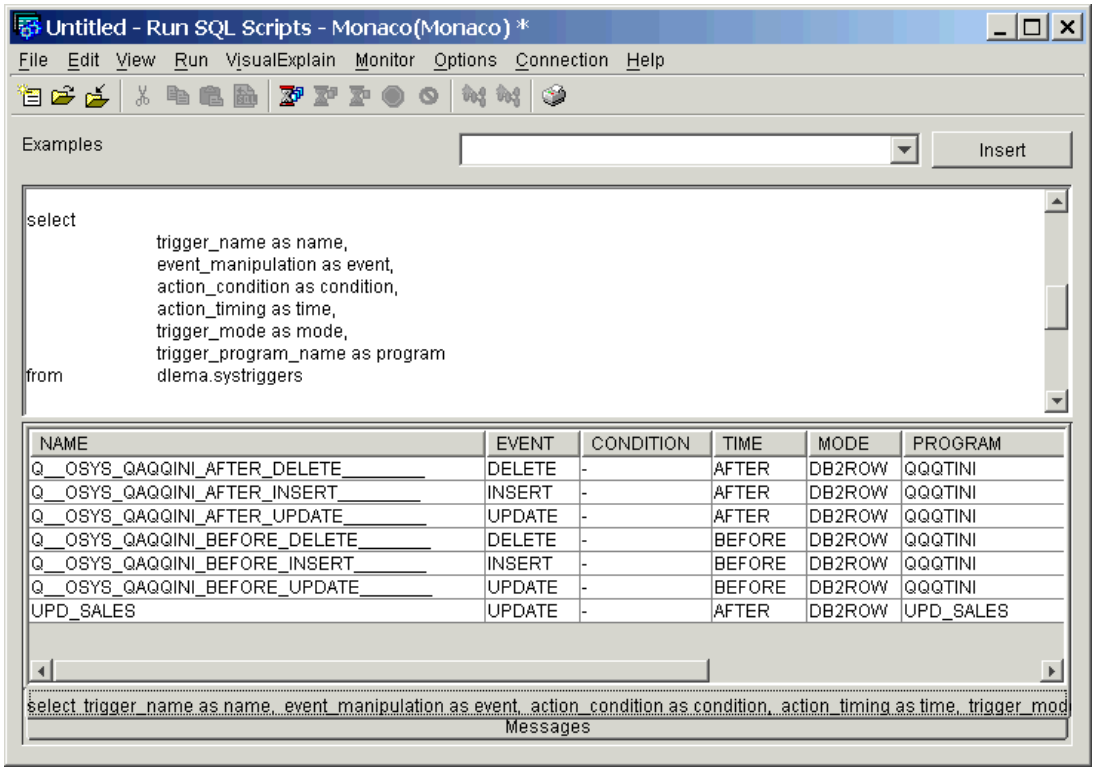


Figure 7-8 SYSTRIGGERS sample data

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 7-9.

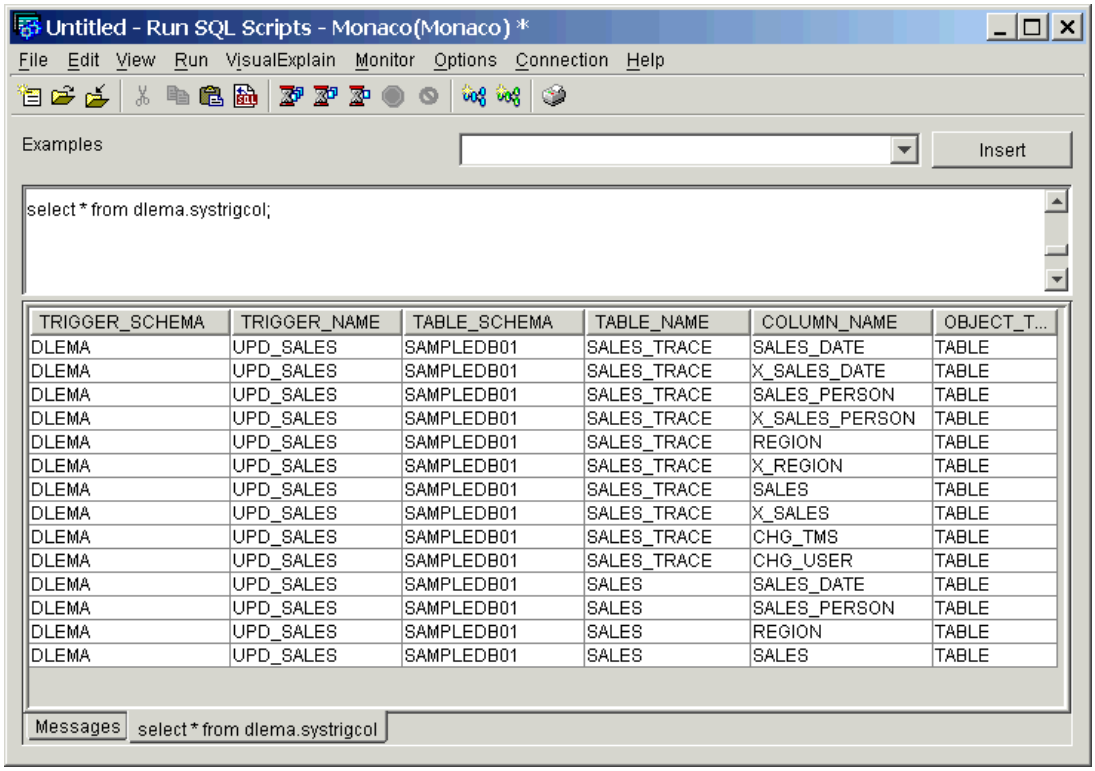


Figure 7-9 SYSTRIGCOL sample data

SYSTRIGDEP

The SYSTRIGDEP view contains one row for each object that is referenced in the WHEN clause or the triggered SQL statements of a trigger, as shown in Figure 7-10.

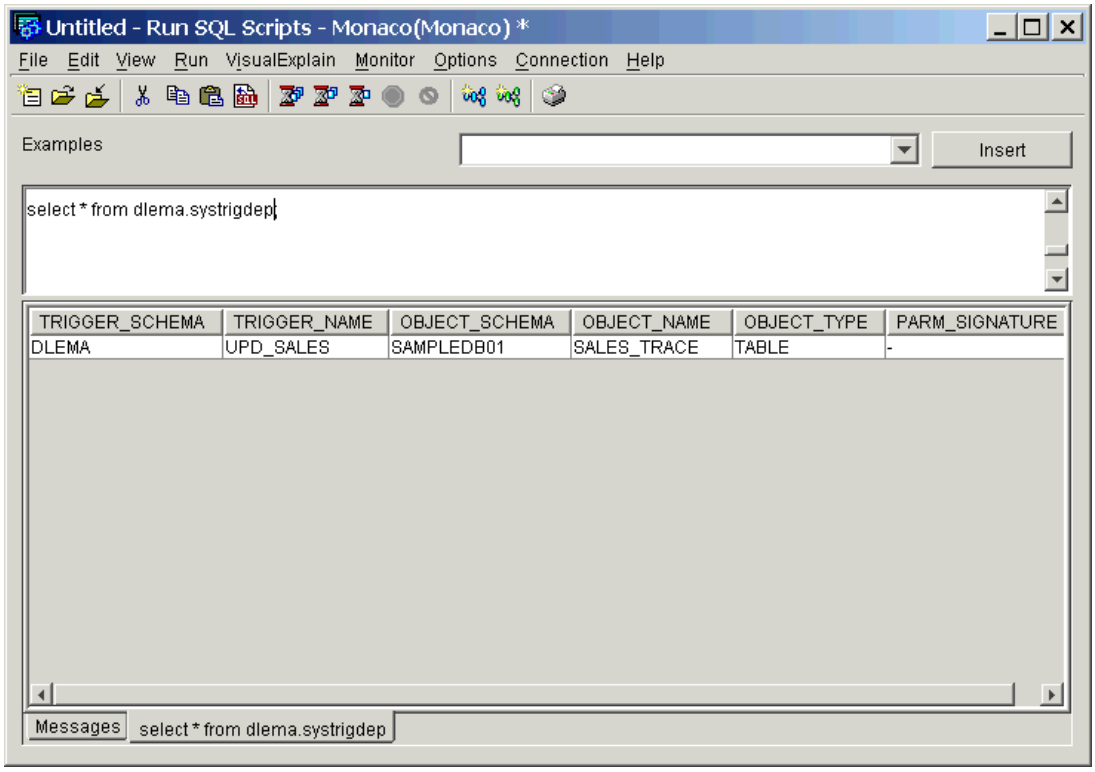


Figure 7-10 SYSTRIGDEP sample data

SYSTRIGUPD

The SYSTRIGUPD view contains one row for each column that is identified in the UPDATE column list, if any, as shown in Figure 7-11.

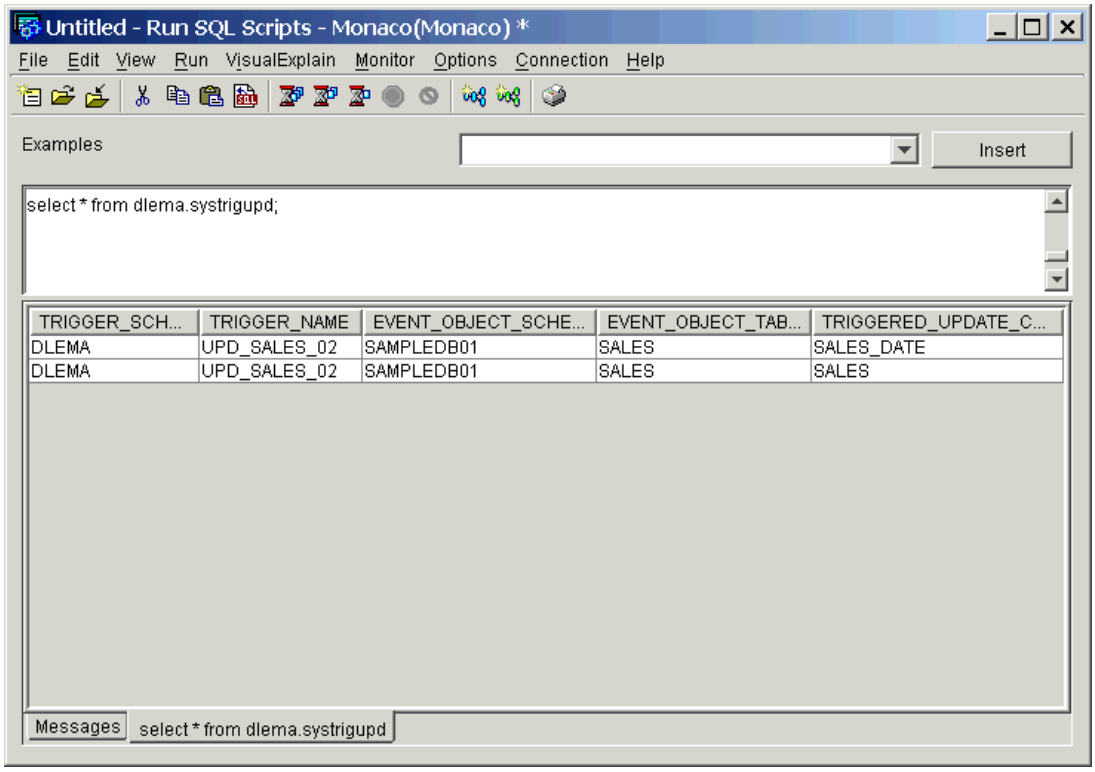


Figure 7-11 SYSTRIGUPD sample data

For more details about the information that is contained in these views, see Appendix G in *SQL Reference*, SC41-5612.

7.6 Authorization and adopted authorities on triggers

For a complete description of authorization and adopted authorities, see *SQL Reference*, SC41-5612. Different behavior might be associated with *SQL* and *system naming conventions*. 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.

A trigger will execute with the adopted authority of the *owner* of the trigger.

For the deployment of triggers, all users need to be able to run most triggers. Therefore, we recommend that you generate trigger programs with the authorization set to *OWNER and by using an owner profile with authorities over the resources that are affected by them.

7.7 Renaming and copying

Any table (including the subject table) that is 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 need to drop the trigger and 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 that is associated with it, the trigger program is called each time that 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 you copy records to a table with an *INSERT trigger program associated with it:

1. Use the **CHGPFTRG** command or System i Navigator to *disable* all triggers that are associated with the target table.
2. Use the **CPYF** command to copy the records to the target table.
3. Use the **CHGPFTRG** command or System i Navigator to *enable* all triggers that are associated with the target table.

Note: The **CHGPFTRG** command will fail if the table is open.

8

External triggers

External triggers are a powerful feature in IBM DB2 for i. 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 describes the following topics:

- Defining a trigger
- Trigger program structure
- Trigger feedback to application programs
- Designing trigger programs
- Applications and triggers: Design considerations
- Recommendations

8.1 Defining a trigger

On the IBM i server, a trigger program can be developed by using any supported high-level language (HLL) compiler. You can include SQL statements or any other high-level language. You can also code a trigger by using the CL. See Figure 8-1.

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 multiple member physical file, such as a yearly sales file with a member for each month, the trigger is activated whenever data is modified in any member.

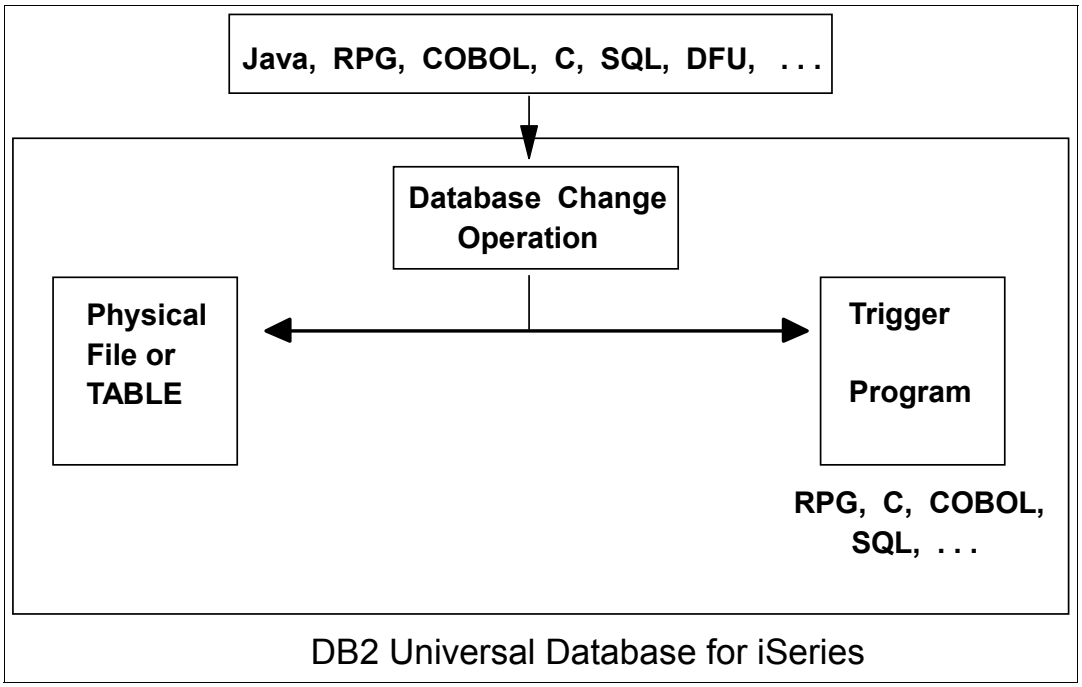


Figure 8-1 Activating triggers on the IBM i server

You need to specify the following information when you add a trigger program to a database table:

- The *trigger event* is the I/O operation that activates the trigger:
  - Insert
  - Update
  - Delete
  - Read (applies to external triggers only)
- The *trigger time* determines whether the trigger is activated before or after the trigger event takes place. Late, we describe in detail how this parameter influences the behavior of DB2 for i. See 9.5, “Constraints and triggers: Ordering the actions” on page 306.
  - Before the operation: Before an update, an insert, or a delete operation
  - After the operation: After an update, an insert, or a delete operation
- The *trigger program* is activated for this type of I/O operation.

- *Replace trigger*:
  - Option (*YES): The trigger program is replaced if another trigger has the same specification.
  - Option (*NO): The trigger program is added if no other triggers have 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 shown:
  - *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.
- The *trigger condition* parameter is relevant to UPDATE triggers only. The options are shown:
  - *CHANGE: The trigger runs only if the update operation 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 was changed.
- The *trigger* parameter is relevant to all triggers. The options are shown:
  - *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 double quotation mark (") delimiters. The case is preserved when lowercase characters are specified.

Triggers can be added to database tables by using a control language (CL) command or by using System i Navigator.

8.1.1 ADDPFTRG

The Add Physical File Trigger (**ADDPFTRG**) command associates a trigger program with a physical file. When this association is established, DB2 for i calls the trigger program when a change operation is performed against the physical file, a member of the physical file, and any logical file that was created over the physical file or views that were created by SQL.

See Figure 8-2 and Figure 8-3.

```
                    Add Physical File Trigger (ADDPFTRG)

Type choices, press Enter.

Physical file  . . . . . .   ORDERHDR      Name
  Library  . . . . . . . .     ORDENTL     Name, *LIBL, *CURLIB
Trigger time . . . . . . .   *BEFORE       *BEFORE, *AFTER
Trigger event  . . . . . .   *INSERT       *INSERT, *DELETE, *UPDATE...
Program  . . . . . . . . .   T4249IADT     Name
  Library  . . . . . . . .     ORDENTLIB   Name, *LIBL, *CURLIB
Replace trigger  . . . . .   *NO           *NO, *YES
Trigger  . . . . . . . . .   *GEN

Trigger library  . . . . .   *FILE         Name, *FILE, *CURLIB
Allow Repeated Change  . .   *NO           *NO, *YES
                                                                  More...
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
```

Figure 8-2 Add Physical File Trigger (Part 1 of 2)

```
                    Add Physical File Trigger (ADDPFTRG)

Type choices, press Enter.

Threadsafe . . . . . . . . . . THDSAFE        *UNKNOWN
Multithreaded job action . . . . MLTTHDACN    *SYSVAL
Trigger update condition . . . . TRGUPDCND    *ALWAYS


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

Figure 8-3 Add Physical File Trigger (Part 2 of 2)

When you add the trigger to the physical file, the file description is updated to reflect that a trigger is 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 if you need to change your business rules. Simply recompile the trigger program. You do not need to modify any applications or change data in your database. All applications that access 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 8-4).

```
                    Add Physical File Trigger (ADDPFTRG)

Type choices, press Enter.

Physical file  . . . . . . . . . FILE        > ORDERHDR
  Library  . . . . . . . . . . .             >   ORDENTL
Trigger time . . . . . . . . . . TRGTIME     > *BEFORE
Trigger event  . . . . . . . . . TRGEVENT    > *INSERT
Program  . . . . . . . . . . . . PGM         > T4249IADT
  Library  . . . . . . . . . . .               *LIBL
Replace trigger  . . . . . . . . RPLTRG        *NO
Trigger  . . . . . . . . . . . . TRG           *GEN
Trigger library  . . . . . . . . TRGLIB        *FILE
Allow Repeated Change  . . . . . ALWREPCHG     *NO
 Threadsafe . . . . . . . . . . . THDSAFE       *UNKNOWN
Multithreaded job action . . . . MLTTHDACN     *SYSVAL

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

Figure 8-4 Add Physical File Trigger (combined)

In this example, the file description reports the actual library where the trigger is stored. By using DSPFD TYPE(*TRG), the trigger library is explicitly reported (Figure 8-5).

```
   9/05/01               Display File Description
DSPFD Command Input
  File  . . . . . . . . . . . . . . . . . . . . : FILE       SALES
    Library . . . . . . . . . . . . . . . . . . :            LIB03
  Type of information . . . . . . . . . . . . . : TYPE       *TRG
  File attributes . . . . . . . . . . . . . . . : FILEATR    *ALL
  System  . . . . . . . . . . . . . . . . . . . : SYSTEM     *LCL
File Description Header
  File  . . . . . . . . . . . . . . . . . . . . : FILE       SALES
  Library . . . . . . . . . . . . . . . . . . . :            LIB03
  Type of file  . . . . . . . . . . . . . . . . :            Physical
  File type . . . . . . . . . . . . . . . . . . : FILETYPE   *DATA
  Auxiliary storage pool ID . . . . . . . . . . :            01
Trigger Description
  Trigger name  . . . . . . . . . . . . . . . . : TRG        XTR_COMCALC
    Trigger library . . . . . . . . . . . . . . :            LIB03
Trigger state . . . . . . . . . . . . . . . . . : STATE      *ENABLED
  Trigger status  . . . . . . . . . . . . . . . :              *OPERATIVE
  Trigger event . . . . . . . . . . . . . . . . : TRGEVENT   *INSERT
  Trigger time  . . . . . . . . . . . . . . . . : TRGTIME    *AFTER
  Allow repeated change . . . . . . . . . . . . : ALWREPCHG  *NO
  Program Name  . . . . . . . . . . . . . . . . : PGM        COMCALC
    Library . . . . . . . . . . . . . . . . . . :            LIB03
  Program is threadsafe . . . . . . . . . . . . : THDSAFE    *UNKNOWN
  Multithreaded job action  . . . . . . . . . . : MLTTHDACN  *SYSVAL
  Trigger type  . . . . . . . . . . . . . . . . :            *SYS
  Trigger orientation . . . . . . . . . . . . . :            *ROW
  Trigger creation date and time  . . . . . . . :            09/04/01  04:38:30
  Number of trigger update columns  . . . . . . :                0
```

Figure 8-5 Display File Description

You can define up to 300 trigger programs for the same database table: *BEFORE and *AFTER insert, delete, update, or read operations.

8.1.2 Using System i Navigator to add an external trigger

By using System i 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 System i Navigator window, expand ***your server*** → **Database**.
2. Choose the database that 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 System i Navigator windows show the steps for adding a system (external) trigger to a database. Figure 8-6 shows the General tab information that is required to add the trigger. The data is similar to the data that is required on the **ADDPFTRG** command (Figure 8-2 on page 220 and Figure 8-3 on page 220).

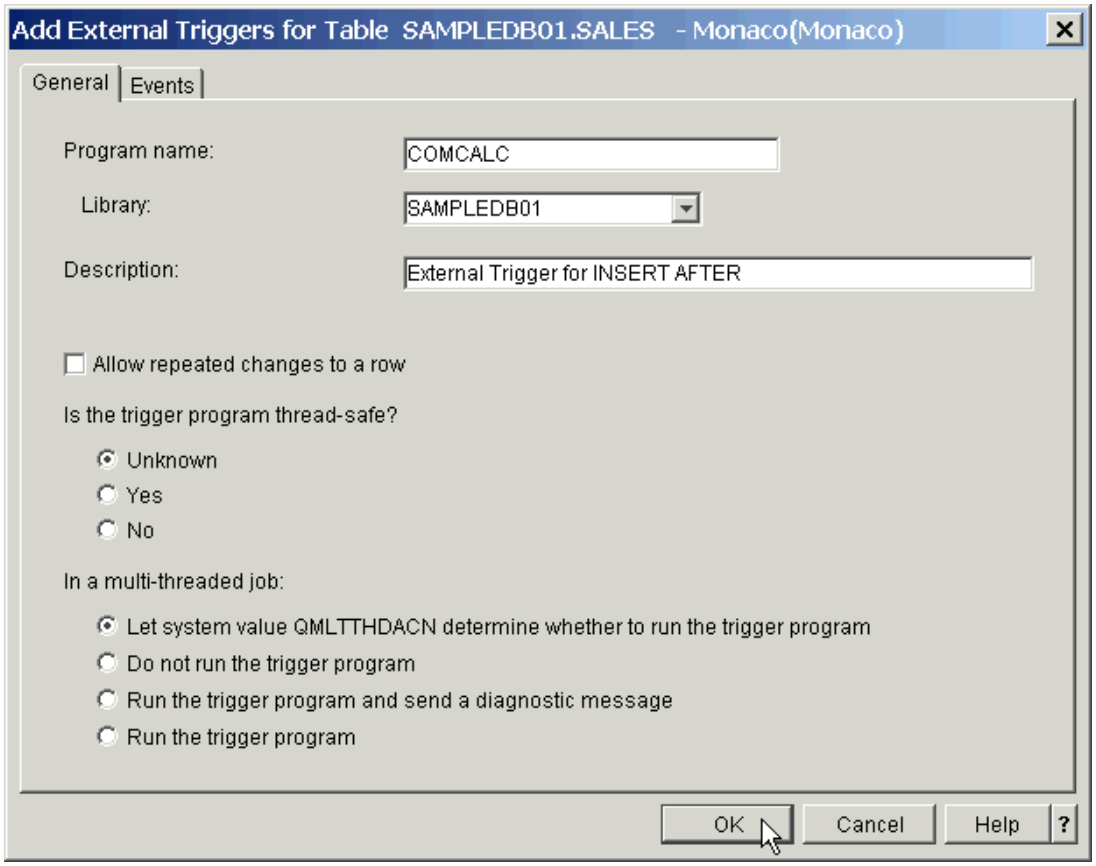


Figure 8-6 Add External Triggers: General tab

Figure 8-7 shows the required information in the Events tab for a trigger. A trigger name and a library name for the trigger are additional information that is needed for a trigger. If this information 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 added.

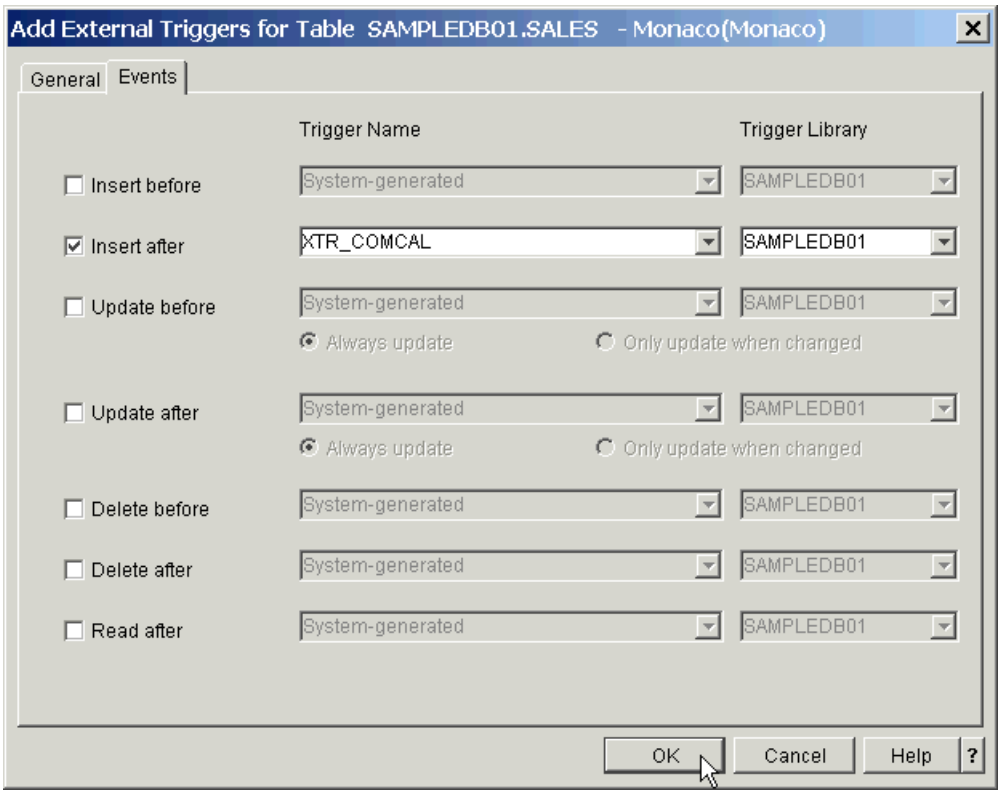


Figure 8-7 Add External Triggers: Events tab

Figure 8-8 shows that the INSERT AFTER trigger was added to the list of triggers for table SALES in library SAMPLEDB01. It also shows a list of all of the triggers for the table.

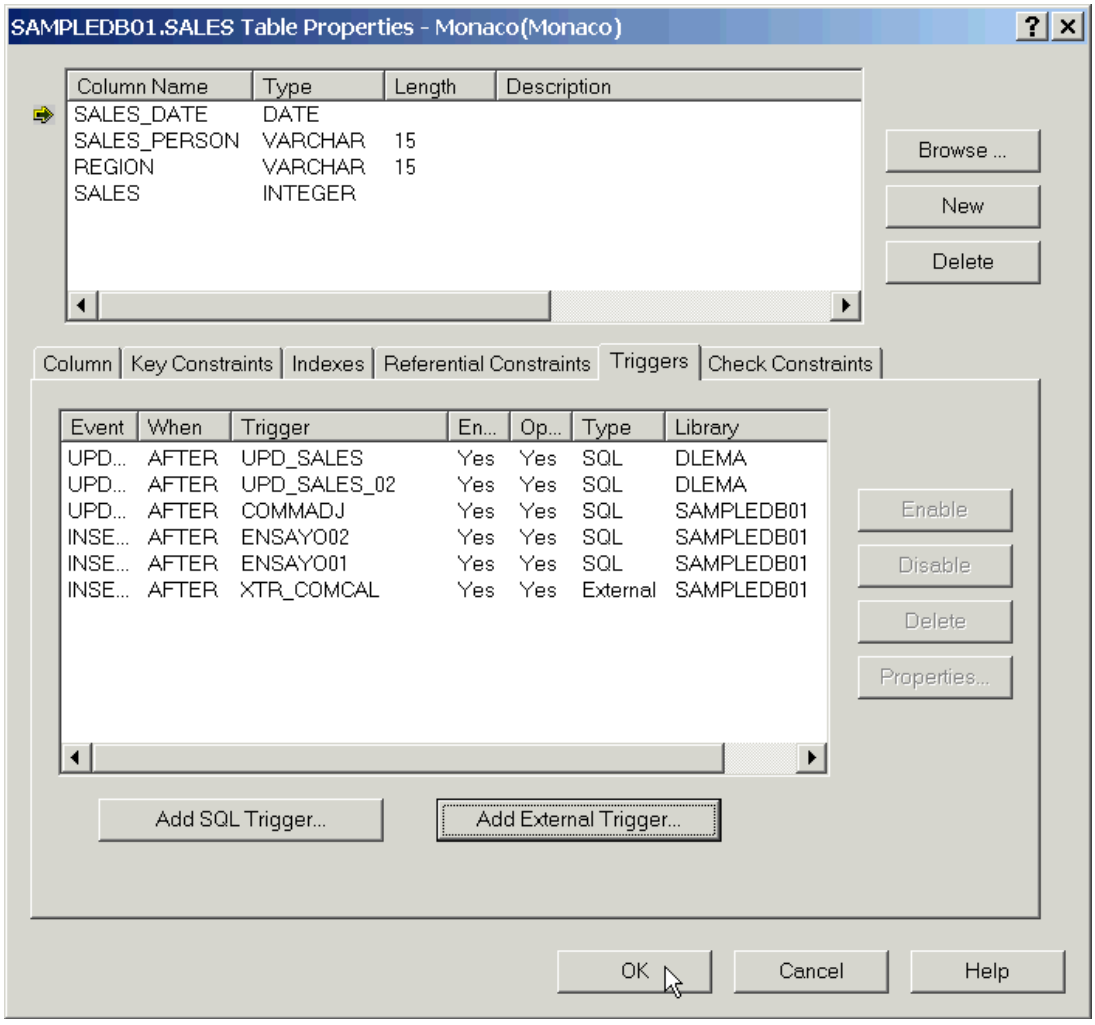


Figure 8-8 List of triggers for a table

8.2 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 coded character set identifier (CCSID) of the current change record and relative record number
    - Offsets and lengths of the record areas and null byte maps

    This area occupies (in decimal) offset 0 - 95.
  - Dynamic area

    Areas for the old record and old null byte map, new record, and new null byte map
- Trigger buffer length: The length of the trigger buffer that is provided by DB2 Universal Database for iSeries

By defining these parameters in your trigger programs, you can take the correct actions based on the kind of data change that occurred and the characteristics of the job that fired the trigger. Our code samples show how you can use the information that is passed through the trigger parameter list. See 8.4, “Designing trigger programs” on page 242.

Table 8-1 describes the trigger buffer structure.

Note: The way that 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. Another technique is called *softcoding the trigger buffer*. In this case, when the structure of the file changes, you only need to recompile the trigger program to access the new layout. We show this technique in 8.4.4, “Softcoding the trigger buffer example” on page 280.

Table 8-1 The trigger buffer structure

Decimal offset	Parameter	Type	Description
0	Physical file name	char(10)	The physical file that is changed.
10	Physical file library name	char(10)	The library in which the physical file resides.
20	Physical file member name	char(10)	The name of the physical file member.
30	Trigger Event	char(1)	The event that caused the trigger program to be called. The possible values can be “1” (Insert), “2” (Delete), “3” (Update), or “4” (Read).
31	Trigger Time	char(1)	Can be “1” (After) or “2” (Before).
32	Commit level	char(1)	Reports the commit lock level of the interface that activated the trigger. The values are “0” (*NONE), “1” (*CHG), “2” (*CS), or “3” (*ALL).
33	Reserved	char(3)	Reserved.
36	CCSID of data	binary(4)	The CCSID of the data in the new or the original records. The data is converted to the job CCSID by the database.
40	Relative record number	binary(4)	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).
44	Reserved	char(4)	Reserved.
48	Original record offset	binary(4)	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.
52	Old record length	binary(4)	The maximum length is 32,766 bytes.
56	Old record null map offset	binary(4)	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.
60	Old record null map length	binary(4)	The length is equal to the number of fields in the physical file.
64	New record offset	binary(4)	The location of the new record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the new value of the record does not apply to the change operation, for example, a delete operation.
68	New record length	binary(4)	The maximum length is 32,766 bytes.

Decimal offset	Parameter	Type	Description
72	New record null map offset	binary(4)	The location of the null byte map of the new record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the new value of the record does not apply to the change operation, for example, a delete operation.
76	New record null map length	binary(4)	The length is equal to the number of fields in the physical file.
80	Reserved	char(16)	Reserved.
*	Original record	char(*)	A copy of the original physical record before it is updated, deleted, or read. The original record applies only to update, delete, and read operations.
*	Original record null byte map	char(*)	This structure contains the NULL value information for each field of the original record. Each byte represents one field. The possible values for each byte are “0” (Not NULL) or “1” (NULL).
*	New record	char(*)	A copy of the record that is inserted or updated in a physical file as a result of the change operation. The new record applies only to the insert or update operations.
*	New record null byte map	char(*)	This structure contains the NULL value information for each field of the new record. Each byte represents one field. The possible values for each byte are “0” (Not NULL) or “1” (NULL).

Important: When the support for the large object (LOB) data types (binary large object (BLOB), character large object (CLOB), and double-byte character large object (DBCLOB)) was added in V4R4, the support came with a minor restriction. That is, you were not able to define triggers 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 that is passed as input to all external trigger programs due to system infrastructure changes. Trigger programs, which either presumed the overall trigger buffer length will not change (for example, placing the entire entry in a permanent/unchanging data queue) or hardcoded their usage of the trigger buffer parameters instead of correctly using the offsets and lengths that are passed in the trigger buffer, are not affected.

However, for trigger programs that were coded correctly 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) by using the offsets and lengths will 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).

8.2.1 Trigger buffer for RPG

Figure 8-9 shows an example of how you can define the trigger buffer.

```
 *====================================================*
 * Definition of the structure to be passed into the  *
 * trigger program = buffer                           *
 *====================================================*
IPARM1       DS
I                                        1  10 FNAME
I                                       11  20 LNAME
I                                       21  30 MNAME
I                                       31  31 TEVEN  1
I                                       32  32 TTIME
I                                       33  33 CMTLCK 2
I                                       34  36 FILL1
I                                    B  37  40OCCSID
I                                       41  48 RRN    3
I                                    B  49  52OOLDOFF
I                                    B  53  56OOLDLEN 4
I                                    B  57  60OONOFF
I                                    B  61  64OONLEN
I                                    B  65  68ONOFF
I                                    B  69  72ONEWLEN 5
I                                    B  73  76ONNOFF
I                                    B  77  80ONNLEN
I                                       81  96 RESV3
I                                       97 142 OREC   6
I                                      143 148 OOMAP  7
I                                      149 194 RECORD 8
I                                      195 200 NNMAP  9
 *====================================================*
 * LENG  = buffer length                              *
 *====================================================*
IPARM2       DS
I                                    B   1   4OLENG
```

Figure 8-9 Trigger buffer for RPG programs

Notes: The following notes refer to the numbers in Figure 8-9:

1 Trigger event
2 Trigger commit level
3 Relative record number
4 Old record length
5 New record length
6 Old record image
7 Old record null map
8 New record image
9 New record null map

8.2.2 Trigger buffer for COBOL

Figure 8-10 shows how you can define the trigger buffer in a COBOL program.

```
*================================================*
* 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 8-10 Trigger buffer for a COBOL program

Notes: The following notes refer to the numbers in Figure 8-10:

1 Trigger event
2 Trigger commit level
3 Relative record number
4 Old record length
5 New record length
6 Old record image
7 Old record null map
8 New record image
9 New record null map

8.2.3 Trigger buffer for C

An include file is in the QSYSINC library for most of the system application programming interfaces (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 8-11 shows the trigger buffer definition for a C program that was taken from the TRGBUF member.

```
/*************************************************************/
/*   INCLUDE NAME : TRGBUF                                   */
/*                                                           */
/*   DESCRIPTION  : The input trigger buffer structure for the  */
/*                  user's trigger program.                  */
/*                                                           */
/*   LANGUAGE     : ILE C                                    */
/*                                                           */
/*************************************************************/
/*************************************************************/
/*   Note: The following type definition only defines the fixed */
/*         portion of the format. The data area of the original */
/*         record, null byte map of the original record, the    */
/*         new record, and the null byte map of the new record  */
/*         is varying length and immediately follows what is    */
/*         defined here.                                        */
/*************************************************************/
 typedef _Packed struct Qdb_Trigger_Buffer {
           char  file_name[10];
           char  library_name[10];
           char  member_name[10];
           char  trigger_event[1]; 1
           char  trigger_time[1];
           char  commit_lock_level[1]; 2
           char  reserved_1[3];
           int   data_area_ccsid;
           int   Current_Rrn;    3
           char  reserved_2[4];
           int   old_record_offset;6
           int   old_record_len;    4
           int   old_record_null_byte_map;7
           int   old_record_null_byte_map_len;
           int   new_record_offset;8
           int   new_record_len;   5
           int   new_record_null_byte_map;9
           int   new_record_null_byte_map_len;
                 } Qdb_Trigger_Buffer_t;
```

Figure 8-11 Trigger buffer for a C program

Notes: The following notes refer to the numbers in Figure 8-11 on page 231:

1 Trigger event
2 Trigger commit level
3 Relative record number
4 Old record length
5 New record length
6 Old record image offset
7 Old record null map offset
8 New record image offset
9 New record null map offset

While we use C, we are not allowed to explicitly define 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 because of the different memory allocation techniques of the C language. If you use C, set a pointer to the record images by using the offsets. You can access the contents of these areas.

8.2.4 Using the trigger buffer

The following descriptions refer to the most important fields in the trigger buffer, as marked in the previous examples:

- Trigger Event 1: This field gives you the capability to determine the event that called the trigger. This information is valuable when a trigger is defined for different events. You might want to identify the record image to use, depending on the event that activated the trigger. The system always initializes the offset fields, even if one of them might address meaningless data. Table 8-2 shows the record images that you receive, depending on the event.

  Table 8-2 Record images and trigger events

Trigger event	Images
*INSERT *UPDATE *DELETE	NEW RECORD NEW/OLD RECORD OLD RECORD

- Trigger CMTLVL 2: This field is the commit lock level of the application that caused the trigger to run. Typically, 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.

  Several ways are available to set an isolation level for triggers, depending on the language that you use. For SQL triggers, you can use the SET TRANSACTION SQL statement. If both SQL and native data access are in your program, SET TRANSACTION affects only the SQL statements. Access to data through native interfaces is not affected by SET TRANSACTION. For information about commitment control and commit lock levels, see *Backup and Recovery*, SC41-5304.

  For native trigger programs, different considerations apply. By using C, you can dynamically open a file with or without commitment control. Integrated Language Environment (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 original program model (OPM) languages do not provide a way to dynamically define commitment control for files. Several ways exist to circumvent this limitation. For example, you can define the database files in your program twice, one time without and one time with commitment control.

Open the correct file definition according to the value that is retrieved in the trigger buffer, for example, by 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  2
                  OPEN I-O FILDEFB
           ELSE
                  OPEN I-O FILDEFA.
```

You can also see the example that is provided in “Invoice trigger example in ILE RPG” on page 270, where an Integrated Language Environment (ILE) RPG program that handles dynamic commitment control is shown. For a full description of the commitment control options of the various programming languages, consult the specific language user’s guide and reference.

Example 8-1 shows part of an RPG trigger program with embedded SQL statements that shows how to set the correct commit lock level.

Example 8-1 RPG trigger program to set the 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 the *SQL Programming Guide*, SC41-5611.

- Trigger relative record number 3: This value 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:

```
select * from ordentlib.orderhdr where  rrn(ordentlib.orderhdr) = :relnum
```

- Trigger old record image 5: When a trigger is activated by an update or a delete operation, the old record image is in this parameter field. In COBOL and RPG, define a storage area with the same length as the database record length.
- Trigger old null record map 6: This value 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 that is associated to the trigger program. DB2 for i can set each character to 1 (NULL field) or 0 (not NULL).
- Trigger new record image 7: When a trigger is activated by an update or an insert operation, the new record image is in this parameter field. In COBOL and RPG, define a storage area with the same length as the database record length.
- Trigger new null record map 8: This value 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 that is associated with the trigger program. DB2 Universal Database for iSeries can set each character to 1 (NULL field) or 0 (not NULL).

  If you use variable length fields, the length of the record images that are provided by DB2 for i is the maximum length that is allowed for the database records. Variable length character fields are padded with blanks, and a 2-byte binary field with the actual data length is added in front of every VARCHAR field.

8.3 Trigger feedback to application programs

When you implement your trigger program, you must consider that triggers cannot pass parameters back directly because trigger programs are activated by the database manager and they are given an input-only parameter list. If a failure occurs while the trigger program is running, an appropriate escape message must be signaled before the trigger terminates. The message can be the original message that is signaled by the system or a user-defined message that is retrieved from a message file by the trigger program.

If no error message is signaled to the calling program after a trigger fails, the database manager assumes that the trigger completed successfully and the operation that activated the trigger completed, also.

We differentiate between the two cases:

- System-generated error, such as a failure that is encountered when 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 to search for an appropriate exception handler. If an exception handler does not handle the exception, the exception is processed by the system database module that performed the I/O operation that fired the trigger. The I/O operation fails.

- Failures that are detected by the trigger program

  This situation is common in data validity checking. For example, consider the case of an insert trigger that checks the records that are inserted. When a record is inserted, the OS/400 module QDBPUT is invoked (Figure 8-16 on page 237). The trigger is displayed in the invocation stack after the OS/400 module QDBPUT.

  If the trigger determines that invalid data is inserted, the insert operation must be rejected. We can achieve this rejection 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 can 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 that is signaled by triggers, depending on the language that you use, the return codes differ:

- SQL application:

  ```
  SQLCODE = -443
  ```

  This code 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:

  ```
  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 that reports the exception identifier is set to "CPF502B".

User messages that are sent by the triggers through the QMHSNDPM API and the CPF502B message are always in the job log. If the application that activates the trigger happens to run in an interactive job, the trigger might send a message to the display. We show an example of this technique in "Audit trail trigger example in COBOL SQL (OPM)" on page 250.

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, after an exception is handled, the trigger ends normally. If you need to reject the change operation that fired the trigger, you must signal an escape message from your exception handler to the correct 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 IBM i server, such as Data File Utility (DFU) and Interactive SQL. The actual message that is sent to these interfaces is generic in both cases. The user message that is sent by the trigger is in the job log.

An Interactive SQL insert operation that fails after a trigger signaled an escape message is shown in Figure 8-12.

```
 > UPDATE ORDENTL/ORDERHDR SET CUSTOMER_NUMBER = '00005' WHERE
    ORDER_NUMBER = '12312'
    Trigger program or external procedure detected an error.
```

Figure 8-12 Escape message that is signaled by a trigger

The SQL interface sends the correct 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 8-13).

```
 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 8-13 User-defined message that is signaled by a trigger

Similarly, if you use DFU, the operation that you perform fails, and the generic message CPF502B is reported to the interface (Figure 8-14).

```
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 8-14 DFU session

After you press Enter on this display, you see the exception that is shown in Figure 8-15.

```
                              End Data Entry

  Number of records processed

    Added  . . . . . :          1
    Changed  . . . . :          0
    Deleted  . . . . :          0


Type choice, press Enter.

  End data entry  . . . . . .    Y           Y=Yes,  N=No

  F3=Exit       F12=Cancel
Message CPF502B was issued.
```

Figure 8-15 Escape message that is signaled to DFU

It is worthwhile to go into more detail to show how the process of signaling a message to the database interface works. The example in Figure 8-16 relates to our application. We are reproducing the invocation stack of the application after the trigger is activated.

```
                            Job Call Stack

     Request  Program or
Opt   Level   Procedure       Library     Statement     Instruction
              QCMD            QSYS                        0351
              QUICMENU        QSYS                        00C1
         1    QUIMNDRV        QSYS                        0455
         2    QUIMGFLW        QSYS                        0483
         3    QUICMD          QSYS                        03E4
              QUOCPP          QPDA                        0541
              QUOMAIN         QPDA                        0FDD
         4    QUOCMD          QSYS                        0176
1           T4249CINS       ORDENTLIB   136             00D9
              QSQROUTE        QSYS                        02F0
              QSQINS          QSYS                        01C0
2           QDBPUT          QSYS                        0193
3           T4249RADT       ORDENTLIB   .GET            021D
```

Figure 8-16 Invocation stack with an OPM trigger

Notes: The following notes refer to the numbers in Figure 8-16:

1 Application program Order Header Entry inserts a record.
2 This module is the OS/400 module for the insert operation.
3 This program 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. Several of the messages on the call message queue are also recorded in the job log. When the trigger is called, the application that caused its invocation waits until the trigger ends. The trigger becomes part of the application flow. The trigger is shown in the invocation stack of the job as though the application called it.

The statements in Figure 8-17 from a COBOL program show the parameters to pass to the QMHSNDPM API to send a message to the DB module that performs the I/O operation.

```
01 SNDPGMMSG.
   03 SND-MSG-ID         PIC X(7)  VALUE "TRG0005". 4
   03 SND-MSG-FILE       PIC X(20) VALUE "ORDMSGF   ORDENTLIB".4
   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 "*". 5
   03 SND-PGM-STACK      PIC 9(8) BINARY VALUE 1. 6
   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 " ".
```

Figure 8-17 Parameters of the QMHSNDPM API

The following notes refer to the numbers in Figure 8-17.

The parameters that are indicated by 4 are initialized with the message identification and the message file, which is associated with this message identification. You can use a system message, or you can create your own message file to signal a message through trigger programs.

If you create your own message file, ensure that your messages are created by setting DMPLST = *NONE. Otherwise, your applications generate dump spooled files when they receive these messages.

The parameter that is indicated by 5 identifies the program message queue entry. If you use the value * (asterisk), you address the message queue of the procedure that is executing. We specified “1” for the program stack counter that is 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 might need to call your HLL trigger from a CL program. For example, you need to perform an OVRDBF SHARE(*YES) as shown in Figure 8-18 before the actual trigger logic is executed. The CL code is shown in Figure 8-18.

```
            PGM        PARM(&BUF &BUFSIZE)

                DCL        VAR(&BUF) TYPE(*CHAR)
                DCL        VAR(&BUFSIZE) TYPE(*CHAR) LEN(2)
                OVRDBF FILE(.....) SHARE(*YES)
                CALL PGM(ORDENTLIB/T4249RADT) PARM(&BUF  &BUFSIZE)

            ENDPGM
```

Figure 8-18 CL trigger example

After the trigger is activated, the job call stack is displayed as shown in Figure 8-19.

```
                            Job Call Stack

      Request  Program or
Opt    Level   Procedure        Library     Statement        Instruction
                QCMD             QSYS                            0351
                QUICMENU         QSYS                            00C1
          1     QUIMNDRV         QSYS                            0455
          2     QUIMGFLW         QSYS                            0483
          3     QUICMD           QSYS                            03E4
                QUOCPP           QPDA                            0541
                QUOMAIN          QPDA                            0FDD
          4     QUOCMD           QSYS                            0176
         --->   T4249CINS        ORDENTLIB   136                 00D9
                QSQROUTE         QSYS                            02F0
                QSQINS           QSYS                            01C0
         ====>> QDBPUT           QSYS                            0193
         --->   CLPGM            ORDENTLIB   600                 000B
         --->   T4249RADT        ORDENTLIB   .GET                0220
```

Figure 8-19 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, which is indicated by **5** in Figure 8-17 on page 238, 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 8-20).

```
                              Job Call Stack
     Request  Program or
Opt   Level   Procedure        Library     Statement        Instruction
              QCMD             QSYS                            0351
              QUICMENU         QSYS                            00C1
         1    QUIMNDRV         QSYS                            0455
         2    QUIMGFLW         QSYS                            0483
         3    QUICMD           QSYS                            03E4
              QUOCPP           QPDA                            0541
              QUOMAIN          QPDA                            0FDD
         4    QUOCMD           QSYS                            0176
    ---->     T4249CINS        ORDENTLIB   136                 00D9
              QSQROUTE         QSYS                            02F0
              QSQINS           QSYS                            01C0
    ===>>     QDBPUT           QSYS                            0193
              _QRNP_PEP_  ...  ORDENTLIB
    ---->     T4249IADT        ORDENTLIB   0000000048
```

Figure 8-20 Invocation stack with an ILE trigger

In Figure 8-20, you can see the call stack entry after an ILE RPG trigger is activated. The call stack entry that is 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_Program name
```

In our example, the full name is _QRNP_PEP_T4249IADT. Also, see the example in “Audit trail trigger in ILE RPG - T4249IADT” on page 255. If you use ILE C, the PEP name is always _C_pep, and it is case-sensitive. (See “Audit trail trigger in ILE C - T4249CCAT” on page 286.)

8.3.1 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 that are accessed by the trigger programs. To avoid data integrity potential exposures, triggers and applications need to share the same commitment definition. In this case, all of the changes that are performed by triggers are committed or rolled back by the application itself. The safest way to ensure that this commitment or rollback happens is by compiling your triggers with ACTGRP(*CALLER). Triggers and applications must also share the same lock level. For more information about handling the commit lock level in trigger programs, see 8.2, “Trigger program structure” on page 226.

If triggers run in a separate commitment control definition, they must commit or roll back their changes, because the application cannot commit or roll back their changes. Potential record-locking and consistency exposures can occur in this situation. If the trigger terminates normally without committing its changes, the application cannot release the locks on those records. Use different commitment definitions for triggers and applications only if they are strictly necessary, which we show in our example. (See “Audit trail trigger in ILE RPG - T4249IADT” on page 255.)

We now describe 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 that are made by the trigger itself.

Several scenarios are possible, 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 causes the automatic rollback of all changes that were made by the trigger program. The originating change operation is also rolled back.

  All of the changes that relate 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 that was previously made by the application is not affected. Consider the scenario in Figure 8-21.

  The application that is shown in Figure 8-21 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, which is indicated by the arrow. DB2 Universal Database for iSeries automatically rolls back all of the changes that are enclosed by the dotted line, but it does not affect the other operations that are made by the application. The application determines whether to commit those changes.

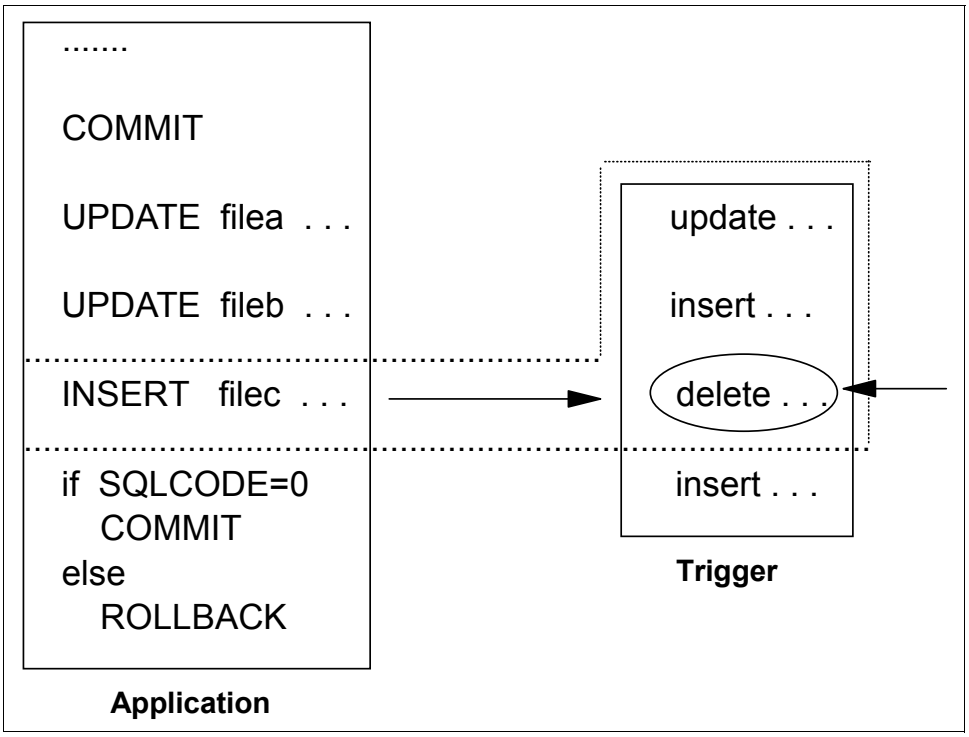


Figure 8-21 Atomic transactions with triggers

- Only the application uses commitment control

  In this case, all of the changes that are performed by the trigger are not rolled back, but the change operation that fired the trigger is rejected. Data integrity might 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 of the changes that were made by the trigger are automatically rolled back. If you handle exceptions inside the trigger, this activation group terminates normally and you need to explicitly issue a rollback operation to bring all of the changes back. The originating change is canceled only in the case of a BEFORE trigger.

  When an OPM trigger fails, the changes that are made by the trigger are not automatically rolled back. An explicit rollback must always be issued by the trigger to avoid lock exposure. The originating change is canceled only in the case of a BEFORE trigger.

- The trigger and application do not use commitment control

  In this case, the changes that are made by the trigger are not rolled back. The originating change is canceled in a BEFORE trigger.

Important: If your triggers modify database data, we suggest that you use commitment control in both applications and triggers because this method is the safest way to ensure the integrity of your data.

Table 8-3 summarizes the behavior of DB2 for i in an unmonitored trigger failure.

Table 8-3 Trigger and commitment control definition

Application program	Trigger program	Behavior
COMMIT=YES	COMMIT=YES	The originating change that is performed by the application and the changes that are made by the trigger are rolled back together.
COMMIT=YES	COMMIT=NO	The change that activated the trigger is rolled back. The changes that are made by the trigger are not rolled back.
COMMIT=NO	COMMIT=YES	After an unhandled exception, the changes that are made by the trigger are rolled back automatically if the activation group ends. For OPM triggers, an explicit rollback operation must be issued. The originating change is rolled back only in the case of a BEFORE trigger.
COMMIT=NO	COMMIT=YES	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.

8.4 Designing trigger programs

This section describes specific implementations of trigger programs that fit our Order Entry application scenario. We show the main advantages of using triggers instead of coding all of the functions as part of a specific application.

8.4.1 Order Entry application scenario

The main flow of the application is described in Chapter 2, “Stored procedures, triggers, and user-defined functions for an Order Entry application” on page 9. We focus our attention on particular functions that can be implemented by trigger programs. Before we describe in detail each of these trigger programs, we briefly overview the functions that were developed and a list of the samples that are included in this book. We included three trigger programs in our scenario:

- Verify salesperson/customer association

  This trigger program 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 251)
  - T4249IADT: Native ILE RPG (“Audit trail trigger in ILE RPG - T4249IADT” on page 255)
  - T4249CCAT: Native ILE C (“Audit trail trigger in ILE C - T4249CCAT” on page 286)

  This example is described in detail in 8.4.2, “Audit trail trigger example programs” on page 244. 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

  This trigger program performs several authority checks on updates of the Order Header and prints the invoice when the grand total is updated. We list the examples in the different languages:

  - T4249CINV: Native COBOL (“Update the trigger on Order Header - T4249CINV” on page 264)
  - T4249IINV: Native ILE RPG (“Invoice trigger in ILE RPG - T4249IINV” on page 271)
  - T4249CCIV: ILE C (“Invoice trigger in ILE C - T4249CCIV” on page 276)

  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 261.)

- Check customer credit limit

  This trigger program prevents the issuance of an order if the customer credit limit is exceeded and sends a fax to customers who reached 90% of their credit limit. If the customer is a special customer and they reached 90% of their credit limit, their credit limit is increased by 30%. This example of a trigger program changes the record that activated the trigger:

  - T4249CCTA: ILE RPG (“ILE RPG trigger program to send a fax - T4249CTA” on page 282)
  - T4249CCTA1: ILE C (“Changing the trigger buffer example” on page 290)
  - T4249CCTA2: ILE C (“Calling the trigger program recursively” on page 292)

8.4.2 Audit trail trigger example programs

Whenever a new order is created, ensure that the sales representative who takes the order is authorized to work with that customer. In our sales department, each team of sales representatives has its own pool of assigned customers. 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 8-22).

```
           ORDER  ENTRY



  ORDER  NUMBER:      00001
  CUST.  NUMBER:      00003
  ORDER  DATE:        2001-05-30
  DELIVERY  DATE:     2001-05-30




ENTER = ACCEPT        F3=EXIT
```

Figure 8-22 Order Entry display

For 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 due to a unique constraint on the corresponding database field. It is useful to also report the data description specifications (DDS) layout of this display file (Figure 8-23).

```
** This is the Order Entry Display File
**
** This covers the take call order header entry
**
                                                      REF(ORDENTREF)
                                                      INDARA
                R ORDER                               TEXT('ORDER ENTRY')
                                                      ASSUME
                                                      OVERLAY
                                                      CA03(15 'END OF PROGRAM')
                                                    2 35'ORDER ENTRY '
                                                      DSPATR(BL)
                                                    5   3'ORDER NUMBER: '
                   ORHNBR      R                B   5  20CHECK(ME)
    99                                                ERRMSG('ORDER ALREADY +
                                                      EXISTS' 99)
                                                      DSPATR(HI)
                                                    6   3'CUST. NUMBER: '
                   CUSNBR      R                B   6  20CHECK(ME)
    98                                                ERRMSG('CUSTOMER NOT FOUND'+
                                                      98)
    97                                                ERRMSG('ERROR - SEE +
                                                      JOBLOG')
                                                       DSPATR(HI)
                                                    7   3'ORDER DATE: '
                                                    8   3'DELIVERY DATE: '
                   ORHDTE             10         B   7  20
                                                         DSPATR(HI)
                   ORHDLY             10         B   8  20
                                                         DSPATR(HI)
                R EXITLINE                              TEXT('EXIT LINE')
                                                   23   3'ENTER = ACCEPT’
             F3 = EXIT'
```

Figure 8-23 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.

In particular, the Order Header input program must check the Customer/Salesrep relationship file to ensure that the current user ID that identifies the sales representative is allowed to place an order for that customer. The pseudocode of this traditional version of the Insert Order Header program is shown in Figure 8-24.

```
 Display input map;
         Read map;
         Initialize grand total to 0;
1      if CUSTOMER_NUMBER not in CUSTOMER file
             send error message;
         else
             retrieve USER PROFILE;
             if USER_PROFILE is not associated to CUSTOMER_NUMBER
2             send error message
                log violation attempt
             else
                insert ORDERHDR
                if duplicate_key
                   send error message
                else
                   return;
```

Figure 8-24 Traditional version of Insert Order Header

In Figure 8-24, 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 in Figure 8-24) to the system, implementing referential integrity between the CUSTOMER file and the Order Header (ORDERHDR) file.

We also need to track any violation attempt against our sales organization policy. This check is performed in the line marked by 2 in Figure 8-24 by the traditional application, which might lead to security exposures because this business rule can be circumvented by using a different data input interface, such as Interactive SQL, Data File Utility (DFU), or a different application. Also, if we need to change the rules, we probably need to modify all of the existing applications that work 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 8-25.

```
         Display input map;
         Read map;
         Initialize grand total to 0;
3      Insert record in ORDERHDR;
         if I/O error
            show error;
         else
            return;
```

Figure 8-25 Advanced version of Insert Order Header

In Figure 8-25, all of the data verification is performed by the database that is shown by 3 when the insert takes place. The resulting application is more compact and simpler. We need to check the return code only after the insert operation completes. All of the logic for business rules enforcement is moved outside of the application, and the logic 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.

Order Header entry program - T4249CINS

Example 8-2 shows the COBOL SQL implementation of this procedure.

Example 8-2 Order Header entry program - T4249CINS

```
       PROCESS  OPTIONS.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. T4249CINS.
         AUTHOR. PROGRAMMER NAME.
         INSTALLATION. ITSC LABORATORY.
         DATE-WRITTEN. APRIL 2001.
         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 "JOBI0400".
          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.
 01 INSERTOK             PIC 9.


     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 INITIAZ-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 ORDNUM   TO ORDNBR.
     IF IN15 IN ORDER-I-INDIC NOT EQUAL IND-ON THEN
*
*      The program inserts an order in ORDERHDR file.
*
          EXEC SQL
     INSERT INTO ORDENTL/ORDERHDR VALUES(:ORDNUM, :CUSTOMER,
     :ODATE, :ODLY, :OTOTAL, :USERNAME) :rk.4:erk.
          END-EXEC
           IF SQLCODE EQUAL 0 THEN
              MOVE 1 TO INSERTOK                                    4
           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 signaled 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 T4249OHRD RECORD INDICATORS ARE ORDER-I-INDIC.
```

The trigger program that is fired by this application function checks that the sales representative was assigned to that customer. If this condition is not satisfied, the trigger logs an entry in an audit file.

The pseudocode of the trigger is shown in Figure 8-26.

```
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;
```

Figure 8-26 Pseudocode for the audit trail trigger

The commit operation, which is highlighted in bold in Figure 8-26, needs description. We want to ensure that, whenever a violation occurs, the trigger logs the attempt. For this reason, the trigger must run in a separate commitment definition, so that the commit operation, which is highlighted in bold, affects only the insert operation in the audit file. This way, the interface that originates 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 must commit or roll back all of the changes at the end of the logical transaction. (See 8.3.1, “Commitment control and triggers” on page 240.)

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 that are made by your triggers.

If you are implementing these kinds of triggers in the IBM Integrated Language Environment (ILE), 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 must start commitment control for the named activation group. See “Audit trail trigger example in ILE RPG” on page 255 and “Audit trail trigger example in ILE C” on page 260.

Audit trail trigger example in COBOL SQL (OPM)

This version of the trigger program must perform uncommitted changes. To create the program that is 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 user with a better understanding of what happened during the trigger execution. Generally, when a trigger fails, applications send the user a generic message that references the job log. The trigger program checks for the job type that is 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

Example 8-3 shows the audit trail trigger in COBOL SQL.

Example 8-3 Audit trail trigger in COBOL SQL - T4249CADT

```
       PROCESS  OPTIONS.
        IDENTIFICATION DIVISION.
        PROGRAM-ID. T4249CADT.
          AUTHOR. IBM.
          INSTALLATION. ITSO.
          DATE-WRITTEN. MAY 2001.
          DATE-COMPILED. MAY 2001.
        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 JOBNAME             PIC X(10).
           03 USERNAME            PIC X(10).
```

```
    03 JOBNUMBER           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 "JOBI0400".
    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     ".
    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.
       05 NEW-REC-NULL-LEN PIC 9(8) BINARY.
       05 FILLER           PIC X(16).
    03 RECORD-JUNK.
       05 OLD-RECORD       PIC X(46).
       05 OLD-NULL-MAP     PIC X(6).
       05 NEW-RECORD       PIC X(46).
       05 NEW-NULL-MAP     PIC X(6).

 01 PARM-2.
    03 TRGBUF-LEN          PIC X(2).
*********************************************************
 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      *
* signaling 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 "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 "QMHSNDPM API ERROR" SND-MSG-ID.
        GOBACK.
```

Audit trail trigger example in ILE RPG

The following example of the audit trail trigger was implemented by using the ILE RPG language. As mentioned in the general description (8.4.2, “Audit trail trigger example programs” on page 244), 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 is opened. Use the User Open (USROPN) option in the F specs, and open the file explicitly in your program.
4. Create the corresponding RPG module.
5. Create a program that specifies a named activation group.

The following commands are needed to create this program:

```
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 8-27 shows the CL code that is needed to start commitment control in a separate activation group.

```
PGM
MONMSG MSGID(CPF0000)
STRCMCTL LCKLVL(*CHG) CMTSCOPE(*ACTGRP)
ENDPGM
```

Figure 8-27 Starting a separate commitment definition

The RPG program is shown in the following section.

Audit trail trigger in ILE RPG - T4249IADT

Example 8-4 shows the audit trail trigger in the ILE RPG program.

Example 8-4 Audit trail trigger in ILE RPG - T4249IADT

```
     FSALESCUS  IF   E           K DISK    INFDS(FILDS1)
     F                                     INFSR(*PSSR)
     F                                     RENAME(SALESCUS:SALECS)
     FAUDTFIL   O    E           K DISK    COMMIT
     F                                     USROPN
     F                                     INFDS(FILDS2)
     FDSPMSGD   CF   E             WORKSTN
     F                                     RENAME(DSPMSGD:DSPM)
      * Program status subroutines
     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
 *====================================================*
 * 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                                   *
 * 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                   *
 * NNOF  = 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  NNMAP                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  JOBTYP                61     61
 *================================================*
 * Outout 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
C                   PARM                    MSGLEN
C                   PARM                    MSGTYP         10
C                   PARM                    MSGQUE         19
C                   PARM                    PGMSTK
C                   PARM                    MSGKEY          4
C                   PARM                    MSGERR
C                   PARM                    MSGQLEN
C                   PARM                    CSEQUAL        20
C                   PARM                    PGMWTT
 *================================================*
 * Retrieve job attributes - USERID               *
 *================================================*
C     PLIST2        PLIST
C                   PARM                    RTVVAR         61
C                   PARM                    RTVLEN
C                   PARM                    RTVFMT          8
C                   PARM                    RTVNAM         26
C                   PARM                    RTVID          16
 *
C     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            RTVLEN
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
 *
C                   MOVE      RTVVAR        RTVAPI
 *================================================*
 * 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
 *=========================================================*
 * Signaling 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 signaling 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'     MSGTYP
C                   MOVEL(P)  MSGQNAM       MSGQUE
C                   MOVEL(P)  MODNAME       CSEQUAL
C                   MOVE      '  '          MSGDTA
C                   Z-ADD     1             PGMSTK
C                   Z-ADD     19            MSGQLEN
C                   MOVE      ' '           MSGKEY
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

You can use ILE C to start commitment control at the activation group scope by using the system ANSI C function. Any CL command that is executed by this function runs in the same activation group as the program that issues the system function.

For the audit trail example in ILE C, see “Softcoding the trigger buffer in ILE C” on page 285. In addition to the features that are covered by the COBOL and RPG programs, the ILE C example shows how to softcode the trigger buffer.

8.4.3 Updating a trigger on the Order Header file program examples

Whenever an update takes place on the ORDERHDR file, we want to ensure 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 that is issued for a customer from being rerouted to another one. See the description in 8.4.2, “Audit trail trigger example programs” on page 244, about the audit trail trigger.
- If the originator updates the grand total field, the order is complete. We need to generate the invoice in this case.

Enforcing all of 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 of 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 then rerouting it to a different customer later.

In our application scenario, this trigger plays a significant role when the order is submitted. Then, the procedure that is responsible for finalizing the order (FNLORD) 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

Example 8-5 shows the program to finalize the order.

Example 8-5 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
C*     returns to the calling program with the return code '1'.
C*
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                   EXFMTFNLORDR
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*
C*     The total order amount is added to the sales_rep's amount.
C*
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,
C*     and set return_code to '1'.
C*
C                    SQLCOD      IFEQ 100
C                    SQLSTT      ANDEQ'02000'
C                   SETON
C                   MOVE  '1'           RTNCD
```

```
C                   EXFMTFNLORDR
C                   RETRN
C                   END
C*
C*   The total order amount on ORDERHDR file is updated and
C*   this update will fire a trigger program.
C*   If the trigger fails, the update also fails and
C*   we rollback any record updated previously.
C*   The program returns an error code =   '1'. to the main.
C*
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
C*
C                                    MOVE '1'           RTNCD
C                   SETON
C                   EXFMTFNLORDR
C                   RETRN

C                                    END
C*
C                                    MOVE  '0'           RTNCD
C/EXEC  SQL
C+     COMMIT
C/END-EXEC
C*
C                                    RETRN
```

Invoice trigger example in native COBOL

This version of the trigger program generates a printed invoice. We use the printer file that is shown in Figure 8-28 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 8-28 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 MCH*xxxx* 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. The program 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 commands:

```
CRTCBLPGM  PGM(ORDENTLIB/T4249CINV)
           SRCFILE(*LIBL/QLBLSRC)
           SRCMBR(*PGM)
CRTCBLPGM  PGM(ORDENTLIB/T4249CHDL)
           SRCFILE(*LIBL/QLBLSRC)
           SRCMBR(*PGM)
```

Update the trigger on Order Header - T4249CINV

Example 8-6 shows the program to update the trigger on Order Header.

Example 8-6 Update the trigger on Order Header - T4249CINV

```
PROCESS   OPTIONS.
 IDENTIFICATION  DIVISION.
 PROGRAM-ID.  T4249CINV.
    AUTHOR. PROGRAMMER NAME.
    INSTALLATION. ITSC LABORATORY.
    DATE-WRITTEN. APRIL 2001.
    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 JOBNAME                      PIC X(10).
      03 USERNAME                   PIC X(10).
      03 JOBNUMBER                  PIC X(6).


*================================================*
* Parameter passed to the API QUSRJOBI to retrieve*
* the job attributes                              *
*================================================*


 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 "JOBI0400".
      03 RTV-JOB-NAME           PIC X(26) VALUE "*".
      03 RTV-JOB-ID              PIC X(16) VALUE " ".


*================================================*
* 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 " ".

*================================================*
* Signaling 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.
           05 NEW-REC-NULL-LEN PIC 9(8) BINARY.
           05 FILLER                  PIC X(16).
      03 RECORD-JUNK.
           05 OLD-RECORD           PIC X(46).
           05 OLD-NULL-MAP        PIC X(6).
           05 NEW-RECORD           PIC X(46).
           05 NEW-NULL-MAP        PIC X(6).

 01 PARM-2.
      03 TRGBUF-LEN               PIC X(2).
```

```
*********************************************************
 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.


       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 NORHDTE   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 ORDERNBR.
       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-WRITTEN. APRIL 2001.
    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 signaling trigger error           *
* This is an escape message sent to the calling  *
* program. The objective is signaling the       *
* 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-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 " ".

 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.

************************************************************
* Signaling 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 8-28 on page 263). We added one field, which is 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 must become part of the application transaction. The application must be able to commit or roll back all of the records that the trigger inserts in the invoice file.

To accomplish this task, we can use the dynamic commitment definition in the F specifications that is provided by ILE RPG. The keyword that 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 correct 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 commitment definition with 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:

```
CRTRPGMOD MODULE(ORDENTLIB/T4249IINV)
          SRCFILE(ORDENTLIB/QRPGILE)
          SRCMBR(T4249IINV)

CRTPGM PGM(ORDENTLIB/T4249IINV)
           ACTGRP(*CALLER)
```

Invoice trigger in ILE RPG - T4249IINV

Example 8-7 shows the invoice trigger in ILE RPG.

Example 8-7 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               *
 * NNOF  = 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  NNMAP                195    200
 *================================================*
 * Definition of the structure to be received by the  *
 * trigger program  - BUFFER LENGTH               *
 *================================================*
D PARM2           DS
D  LENG                   1      4B 0
 *
 *================================================*
 * 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  ORHDLY                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  PGMSTK                 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
 *
 *================================================*
 * Signaling exception inside trigger program       *
 *================================================*
```

```
C     PLIST1        PLIST
C                   PARM                    MSGID             7
C                   PARM                    MSGF             20
C                   PARM                    MSGDTA           25
C                   PARM                    MSGLEN
C                   PARM                    MSGTYP           10
C                   PARM                    MSGQUE           19
C                   PARM                    PGMSTK
C                   PARM                    MSGKEY            4
C                   PARM                    MSGERR
C                   PARM                    MSGQLEN
C                   PARM                    CSEQUAL          20
C                   PARM                    PGMWTT
 *
C     KEYS          KLIST
C                   KFLD                    ORHNBR
 *================================================*
 * Retrieve job attributes - QUSRJOBI API - USERID   *
 *================================================*
C     PLIST2        PLIST
C                   PARM                    RTVVAR           50
C                   PARM                    RTVLEN
C                   PARM                    RTVFMT            8
C                   PARM                    RTVNAM           26
C                   PARM                    RTVID            16
C     CMTLCK        IFNE      '0'
C                   MOVEL     '1'           VAR
c     '*YES'        dsply
C                   ELSE
C                   MOVEL     *BLANKS       VAR
c     '*NO'         dsply
C                   END
 *================================================*
 * Initialize MSGID in case the PSSR is called.        *
 *================================================*
C                   MOVE      'TRG0005'     MSGID
 *
 *================================================*
 * Move new record image received from the input     *
 * parameter into the work area                      *
 *================================================*
 *
C                   MOVEL     RECORD        NORDER
 *
 *================================================*
 * Move old record image received from the input     *
 * parameter into the work area                      *
 *================================================*
 *
C                   MOVEL     OREC          OORDER
 *
 *================================================*
 * Get job attributes - USERID                       *
 *================================================*
C                   Z-ADD     50            RTVLEN
C                   MOVEL     ' '           RTVVAR
C                   MOVE      'JOBI0400'    RTVFMT
C                   MOVEL(P)  '*'           RTVNAM
C                   MOVE      ' '           RTVID
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                                 9899
C     *IN98         IFEQ      *ON
C                   EXSR      *PSSR
C                   GOTO      EOFIN
C                   END
C     *IN99         IFEQ      *OFF
C                   WRITE     DETAIL                                 81
C     *IN81         IFEQ      *ON
C                   EXSR      *PSSR
C                   GOTO      EOFIN
C                   END
C                   END
C                   ENDDO
C                   END
C     EOFIN         TAG
C                   close     invoice
C                   RETURN
 *============================================================*
 * Trigger signaling 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'     MSGTYP
C                   MOVEL(P)  MSGQNAM       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 by using this language. For a description of commitment control in this trigger, see “Invoice trigger example in ILE RPG” on page 270. You can use the _Ropen function to specify whether the file must 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

Example 8-8 shows the same trigger program in ILE C.

Example 8-8 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 {               /*.... Trigger Parameter List ....*/
             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] = "JOBI0100";
 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);
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);


 strncpy(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 Excpt[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.Excpt);
     exit(1);
     }
 }
```

```
void ExcptHandler(int sig)
{
 char MsgId[7];
 memcpy(MsgId, "TRG0005", 7);
 sendMessage(MsgId); /*... Send generic trigger error message ....*/
 exit(0);
}
```

8.4.4 Softcoding the trigger buffer example

The trigger program example that is shown in this section illustrates two important concepts:

- Softcoding the trigger buffer
- Changing the trigger buffer

In all of the preceding examples, the structure of the table for which the trigger program is written is hardcoded inside the trigger program. Therefore, if you change the structure of the physical file, you also need to change the structure of the physical file inside the trigger program. This change must be repeated for all of the trigger programs where the structure of the related table changed. The alternative is that, at the time of writing the trigger program, you do not hardcode the structure of the table, but you softcode it. That way, if and when you change the structure of a table, you need to recompile the trigger program only and not change the trigger program itself.

Softcoding 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 softcoded 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 the *SQL/400 Developer's Guide* by Paul Conte and Mike Cravitz.

Softcoding 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 approaching their monthly credit limit so that they can either control their orders or apply to increase their credit limit. The notification must be sent through a fax when the monthly total for a customer amounts to more than 90% of the credit limit. If the customer is a special customer, which is recognized because its customer number starts with 9, the trigger program automatically increases the credit limit by 30%. This increase involves changing the record that activated the trigger.

If you are dealing with an existing application, integrating this new function without using triggers involves several 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, you must also move the logic for the advanced technology integration on the client platform.

By using triggers, you can incorporate this new function with almost no change to existing applications. Ensure 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, which leads to an abnormal end of the application.

The example program that 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 that is performed by the program FNLORD, which is shown in “Finalize order program - T4249FNLO” on page 261. When the customer information is updated to track the monthly balance of this specific customer, the trigger checks the fax number and possibly sends the fax to the customer.

Note: We do not show, for simplicity, the program that formats the document and sends the fax to the customer. The program SENDFAX that is 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

Example 8-9 shows the trigger buffer data structure.

Example 8-9 Data structure with trigger buffer - T4249BUF

```
  ****************************************************
  *   TRIGGER BUFFER data structure
  *
  ****************************************************
D TgBufDs              DS                                                  1
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   TgBfrOfs                                      Like( TypeBin4 )
D   TgBfrLen                                      Like( TypeBin4 )
D   TgBfrNulOf                                    Like( TypeBin4 )
D   TgBfrNulLn                                    Like( TypeBin4 )
D   TgAftOfs                                      Like( TypeBin4 )
D   TgAftLen                                      Like( TypeBin4 )
D   TgAftNulOf                                    Like( TypeBin4 )
D   TgAftNulLn                                    Like( TypeBin4 )
D   TgBufChr                        1   32767A                            2
D   TgBufAry                          1A     Overlay(        TgBufChr  )
D                                               DIM ( %Size( TgBufChr ) )
D
D
D*   End of TbBufDs
```

Notes: The following notes refer to Example 8-9 on page 281:

1 The TgBufDs data structure defines the trigger buffer. It is coded in a separate member so that 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 that is allowed, which is 32767. At the same time, it is redefined as an array of bytes by using another subfield that is called TbBufAry.

ILE RPG trigger program to send a fax - T4249CTA

Example 8-10 shows the ILE RPG trigger program to send a fax.

Example 8-10 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                                      1
     *****************************************************
     *  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 )2
     D                                     Prefix( Bf )
     D                                     Based( TgBfrPtr )
     D AfCustomer    E DS                  ExtName( Customer )3
     D                                     Prefix( Af )
     D                                     Based( TgAftPtr )
     *****************************************************
     *****************************************************
     *  OUTPUT PARAMETERS FOR QMHSNDPM
     *
     *****************************************************
     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
D  PGMSTK                5      8B 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
 ************************************************
 *  PARAMETERS NEEDED TO SIGNAL AN EXCEPTION INSIDE
 *     TRIGGERS
 ************************************************
C     PLIST1        PLIST
C                   PARM                    MSGID             7
C                   PARM                    MSGF             20
C                   PARM                    MSGDTA           30
C                   PARM                    MSGLEN
C                   PARM                    MSGTYP           10
C                   PARM                    MSGQUE           10
C                   PARM                    PGMSTK
C                   PARM                    MSGKEY            4
C                   PARM                    ERROR
 ************************************************
 *  PARAMETER CUSTOMER NUMBER TO SEND A FAX
 *
 ************************************************
C     PLIST2        PLIST
C                   PARM                    CUSNBR            5
 ******************************************************
 *  THIS TRIGGER PROGRAM WILL:
 *     - RETURN IMMEDIATELY IF THE TOTAL AMOUNT IS NOT
 *       BEING UPDATED
 *     - IF THE TOTAL AMOUNT IS BEING INCREASED AND REACHED
 *       90% OF THE CREDIT LIMIT
 *          *  SEND A FAX TO THE CUSTOMER
 *          *  IF THE CUSTOMER NUMBER STARTS WITH A 9 IS BECAUSE
 *             IT IS A VERY IMPORTANT CUSTOMER SO ITS CREDIT LIMIT
 *             IS INCREASED BY 30%.  THIS REQUIERES TO CHANGE THE
 *             AFTER IMAGE BUFFER OF THE RECORD THAT FIRED THIS
 *             TRIGGER.
 *     - IF THE FAX NUMBER IS *BLANKS   , AN INFO MESSAGE
 *       IS SENT AND WILL BE FOUND IN THE JOB LOG.
 ******************************************************

 *   LETS EVALUATE THE VALUE OF THE POINTERS FOR THE BEFORE
 *     AND AFTER IMAGES                                    4
 ******************************************************
C                   Eval      TgBfrPtr = %Addr(TgBufAry(TgBfrOfs + 1))
C                   Eval      TgAftPtr = %Addr(TgBufAry(TgAftOfs + 1))
 ******************************************************
 *  IF THE NEW TOTAL IS EQUAL OR LESS THAN THE OLD ONE,
 *  GO BACK IMMEDIATELY
 ******************************************************
C     AfCusTot      IFLE      BfCusTot
```

```
C                   RETURN
C                   ENDIF
 ******************************************************
 *  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'
C     AfCusCrd      Mult      1.3           AfCusCrd               5
C                   ENDIF
 ************************************************************
 *  CHECK THAT THE CUSTOMER HAS A FAX NUMBER TO CALL TO
 *
 ************************************************************
C     AfCusFax      IFNE      *Blanks
C                   MOVE      AfCusNbr      CUSNBR
C                   CALL      'SENDFAX'     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      '  '          MSGDTA
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
```

Notes: The following notes refer to Example 8-10 on page 282:

1. The /COPY statement incorporates the trigger buffer data structure into the source member. This approach is a good programming technique because all of the trigger programs can use it.
2. This data structure 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 data structure 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 with 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 variable is set by getting the address (by using the %Addr function) of the corresponding byte in the array that was declared to contain the trigger buffer. This same approach is used 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 reference the correct fields in the trigger buffer.
5. In this statement, the program updates one of the fields of the after image record in the trigger buffer.

Softcoding the trigger buffer in ILE C

This section shows how to code trigger programs in ILE C that use the trigger buffer softcoding 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 work with the customer that is specified in the record that is 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 who is working 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 specific customer by querying the SALESCUS table.

To create the ILE C trigger program, submit the following commands:

```
CRTCMOD MODULE(ORDENTLIB/T4249CCAT)
        SRCFILE(ORDENTLIB/QCSRC)
CRTPGM PGM(ORDENTLIB/T4249CCAT)
           ACTGRP(AUDIT)
```

Audit trail trigger in ILE C - T4249CCAT

Example 8-11shows the audit trail trigger in ILE C.

Example 8-11 Audit trail trigger in ILE C - T4249CCAT

```
/*............................................................*/
/*. This is a BEFORE INSERT trigger program associated with the  .*/
/*. ORDERHDR file. The program checks that:                     .*/
/*.   - The originator 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>
#include <decimal.h>

#include  <string.h>
#include <trgbuf.h> /*..... Header file for trigger buffer .....*/       1
/*.....        definition


/*..... Externally described files .....*/


#pragma mapinc("SalesCus", "ORDAPPLIB/SALESCUS(*ALL)",\
                           "input key", "d z _P", " ", "sc")
#pragma mapinc("Orderhdr", "ORDAPPLIB/ORDERHDR(*ALL)",\  2
                           "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 *);


ord_ORDERHDR_i_t     *Order;        /*.... Record Image ....*/  5
sc_SALESCUS_key_t          SalesCustomerKey;

Qdb_Trigger_Buffer_t     *TrgBuffer;  6


static _RFILE               *SalesCus, *Audit;
```

```
_RIOFB_T                        *SalesCusFB;


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

TrgBuffer->New_Record_Offset);


 /*.... Retrieving the current USERID ....*/

 QUSRJOBI(JobInfo, JobInfoLen, JobFmt, JobName, JobId);  8
 memcpy(UserId, JobInfo+18, 10);
 if(SalesCus == NULL) /*.... First time ....*/
      {
      system("STRCMTCTL LCKLVL(*CHG) CMTSCOPE(*ACTGRP)"); 9
      SalesCus = _Ropen("ORDAPPLIB/SALESCUS", "rr arrseq=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,  10
                              (void*) &SalesCustomerKey,
                                   sizeof(SalesCustomerKey), __KEY_EQ);
 if (SalesCusFB->num_bytes == 0)
        {
        _Rwrite(Audit, (void *) &SalesCustomerKey,  11
                                                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;
 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 Example 8-11 on page 286:

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. Use this statement to declare the externally described files in your program. Therefore, you do not need to code the record structure in your program.
3. This API can retrieve the user ID for the current job into the program.
4. This API can be used to send a message to another program. In the current example, this API 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 work 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 that is passed to the program is present.
7. Set the pointer to the record structure to the address of the data that was passed to the program. This example shows 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 work with the customer that is specified in the record that is 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 work with the specified customer, call the QMHSNDPM() API to send a message to the database manager to make the current operation fail.

8.4.5 Changing the record that fired a trigger

In certain situations, it might 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`.

Two methods exist 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 must be a BEFORE trigger.
- Update the trigger record by performing another I/O operation by 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 alternative is good when the first method cannot be used. This method cannot be implemented in languages that do not support recursion.

The Allow repeated change parameter is used only for those programs that 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 update the database record.

If you use 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 update the trigger record by using an I/O statement. You cannot update a record before it is written. Therefore, the trigger must be an AFTER trigger.
- When you insert or update records into tables with attached recursive triggers, set the isolation level of your program to *NONE 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 differs from the activation group of the program that inserts 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 “Softcoding the trigger buffer in ILE RPG” on page 280. 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

Example 8-12 shows the code to change the trigger buffer example.

Example 8-12 Changing the trigger buffer example

```
/*  ****************************************************************
     . This is a BEFORE UPDATE trigger on the CUSTOMER table. If the  .
     . 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  <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 +  1
                                                        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;  2



/* ...... Check if the total sales amount exceeds 90% of credit ...... */
/* ...... limit ...... */
        if ( CheckCrd <= NewRec->CUSTOT ) 3
        {
             strncpy( CustomerNumber, NewRec->CUSNBR, 5 );


/* ...... Check if the customer number begins with a '9' ...... */
            if ( NewRec->CUSNBR[0] == '9' ) 4
            {
/* ...... Change the trigger buffer and increase the credit ...... */
/* ...... limit by 30% ....... */
                 NewRec->CUSCRD = NewRec->CUSCRD * 1.3 ; 5
                 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( "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 Example 8-12 on page 290:

1. Obtain the address of the after image of the trigger record. This image needs to be changed.
2. Determine 90% of the credit limit.
3. Determine whether the total sales amount for the customer exceeded 90% of the credit limit.
4. If 90% of the credit limit is exceeded, determine whether the customer in question is a special customer by checking whether the customer number begins with 9.
5. If the customer number begins with 9, increase the credit limit by 30%.

Calling the trigger program recursively

Example 8-13 shows the code to call the trigger program recursively.

Example 8-13 Calling the trigger program recursively

```
/*  ********************************************************************
     . This is a AFTER UPDATE trigger on the CUSTOMER table. If the    .
     . 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 trigger.
     . record by using an update statement and therefore calls itself  .
     . recusively. The NumTimes variable which is a static integer     .
     . keeps a record of the number of the times the trigger program   .
     . was called                                                       .
     ******************************************************************** */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <decimal.h>
#include <recio.h>
#include <trgbuf.h>


/* ..... Include the 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;
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 +  1
                                                    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; 2

/* ...... Check if total sales amount exceeds 90% of the credit ...... */
/* ...... limit ...... */
        if ( CheckCrd <= NewRec->CUSTOT ) 3
        {
             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 ) 4
                  {
                      NewCrd      = NewRec->CUSCRD * 1.3;
                      OldCrd      = OldRec->CUSCRD;
                      NewCredit = NewCrd;
/* ...... Update the trigger record ...... */
                      EXEC SQL  5
                      update
                      ordapplib/customer
                      set
                      CUSCRD = :NewCrd
                      where
                      CUSNBR = :CustomerNumber;
                      NumTimes++;  6
                  }
             }
             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 Example 8-13 on page 292:

1. Get the after image of the trigger record that is updated.
2. Determine 90% of the credit limit.
3. Determine whether the total sales amount for the customer exceeded 90% of the credit limit.
4. Determine whether the time the trigger program is executed for the first time. This condition is not true if the trigger program is recursively called a second time. Check it to ensure 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 that 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 invocation 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 calling the trigger recursively.
6. If this invocation is the first time that the trigger is called, incrementally increase the variable that records the number of times that the trigger program is called.

8.5 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 will call other programs, you might 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 must use a separate open data path (OPD) and a full open is required. See Figure 8-29.

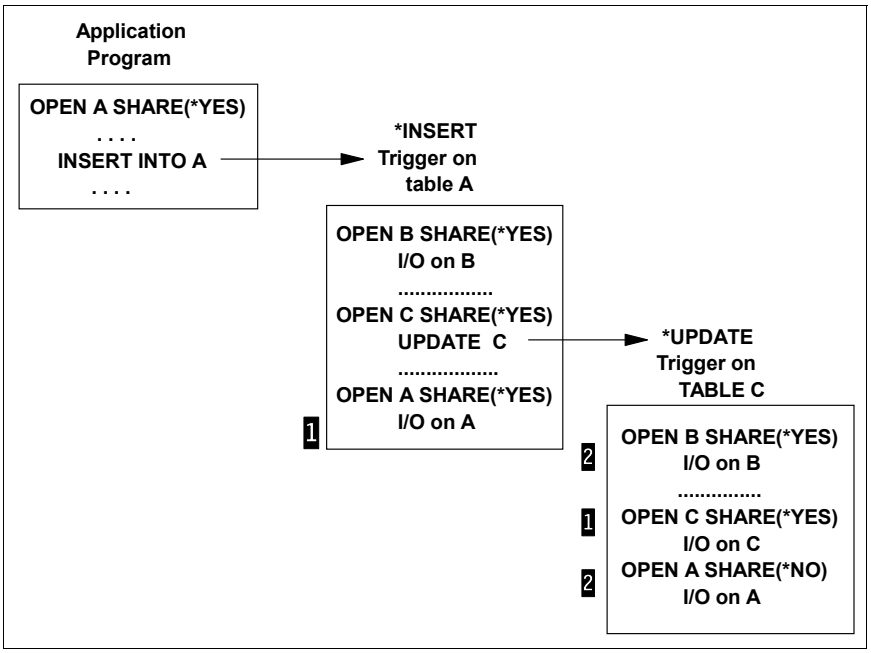


*Figure 8-29 SHARE(*YES) example that uses shared open in triggers*

Notes: The following notes refer to the numbers in Figure 8-29:

1 These operations fail. You are not allowed to share the same open data path (ODP) on the file that fired the trigger.

2 These operations succeed. The update trigger is associated with TABLE C. The update trigger can share the same ODP as the insert trigger on TABLE B. The update trigger opens TABLE A with the SHARE(*NO) option, which creates a separate ODP for that file.

- You cannot perform Distributed Relational Database Architecture (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 that is at a remote site from within a trigger, you can open distributed data management (DDM) files or start an Advanced Program-to-Program Communication (APPC) session with a remote partner program. When you access remote data in those ways, you can take advantage of the two-phase commit support that is offered by DB2 Universal Database for iSeries. If the trigger fails after a remote access was performed and an exception is sent back to the originating application, the entire transaction is put into a rollback-required state.

  In these cases, you must send back an escape message to the calling interface, either by the system or by the trigger, to ensure that all of 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 8-30.

  In Figure 8-30, 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 that are marked by an asterisk (*).

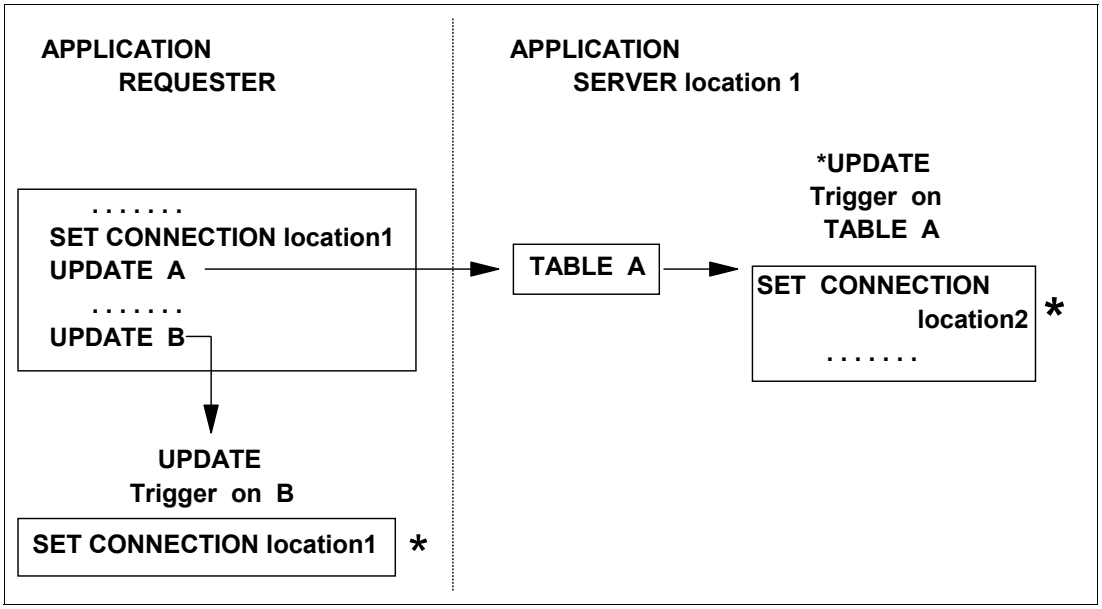


Figure 8-30 Changing the DRDA connection in triggers

If you plan to access data that is stored on a remote IBM i server from within a trigger program, you can use DDM files or start an APPC conversation. In Figure 8-31, you can see how a trigger can be activated remotely by updating a DDM file. The lower 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.

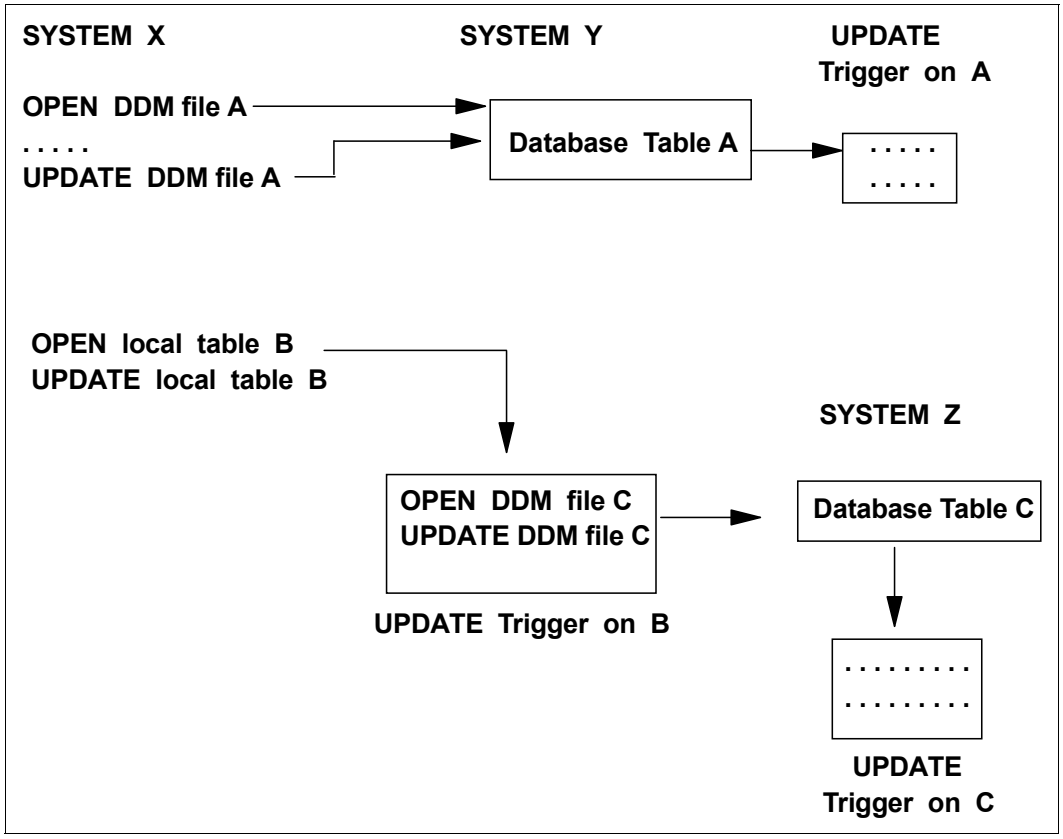


Figure 8-31 DDM file

If you need to access multiple locations in the same logical unit of work, the client application must control the connection switching that is shown in Figure 8-32.

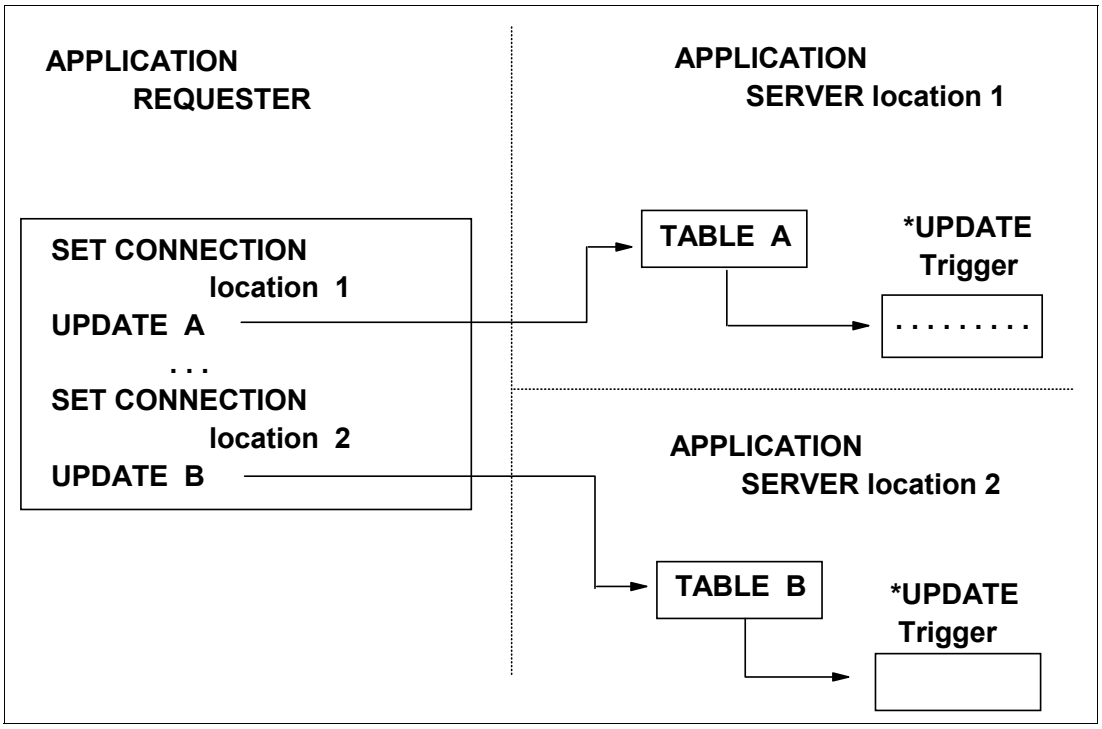


Figure 8-32 Triggers in a DRDA2 application

- 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 was 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: In certain cases, you might want to update the record that fired the trigger. For more information, see 8.4.5, “Changing the record that fired a trigger” on page 289.

  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 trigger.

  If both of your triggers and the applications run under commitment control, all of the rows that are modified by your trigger programs are locked and cannot be changed, not even through the same ODP that was used to perform the first data change.

For example, in Figure 8-33, the records that are marked with 1 are protected until the last trigger in the chain completes its execution. Therefore, the database I/O operations that are 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 that is marked with 4 will fail. You cannot use triggers to change the same row more than one time when the triggers are 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.

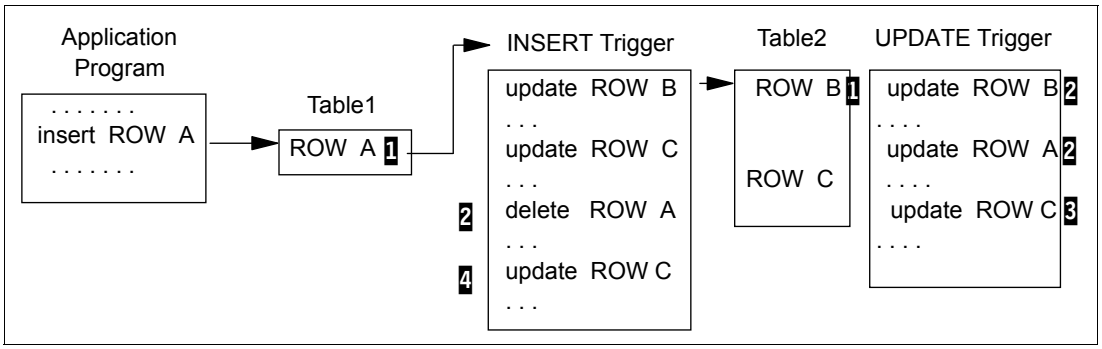


Figure 8-33 Destructive data change

Table 8-4 summarizes the different levels of protection, depending on the various commitment control scenarios.

Table 8-4 Various commit scenarios

	Application program	Trigger program	Behavior
1	COMMIT(*YES)	COMMIT(*YES)	All rows are protected.
2	COMMIT(*YES)	COMMIT(*NO)	Only the change operation is protected.
3	COMMIT(*NO)	COMMIT(*NO)	Only the change operation is protected.
4	COMMIT(*NO)	COMMIT(*YES)	Only the change operation is protected. Triggers can change the same record more than one time by using the same ODP.

- ILE procedures cannot be defined as triggers. Only objects of type *PGM can be added to a physical file. Therefore, 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). Nested triggers are limited to 200 to avoid potentially infinite loops. This limitation does not apply to recursive DELETE triggers because they delete all of the records in the file and no risk of an infinite loop exists.

  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 must be coded carefully to avoid 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.
- The system changes SEQONLY(*YES) to SEQONLY(*NO).

  If the physical file or the dependent logical file is opened for SEQONLY(*YES) and a trigger is associated with it, the system takes care of changing the open to SEQONLY(*NO) so that the trigger can be invoked for each record that changed.
- Triggers and object management

  Because 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 performed because no “hard” link exists 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 before V3R1. However, the trigger program must 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 of 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 the use of the **CPYLIB** command or the **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.

8.6 Recommendations

This section summarizes several of the recommendations that are presented in this chapter and includes more considerations about trigger development:

- Create the program with ACTGRP(*CALLER) if the program is running in an ILE environment to ensure 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 correct file definition to open the files with or without commitment control at run time, based on the application commitment definition.
- The trigger inherits the library list of the job that activated it. Do not forget to explicitly qualify the objects that are referenced inside the trigger with their libraries or to add those libraries to the library list because you might get a failure, depending on the application that activates the triggers.
- You might need your trigger programs to run asynchronously, for example, when you want to trigger a long-running process that must prevent 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. Because they might access objects to which the current user is not authorized, you might create them with the **USRPRF(*OWNER)** parameter. Alternatively, because triggers are 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 users from easily circumventing the rules by removing the triggers from the database files.

  Also, remove all of the authorities on the trigger program from the public because they are not necessary for the triggering mechanism. The system can always invoke the trigger, regardless of which user performs the data change.

- Performance considerations

  It is important to consider performance when you decide to implement triggers in your database design. Triggers are activated by using an external call. Try to evaluate carefully the trade-off of the performance impact over the benefit of the trigger functions.

  Consider these suggestions when you develop trigger programs:

  - Avoid compiling an ILE trigger with ACTGRP(*NEW).
  - Creating an activation group is expensive. Try to prevent it as much as you can.
  - If for a special reason your trigger runs in a separate activation group, remember to handle all of the exceptions. An unhandled exception will terminate the activation group, close all of the files, and cause an implicit rollback for the changes that were made by your trigger.
  - Minimize the number of file opens and closes.
  - Try to exit a trigger program in the “soft” way. Avoid, if you can, SETON LR in RPG, STOP RUN in COBOL, and exit() in C. Use this method to leave several 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 a 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.

9

Triggers, referential integrity, and constraints

You can use IBM DB2 for i 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 emphasis on the role that is played by commitment control in this scenario. The chapter starts with a description of transaction isolation and recovery before we describe the coexistence of triggers and referential integrity in detail.

This chapter describes these topics:

- Transaction isolation and recovery
- Trigger journal entries
- Triggers and referential integrity
- Comparing referential integrity and triggers
- Constraints and triggers: Ordering the actions
- Triggers, referential integrity, and commitment control
- Referential integrity, triggers, and journal entries

9.1 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 run. The user might put 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, the trigger will run with the isolation level that is 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 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)), the operations of a trigger are written to the database immediately and cannot be rolled back.

If both system triggers that are defined by the Add Physical File Trigger (**ADDPFTRG**) control language (CL) command and SQL triggers that are 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)), 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 that they perform under commitment control. SQL triggers, which are 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 that are 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, 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.

9.2 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 use journals other than the application journals or if they perform non-database activity (data queues, messages, data areas, and so on), you cannot find any evidence of the trigger activity in the application journal. Applying the journal entries on a restored version of a database file might not lead to the same result as the application execution produces.

In these cases, when you develop a trigger program, we recommend that you code the appropriate actions to facilitate a possible recovery process. For example, you can send user-defined journal entries to the application journals by using the QJOSJRNE application programming interface (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 trigger 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.

9.3 Triggers and referential integrity

This section describes these topics:

- 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 when you start to implement your applications.

9.4 Comparing referential integrity and triggers

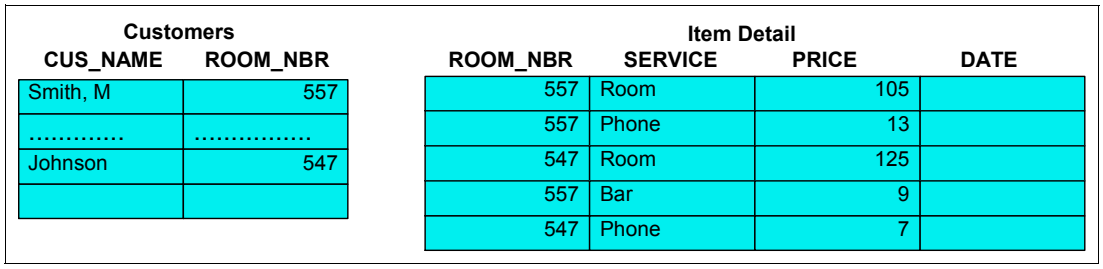
You can use DB2 Universal Database for iSeries to define both referential constraints and triggers on the same database file. This chapter explains the coexistence of triggers and referential integrity with particular regard to the role that is played by commitment control in this scenario. It also describes how the journal entries reflect the effects of triggers and referential integrity enforcement and the implications on the recovery process.

9.4.1 Using triggers to implement referential integrity rules

Many different relational database platforms implemented referential integrity by using 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 might not be guaranteed. DB2 Universal Database for iSeries implements *declarative* referential integrity.

Declarative referential integrity means that data consistency in a referential integrity network is verified also after a restore operation, after an application of the journal entries, or when a referential constraint is created in an existing database. The use of triggers to enforce referential constraint ensures data validation only at a single I/O operation, rather than at the global contents of the database. Consider this basic difference when business rules are enforced by using trigger programs.

A practical situation where you might 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 might use the database layout that is shown in Figure 9-1.



Customers

CUS_NAME	ROOM_NBR
Smith, M	557
.....	.....
Johnson	547

Item Detail

ROOM_NBR	SERVICE	PRICE	DATE
557	Room	105	
557	Phone	13	
547	Room	125	
557	Bar	9	
547	Phone	7	

Figure 9-1 An UPDATE CASCADE example

In Figure 9-1, the Customers file contains all of the customers that are currently present at the hotel and reports their room numbers. The Item Detail file contains all of the items that are 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 number. We need to propagate the update operation on the Customer file down to the Item Detail file, but no referential integrity rule 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 whether the room number changed and update all of 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 using 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 because this type of checking is done by the system at a low level.

9.5 Constraints and triggers: Ordering the actions

When constraints and triggers coexist in the same file, you need to remember how DB2 Universal Database for iSeries orders the various actions. The following description helps 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.

Because DB2 Universal Database for iSeries allows the coexistence of multiple constraints and triggers in the same file, we must analyze several possible combinations.

9.5.1 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 now.
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 now.
6. If commitment control is started, a check constraint is processed immediately after referential constraint enforcement.

You can use this method of sequencing the actions to write an INSERT trigger that, for example, removes or updates a previously existing key value to avoid a duplicate key exception.

9.5.2 Update operations

Updates are more complex to analyze because they influence data integrity either when they affect a parent or a dependent file of a referential integrity network. The detailed sequence is shown:

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 now.
4. The record is 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.

9.5.3 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. DB2 Universal Database for iSeries acts in this sequence when a delete operation is performed:

1. The *BEFORE trigger is activated.
2. If the file is a parent file of a *RESTRICT delete rule, this constraint is enforced now.
3. If commitment control is not started, a check constraint is processed now.
4. The record is deleted from the database file, and any non-constraint checking is performed now.

5. The *AFTER trigger is activated.
6. If the file is a parent file of a *CASCADE delete rule, this constraint is enforced now. If multiple *CASCADE delete constraints are defined, all of them are enforced now. Even if several of the files that are affected by the cascade process are parent files with a *RESTRICT delete rule, all of the matching records are deleted first. Then, the *RESTRICT delete rule is enforced. In Figure 9-2, you can see how this sequence influences the result of a delete operation in practice.

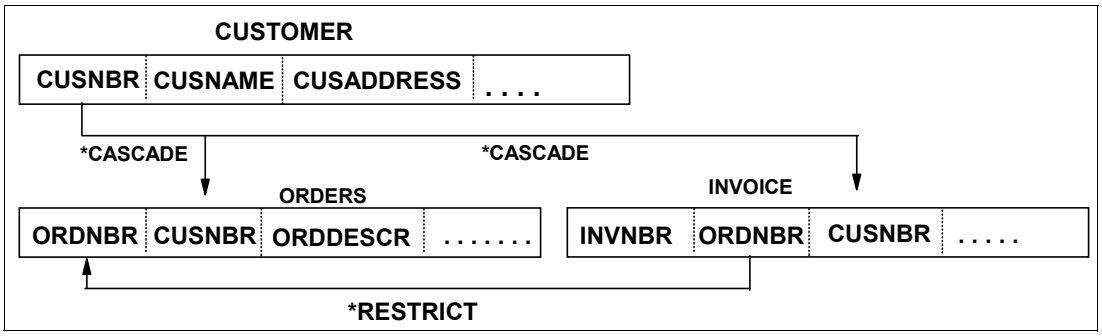


Figure 9-2 Rule ordering in a cascade network

The scenario that is shown in Figure 9-2 implements the following business rule: If a customer is deleted, all of the related orders and invoices must also be deleted. Alternatively, deleting an invoice that relates to an existing order must be prevented. If an application deleted a customer record, DB2 Universal Database for iSeries deletes all of the corresponding invoices and orders. When this process completes, the *RESTRICT rule in the invoice file is enforced, but by that time, all of the matching orders are already 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 now.
8. If the file is a parent file of a *NOACTION delete rule, this constraint is enforced now.
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. DB2 Universal Database for iSeries activates the *AFTER triggers before it enforces any referential integrity constraint with update or delete rules that differ from *RESTRICT. When you develop *AFTER triggers for your database, you might not always assume that the I/O operation that activated them already completed successfully when the trigger programs are in execution. The reason is because the operation might fail later as a result of a referential constraint violation.

An example of this situation is shown in Figure 9-3.

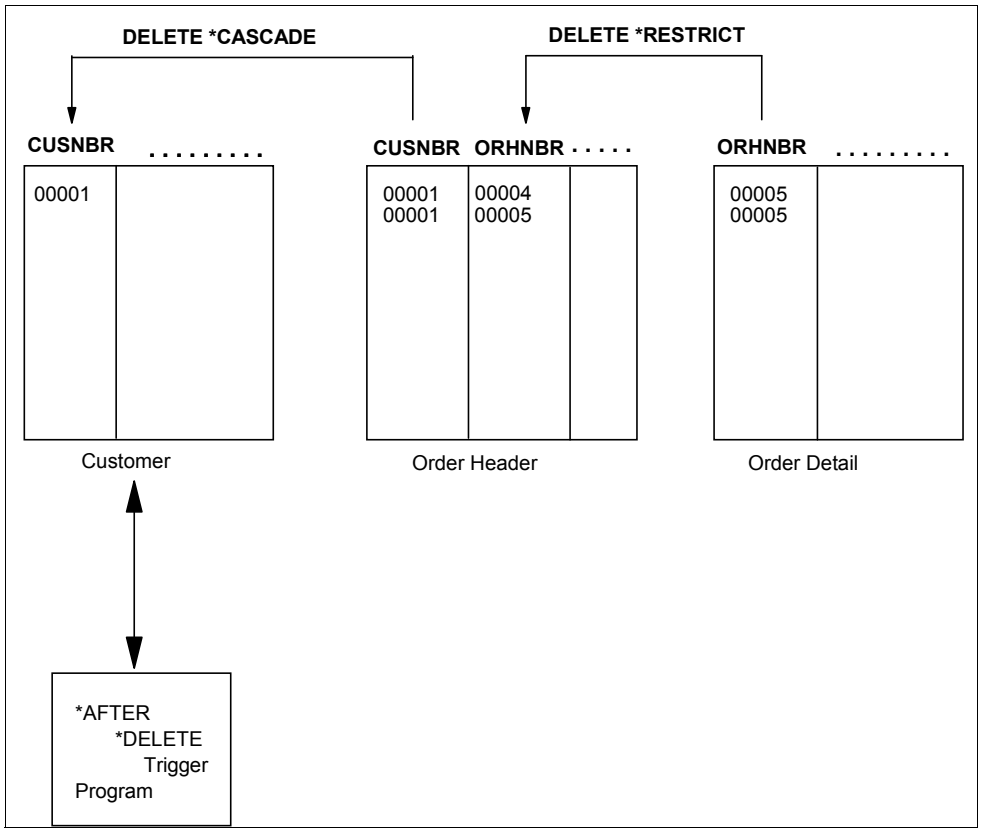


Figure 9-3 Delete trigger and delete rules

The example that is shown in Figure 9-3 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 no orders are outstanding or if all of the orders are “empty” (that is, with no detail rows). Empty orders might 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 actions occur in the following sequence:

1. The record is deleted from the Customer file.
2. The AFTER DELETE trigger is activated.
3. DB2 Universal Database for iSeries removes the related records in the Order Header file.
4. The *RESTRICT constraint is enforced now. If any related records are in the Order Detail file, the entire transaction is rolled back, and the original delete on the Customer file fails.

You might notice that the AFTER DELETE trigger must not assume that the original delete terminated successfully. Design this trigger carefully. Avoid, for example, any non-database activity because this kind of operation cannot be rolled back if the subsequent delete cascade fails.

9.6 Triggers, referential integrity, and commitment control

We described the implications of triggers and referential integrity on journaling and commitment control in 8.3.1, “Commitment control and triggers” on page 240. 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 that changes the data in the database is running commitment control. We address these two cases separately.

9.6.1 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 that is accessed. Therefore, if a trigger program is activated in this environment, the commitment control parameter that is passed to the program by DB2 Universal Database for iSeries in the trigger buffer (8.2, “Trigger program structure” on page 226) is set to a value different from 0, which indicates that commitment control started.

The trigger has no direct way to determine whether commitment control was started by the system or by the application. If the trigger opens a file with commitment control, according to the parameter that is received into the trigger buffer, either of the following two situations occurs:

- The file that is opened by the trigger program has no referential integrity constraints, or it has *RESTRICT rules only.

  In this case, the open fails because no user commitment definition was started. The system issues an exception (CPF4326). The trigger monitors the exception and reopens the file without commitment control. All of the changes that are made by the trigger are uncommitted. (That is, they are immediately permanent.)

- The file that is opened by the trigger program has referential integrity constraints with rules other than *RESTRICT.

  The open succeeds, and the changes fall into the system commitment definition. The changes that are made by the trigger are automatically rolled back if a failure occurs before the originating I/O completes.

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, which implies that a failure that is due to the enforcement of a referential integrity constraint does not roll back any changes that were previously made by the trigger program.

9.6.2 When the application runs under commitment control

In this scenario, the trigger program must use the SET TRANSACTION statement (SQL triggers) or open any database file with the commitment control option. (See 8.2, “Trigger program structure” on page 226 for a full description of this topic.) This way, you are guaranteed the atomicity of the whole transaction. All of the changes that result from the original database I/O are treated as a single transaction. Triggers must share the commitment definition with the application. Otherwise, their database changes are not considered a part of the atomic transaction.

Consider the example in Figure 9-3 on page 309. Assume that the AFTER DELETE trigger that is defined on the customer file performs database changes. In Figure 9-4, you can see the flow of the database operations that were caused by delete operation 1.

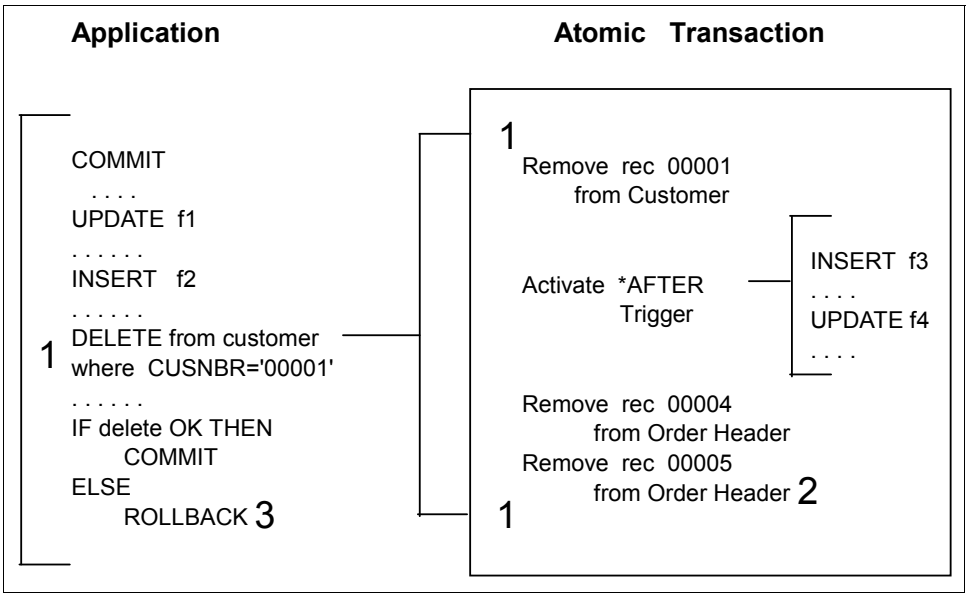


Figure 9-4 Atomic transaction with triggers and referential integrity

The application in Figure 9-4 runs under commitment control. The trigger program shares the commitment definition, which you can ensure by compiling the trigger with ACTGRP(*CALLER) if it is an Integrated Language Environment (ILE) program. The delete operation fails after the trigger terminated (2) and all of the changes that are included in box 1 are rolled back by the system. The application still can commit or roll back any other previous change operation (3) that is not affected by the implicit rollback.

Summarizing the previous description, 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 correct error message when you open files under commitment control.

9.7 Referential integrity, triggers, and journal entries

In 9.2, “Trigger journal entries” on page 305, we explained how DB2 Universal Database for iSeries logs additional information to identify that a journal entry was generated by a database change that resulted from a trigger action.

This additional information is relevant to those applications that implement a 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 required action, 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 database and the production database are exact copies of each other in terms of definitions, referential constraints, and triggers. Therefore, the journal entries that are 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 needs to perform only the same operations as the original application.

The amount of information that flows across the network can be greatly reduced by using this technique. Consider the example where the deletion of 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 that relates 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 of the journal entries that relate to the dependent files are also sent.

10

User-defined functions

This chapter describes the advantages of developing user-defined functions (UDFs) on IBM DB2 for i as the facility to create scalar or table functions that are similar to other system-supplied functions that are used in SQL statements.

This chapter describes the following topics:

- Introduction
- Nature of user-defined functions
- Type of user-defined functions
- Creating user-defined functions
- Resolving a UDF
- System catalog tables
- Authorization and adopted authority
- Transaction management considerations
- Coding considerations

10.1 Introduction

UDFs are host-language functions to perform customized, often-used tasks in applications. Programmers use UDFs to modularize a database application, creating a function that can be used in SQL.

DB2 for i comes with a rich set of built-in functions, but users and programmers might have requirements that are not covered by them. UDFs play an important role. Users and programmers can use UDFs to enrich the database manager by providing their own functions.

UDFs offer the following advantages:

- Customization

  Functions that are required by your application that do not exist 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 can choose a UDF to do the job.

- Flexibility

  You can use functions with the same name in the same library that accept different sets of parameters.

- Standardization

  Many of the programs that you implement use the same basic set of functions, but minor differences exist in all of the implementations. If you correctly implement your business logic as a UDF, you can reuse those UDFs in your other applications by using SQL.

- Object-relational support

  UDFs also provide additional functions for User-defined Distinct Types (UDTs) that are 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. A UDF is useful for performing calculations in the database manager server. Another area where performance might improve is working with large objects (LOBs). UDFs can be used to extract or modify portions of the information that is contained in a LOB directly in the database manager server instead of sending the complete LOB to the client side.

- Migration

  When you migrate from other database managers, certain built-in functions might not be defined in DB2 for i. UDFs allow us to create those functions to facilitate the migration process.

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 that are supplied with DB2 are a useful set of functions, but they might not satisfy all of your requirements. Therefore, you might need to extend SQL. For example, porting applications from other database platforms might require coding of certain platform-specific functions.

- Handle user-defined data types

  You can implement the behavior of a user-defined distinct type (UDT) by 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 two or more functions with the same name can be in the same library. For example, several instances of the SUBSTR function can accept different data types as input parameters. Function overloading is a key feature that is required by object-oriented development.

- Allow code reuse and sharing

  Business logic that is implemented as a UDF becomes part of the database, and it can be accessed by any interface or application by using SQL.

10.2 Nature of user-defined functions

A *function* is a relationship between a set of input values and a set of result values. When a function is invoked, a function performs an operation (for example, concatenate) based on the input and returns a single result or multiple results to the invoker. Depending on the nature of the return value or values, UDFs can be classified into one of two groups:

- User-defined scalar functions
- User-defined table functions

10.2.1 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 specific temperature in Fahrenheit is a scalar function. The statement in Example 10-1 uses two different scalar functions (UDTs and user-defined table functions (UDTFs)) that are combined in the same SELECT statement.

Example 10-1 Scalar UDTs and UDTFs that are combined in the same SELECT statement

```
SELECT
   DEC2DATE(ORDERDATE),
   DEC2DATE(SHIPDATE),
   WORKINGTIME(ORDERDATE,SHIPDATE),
   FROM ORDERS
```

The scalar functions in the example are DEC2DATE and WORKINGTIME.

DEC2DATE executes two times for each row that is processed by the SELECT statement.

10.2.2 User-defined table functions

User-defined table functions (UDTFs) are UDFs that can return 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. The following examples are UDTFs:

- A function that returns the names of sales representatives in a specified region
- A function that returns all employees whose annual compensation is higher than the average of the organizational unit to which they belong
- A function that returns the *k* most profitable customers is a table UDF

Note: One useful and important use of a table function is that it can access data in non-relational objects with SQL. A table function can be written to extract data out of a stream file in the integrated file system (IFS). Then, the invoking SQL statement can process that data in the same manner as data from a table that was created by SQL.

10.3 Type of user-defined functions

UDFs can be divided into three categories:

- Sourced UDFs
- SQL UDFs
- External UDFs

10.3.1 Sourced UDFs

Sourced UDFs are functions that are registered to the database that refer to another function. In fact, they map to the sourced function, which means that no coding is involved. Nothing more is required to implement a sourced UDF than to register it to the database by 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, such as +, -, *, /, or ||. This capability 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 that are defined as UDT MONEY, the function “+” can be defined as a function “+”(MONEY, MONEY) that returns MONEY, which is based on the standard “+”(DECIMAL, DECIMAL) that returns DECIMAL. See Example 10-2.

Example 10-2 Sourced function + for MONEY UDT

```
CREATE FUNCTION Library/“+”(MONEY, MONEY)
returns MONEY
specific plus00001
source QSYS2/“+”(decimal, decimal);
```

10.3.2 SQL UDFs

SQL UDFs are functions that are written entirely by using procedural SQL language. Their "code" consists of SQL statements that are embedded within the CREATE FUNCTION statement. SQL UDFs provide several advantages:

- They are written in SQL, which makes them portable to other database platforms.
- You define the interface between the database and the function by using SQL declares. You do not need to worry about the details of passing the parameters.
- You can pass large objects (LOBs), datalinks, and UDTs as parameters, and manipulate them in the function itself.

For example, consider the situation where many tables with columns DECIMAL(8) represent dates in format *YYYYMMDD*. Those dates must be converted to DATE. No built-in function performs the requested operation, but we can use an SQL statement, as shown in Example 10-3.

Example 10-3 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 converts the DECIMAL to DATE and changes the previous SELECT statement to the SELECT statement in Example 10-4.

Example 10-4 Select statement that uses a UDF for DECIMAL to DATE conversion

```
SELECT DEC2DATE(ORDER_DATE) AS CONVERTED_DATE FROM SOURCE_TABLE
```

The SQL UDF for this conversion is shown in Example 10-5.

Example 10-5 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 ;
```

Also, SQL UDFs are useful when you want to see the result of a query as a table, for example, a table of a group of employees in a particular project. The implementation of these SQL UDFs is described in *SQL Procedures, Triggers, and Functions on DB2 for i,* SG24-8326.

10.3.3 External UDFs

External UDFs are references to programs and service programs that are written in high-level languages (HLLs):

- C
- C++
- Integrated Language Environment (ILE) CL
- COBOL
- ILE COBOL
- FORTRAN
- PL/I
- RPG
- ILE RPG
- Java

After the function is registered to the database, the database invokes the program or service program whenever the function is referenced in a data manipulation language (DML) statement. As in SQL UDFs, external UDFs can return a scalar value or table.

Work with external UDFs for the following reasons:

- To perform non-database functions
- To access non-relational data
- To reuse existing code
- To use existing skills

For example, you can write an external function that checks whether a binary large object (BLOB) that is passed contains a picture in GIF format. See Example 10-6.

Example 10-6 Creation of an external function

```
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;
```

10.4 Creating user-defined functions

Before a UDF can be recognized and used by the database manager, it must be created by using the CREATE FUNCTION statement. Use this statement to specify the name and language of the function. Also, you can use this statement to specify certain behavioral characteristics, such as whether the function is deterministic, can be used in parallel, or reads or modifies SQL data.

You can use the DROP FUNCTION statement to delete the function in the catalog information entry. For more information about the CREATE FUNCTION, see the *SQL Reference for Cross-Platform Development*, which is available at this website:

ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/cpsqlrv11.pdf

10.4.1 CREATE FUNCTION

Use the CREATE FUNCTION statement 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 how the UDF is identified in DB2 for i. This section explains several ways. For a complete description of the CREATE FUNCTION command, see the *SQL Reference for Cross-Platform Development*, which is available at this website:

ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/cpsqlrv11.pdf

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 coded character set identifier (CSID) 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 specified schema 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 no current library exists, 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 Parameter-Declaration is not required, you can give each parameter a name.

The maximum number of parameters that are allowed in CREATE FUNCTION is 90. For external functions that are created with PARAMETER STYLE SQL, the following information is included:

- Specified input and result parameters
- Implicit parameters for indicators, SQLSTATE, function name, specific name, and message text
- Any optional parameters

The maximum number of parameters is also limited by the maximum number of parameters that are allowed by the licensed program that is used to compile the external program.

Tip: For the portability of functions across other DB2 Universal Database platforms, do not use the following data types, which might represent different information on other platforms:

- FLOAT: Use DOUBLE or REAL instead.
- NUMERIC: Use DECIMAL instead.

RETURNS

RETURNS specifies the output of the function.

SPECIFIC specific-name

SPECIFIC *specific-name* defines a unique name for the function.

When you define multiple functions with the same name and schema but different parameters (10.5, “Resolving a UDF” on page 325), we recommend that you define a specific name. The specific name can be used to uniquely identify the function. The specific name can be used when you source on this function, drop the function, or comment 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, the schema name 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.

Important tip: You can use the `SPECIFIC` keyword 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 that is associated with the external program at the time that the function is created. The language of the program is assumed to be C if the following characteristics are true:

- The program attribute information that is 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. Specify NOT DETERMINISTIC 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 when the Fahrenheit temperature is provided is deterministic. No matter under which circumstances the function is called, the function always returns the same result when the parameter values are equal.

A UDF that accesses a thermometer and returns the temperature is non-deterministic because it might 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 that is 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 might run in the same thread as the invoking SQL statement. NOT FENCED functions can keep SQL cursors open across individual calls to the function. Because cursors can be kept open, the cursor position is also preserved between calls to the function.

A UDF, when it is defined as FENCED, runs in the same job as the SQL statement that invoked it. However, the UDF runs in a system thread, which is separate from the thread that runs the SQL statement. By default, UDFs are created as FENCED. For complex UDFs, this separation is meaningful because it avoids potential problems, such as generating unique SQL cursor names. A UDF that is 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.

Tip: The use of UNFENCED versus FENCED UDFs provides better performance because the original query and the UDF can run within the same thread.

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 input argument is null. The result is the null value.
- CALLED ON NULL INPUT: Specifies that the function will be invoked, if any, or all, argument values are null, which makes 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 an external action (outside the scope of the function program). The function must be invoked with each successive function invocation. EXTERNAL ACTION must be specified if the function contains a reference to another function with an external action. An example of an external action is to insert a row to a table or put an entry on a data queue.
- NO EXTERNAL ACTION: The function does not perform an external action. It does not need to 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 16000000. 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 after 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 is allocated for the function.

The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, one scratchpad exists. For example, assume 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 approach 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 by the parallel task and not the SQL statement. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

SCRATCHPAD is only allowed with PARAMETER STYLE DB2SQL or PARAMETER STYLE DB2GENERAL.

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 that indicates 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 shown:
  - 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: A final call occurs when the cursor is closed for cursor-oriented statements, or the execution of the statement completed.
    - End of a parallel task: A final call occurs when the function is executed by parallel tasks.
    - End of transaction: A final call occurs when normal end of statement processing does not occur. For example, the logic of an application bypasses closing the cursor.

    Certain 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 one time for the function. Specify the DISALLOW PARALLEL clause for functions that have inappropriate actions when they are executed in parallel.

    If a commit operation occurs while a cursor that is 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.

    Committable operations must not be performed during a FINAL CALL because the FINAL CALL might occur during a close that is invoked as part of a COMMIT operation.

PARALLEL

The PARALLEL parameter indicates whether the function can run in a parallel implementation of the query (if the optimizer chooses to do so). It applies only when DB2 Symmetric Multiprocessing (SMP) is installed and activated. The same UDF program can run in multiple threads at the same time. Therefore, if ALLOW PARALLEL is specified for the UDF, ensure that it is threadsafe.

The default is DISALLOW PARALLEL, if you specify one or more of the following clauses:

- NOT DETERMINISTC
- 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 you create the function.

DBINFO

DBINFO specifies whether the function requires that the database information is passed:

- DBINFO

  DBINFO specifies that the database manager must pass a structure that contains status information to the function. Detailed information about the DBINFO structure is in the include file `SQLUDF` in QSYSINC.H.

  DBINFO is only allowed with PARAMETER STYLE DB2SQL or PARAMETER STYLE DB2GENERAL.

- NO DBINFO

  NO DBINFO specifies that the function does not require that the database information is passed.

PARAMETER STYLE

PARAMETER STYLE specifies the conventions that are used for passing parameters to and returning the values from functions.

10.4.2 Modifying a UDF

No ALTER statement exists for altering or modifying a UDF. When a change must be made to an existing UDF, it is necessary to drop and re-create the function.

10.4.3 Dropping a UDF

To drop a UDF by using the SQL interface, use the DROP FUNCTION statement. The DROP FUNCTION statement references the function by one of the following identifiers:

- *Name*: For example, DROP FUNCTION myUDF. Name is only valid if one function of that name exists in that schema. Otherwise, SQLSTATE `42854` ('`More than one found`') or SQLSTATE `42704` ('`Function not found`') is signaled.
- *Signature* (name and parameters): For example, DROP FUNCTION myUDF(int). The data type of the parameters must match the data type of the function. Also, if length, precision, or scale is specified, it must match the function to drop. SQLSTATE `42883` is signaled if a match to an existing function is not found.
- *Specific name*: For example, DROP SPECIFIC FUNCTION myFun0001. Because the SPECIFIC name must be unique for each schema, one function, at most, is found. If the function is not found, SQLSTATE `42704` ("`Function not found`") is signaled.

To drop a UDF by using System i Navigator, you open the required library, right-click the UDF that you want to delete, and select **Delete**. See Figure 10-1.

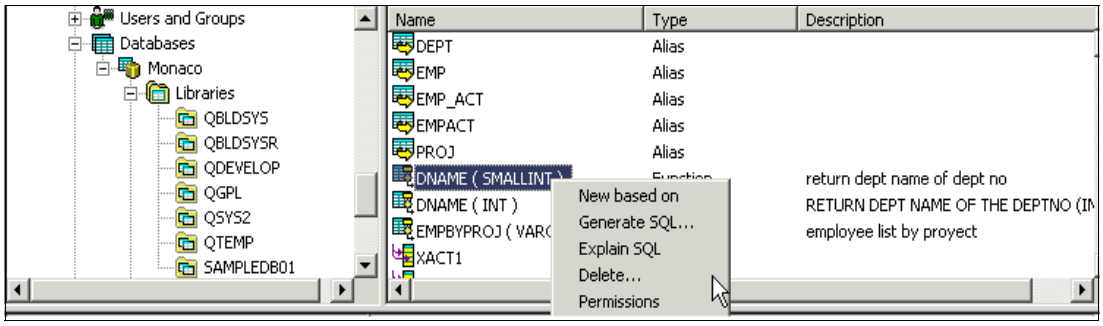


Figure 10-1 Dropping functions with System i Navigator

If no dependent functions exist, the right panel refreshes, and you can see that the UDF object was removed from the library.

10.5 Resolving a UDF

Resolving to the correct function to use for an operation is more complicated than other resolution operations because DB2 Universal Database supports function overloading. Function overloading means that a user can 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 can define its own SUBSTR function that takes slightly different parameters. Therefore, even resolving to a supposedly built-in function still requires that function resolution is performed. The following sections explain how DB2 for i resolves references to functions.

10.5.1 UDF overloading and function signature

DB2 for i supports the concept of function overloading. You can have two or more functions with the same name in the same schema, library, or collection, if they have different signatures. 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 is considered to be part of the signature. Therefore, if a function that is called DNAME in library `SAMPLEDB01` accepts an input parameter of type CHAR(10), you cannot have another function that is called DNAME in the same `SAMPLEDB01` library that accepts CHAR(12). However, it is possible to have another function DNAME in library `SAMPLEDB01` that accepts an INTEGER value as an 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))
```

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 that is returned by the function is *not* considered to be part of the function signature. Therefore, you cannot have two functions that are called DNAME in library SAMPLEDB01 that accept input parameters of the same data type, even if they return values of different data types.

10.5.2 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 with input parameters that exactly match those input parameters that are specified in the DML statement, that function is chosen for execution.

If the system cannot find any function in the path that exactly matches those parameters that 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 character large object (CLOB). Restrictions exist on the data type to which a particular parameter can be promoted. We explain this concept with an example.

Assume that you created a table CUSTOMER in library LIB1. This table has, among its other columns, a column that is named CUSTOMER_NUMBER, which is a CHAR(5). Also, assume that you wrote a function, GetRegion, that will process and return the region to which your customer belongs. The data type of the parameter that this function accepts as input is defined as type CLOB(50K). Assume that no other functions are called GetRegion in the path. If you execute the following query, you see that the function GetRegion(CLOB(50K)) is executed:

```
select GetRegion( customer_number ) from customer
```

How is this query successful? The field CUSTOMER_NUMBER from the CUSTOMER table has the data type CHAR(5). The function GetRegion accepts a CLOB as a parameter, and no other functions that are called GetRegion are in the path.

In its attempt to resolve the function call, the system first searched the library path for a UDF that was 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 a UDF that was 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 a UDF that was 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 demonstrated in the previous example. Table 10-1 indicates the data types and the data types to which they can be promoted.

Table 10-1 Precedence of data types

Data type	Data type precedence list (in best to worst order)
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB, or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB, or DBCLOB
CLOB or DBCLOB	CLOB or DBCLOB
BLOB	BLOB
SMALLINT	SMALLINT, INTEGER, DECIMAL or NUMERIC, REAL, or DOUBLE
INTEGER	INTEGER, DECIMAL or NUMERIC, REAL, or DOUBLE
DECIMAL or NUMERIC	DECIMAL or NUMERIC, REAL, or DOUBLE
REAL	REAL or DOUBLE
DOUBLE	DOUBLE
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
DATALINK	DATALINK
A user-defined type	The same user-defined type

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 10-1. Therefore, 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:

```
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, which are not compatible with a fixed-length character data type.

10.5.3 Function path and the function selection algorithm

On the IBM i system, two types of naming conventions are possible when you use 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 IBM i system, and the SQL naming convention is specified by the American National Standards Institute (ANSI) SQL standard.

The function resolution process depends on which naming convention you are using at the time that you execute the SQL statement, which refers to a UDF.

Function path

When *unqualified* references are made to a UDF inside an SQL statement, DB2 for i 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 and UDTs. If 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 set of libraries that you want by 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. 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 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 by using the outlined steps:

1. The algorithm finds all functions from the catalog (SYSFUNCS) and the built-in functions that match the name of the function. If a library was specified, the algorithm gets those functions from that library. Otherwise, the algorithm gets all functions whose library is in the function path.
2. The algorithm eliminates those functions whose number of defined parameters does not match the invocation.
3. The algorithm 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 specific 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 preceding steps, it must be the case (the way the algorithm works) that all of 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 no candidate functions exist, it signals the error SQLSTATE 42884.

10.6 System catalog tables

The database manager provides several data dictionary facilities to track UDFs. In this section, we show how to view UDF information by using the SYSROUTINES catalog, the SYSPARMS catalog, and the SYSFUNCS view.

10.6.1 SYSROUTINES catalog

UDF references 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:

ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/cpsqlrv11.pdf

Note: The SYSROUTINES catalog contains details for both UDFs and stored procedures. When you want to see only UDFs, you can use a view that is called SYSFUNCS, or you can select rows in the SYSROUTINES catalog where ROUTINE_TYPE is FUNCTION.

The sample SQL statement in Example 10-7 displays SYSROUTINES information about UDFs in our test SAMPLEDB01 library.

Example 10-7 Query of SYSROUTINES that shows UDFs that are defined on a specific schema

```
SELECT      *
FROM        QSYS2.SYSROUTINES
WHERE       ROUTINE_SCHEMA = 'SAMPLEDB01'
AND         ROUTINE_TYPE = 'FUNCTION';
```

Figure 10-2 shows that two UDFs, DNAME and EMPBYPROJ, are in this schema. The first is a scalar function. The second is a table function. Neither of them allows parallelism. EMPBYPROJ is fenced. DNAME is not fenced.

```
/* Enter one or more SQL statements separated by semicolons */
select SPECIFIC_SCHEMA,SPECIFIC_NAME, ROUTINE_NAME, ROUTINE_BODY, FUNCTION_TYPE,PARALLELIZABLE,FENCED from
qsys2.sysroutines where routine_schema = 'SAMPLEDB01' and routine_type = 'FUNCTION';
```

SPECIFIC_SCHEMA	SPECIFIC_NAME	ROUTINE_NAME	ROUTINE_BODY	FUNCTION_TYPE	PARALLELIZABLE	FENCED
SAMPLEDB01	DNAME	DNAME	SQL	S	NO	NO
SAMPLEDB01	EMPBYPROJ	EMPBYPROJ	SQL	T	NO	YES

Figure 10-2 Content of SYSROUTINES catalog

10.6.2 SYSPARMS catalog

The SYSPARMS catalog contains one row for each parameter of a UDF that was created by the CREATE FUNCTION statement.

Assume that you want to retrieve the parameter details for all instances of the DNAME function that is in the SAMPLEDB01 library. You can run the SQL statement in Example 10-8 to display this information.

Example 10-8 Sample query on SYSPARMS that shows the parameters for the 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');
```

Due to function overloading, the SAMPLEDB01 schema can contain functions with the same routine name. By running this query, we produced the results that are shown in Figure 10-3.

Two instances of the DNAME function are in the SAMPLEDB01 library. Their signatures differ because they accept an input parameter of type SMALLINT or INTEGER. Also, the result of a function is stored in the SYSPARMS catalog as an OUTPUT parameter.

```
/* Enter one or more SQL statements separated by semicolons */
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');
```

SPECIFIC_SCHEMA	SPECIFIC_NAME	ORDINAL_POSITION	PARAMETER_MODE	PARAMETER_NAME	DATA_TYPE
SAMPLEDB01	DNAME	1	IN	DEPTNO	SMALLINT
SAMPLEDB01	DNAME	2	OUT	-	CHARACTER
SAMPLEDB01	DNAME00001	1	IN	DEPTNO	INTEGER
SAMPLEDB01	DNAME00001	2	OUT	-	CHARACTER

Figure 10-3 UDF parameter details in SYSPARMS catalog

10.7 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 that corresponds to that UDF. It depends on how it was defined in the USRPRF attribute at the program or service program object creation time.

The authorization and adopted authorities behave as they do in a stored procedure, as explained in 3.7, “Authorization and adopted authority” on page 35.

10.8 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 must not affect the transactional behavior of their callers. Therefore, a UDF must not perform COMMIT, ROLLBACK, SAVEPOINT, or SET TRANSACTION operations.

10.9 Coding considerations

When you code UDFs, keep in mind the limitations and restrictions that apply to them. The following list contains important recommendations and hints for UDF developers:

- UDFs must 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) during the UDF execution.
- If a UDF does not finish in an allocated time, the SQL statement fails. You can override the system timeout value with the UDF_TIME_OUT parameter in the query option file QAQQINI. For more information, see *DB2 UDB for AS/400 SQL Programming*, SC41-5611.
- Avoid inserts, updates, and delete operations on the same tables as the table that is referred to in the invoking statement.
- A UDF runs in the same job as the invoking SQL statement, but it runs in a separate system thread, so secondary thread considerations apply:
  - UDFs will conflict with thread-level resources that are 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 the program-adopted authority that was active. Authority comes from the UDF program or the user that runs the SQL.

11

External user-defined functions

External user-defined functions (UDFs) are coded in one of the high-level languages (HLLs) that are available on the IBM i server. If you want complex sophisticated processing is required, or you plan to reuse existing code, external UDFs are the best choice.

This chapter describes external UDFs that are written in HLLs. It explains how to register and code external UDFs. It also reviews the differences in coding external UDFs as far as the different parameter styles that are supported by IBM DB2 for i. It describes how to invoke external UDFs and handle errors.

Also, all of the benefits of UDFs that are described in Chapter 10, “User-defined functions” on page 313 apply to external UDFs.

This chapter includes the following topics:

- User-defined function considerations
- Registering an external UDF
- Parameter styles in external UDFs
- Scratchpad in UDFs and UDTFs
- UDF and UDTF calling sequence
- Coding an external UDF
- Error handling in external UDFs
- Pointer arithmetic and the scratchpad
- Coding example for an external user-defined table function

11.1 User-defined function considerations

A UDF is written by a user in one of the programming languages on the IBM i system. External UDFs can be written in C, C++, RPG, COBOL, CL, and Java. SQL programmers can invoke business calculations or processes that are written on one of the listed languages from an SQL statement without needing to know how the UDF is implemented. You can create external UDFs that are based on programs or service programs. To create an external UDF, the HLL source code must be compiled and the program or service program object must be created.

When an external UDF that is associated with an Integrated Language Environment (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 system 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 Integrated Language Environment (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) must normally be used so that the UDF runs in the same activation group as the calling program.

To run Java functions, you need to install the Developer Kit for Java (5722-JV1) on your system. Otherwise, an SQLCODE of -443 is returned, and a CPDB521 message is placed in the job log.

11.2 Registering an external UDF

Before you use a UDF, you must register the UDF in the database by using the CREATE FUNCTION statement. When an external UDF is registered within the database, entries are made in the SYSROUTINES and SYSPARMS system catalog tables. The content of these system catalog tables is described in 3.4, "System catalog tables" on page 31.

11.2.1 Registering an external UDF with System i Navigator

The following steps show how to create an external scalar and table UDF by using System i Navigator.

Registering an external scalar UDF

In the following example, we register an external UDF, which is based in an RPG program, that converts a decimal number that represents a date in format *YYYYMMDD* to a DATE value, such as `2003-10-10`. To see the source code of this example, see 11.6.1, "Coding the SQL parameter style" on page 352.

Follow these steps:

1. Double-click the **System i Navigator** icon on the desktop. Expand **My Connections** and the IBM i 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 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 11-1.

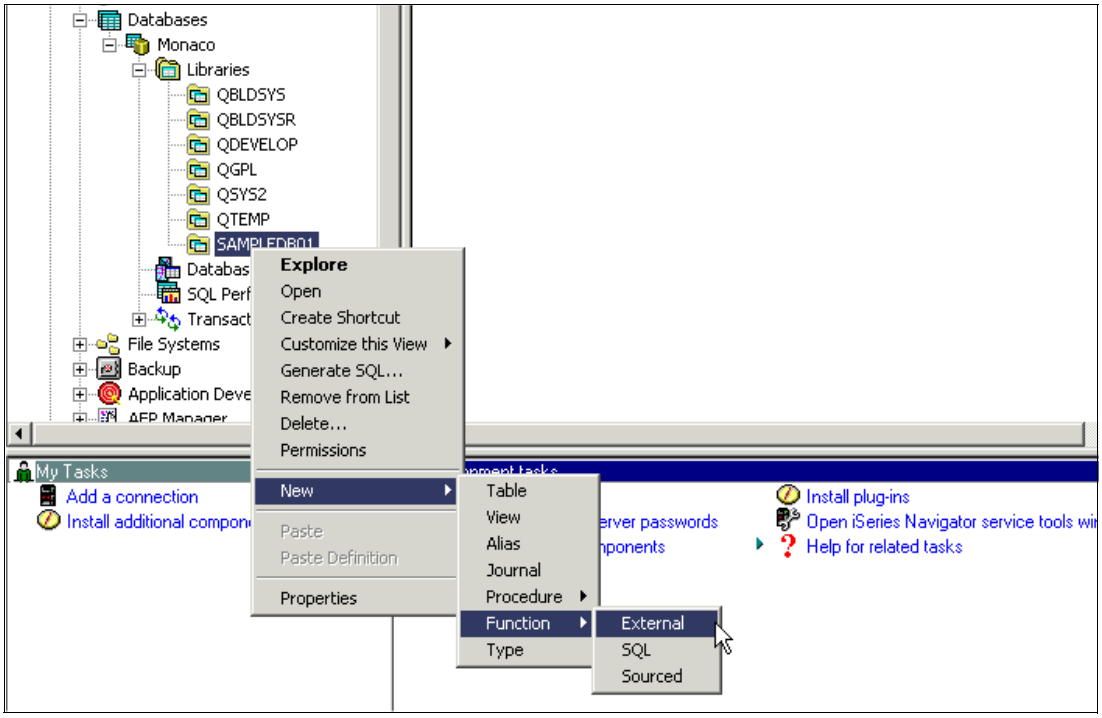


Figure 11-1 Creating an external UDF with System i Navigator

3. The New External Function window (Figure 11-2) opens. Follow these steps:

   a. On 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** if you want to return a scalar, or click **Table** if you want to create a user-defined table function (UDTF). In our DEC2DATE example, we choose **Single value** and select the return type **DATE**.

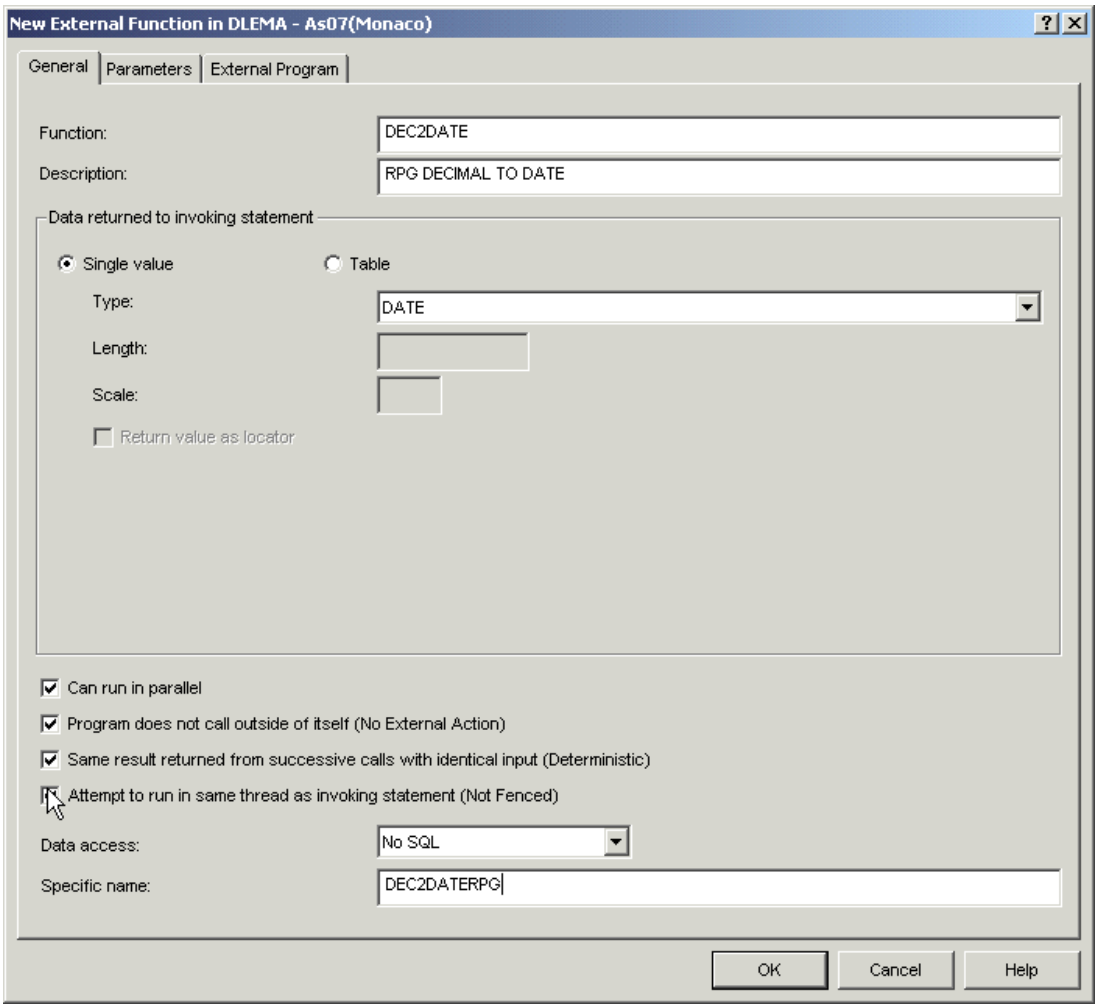


Figure 11-2 Creating an external UDF - General tab

   b. At the bottom of the tab, you are presented with 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 whether the function performs an external action, such as inserting, updating, or deleting rows in a table, or calls a function or stored procedure that performs an 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): Check this option if the function will always return the same result from successive invocations when identical input arguments are provided, which is the case in our example.
- Attempt to run in same thread as invoking statement (Not Fenced): The function can run (NOT FENCED) or cannot 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, the function can run in the same thread.

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.

4. Click the **Parameters** tab. Define the input parameters for the UDF, as shown in Figure 11-3.

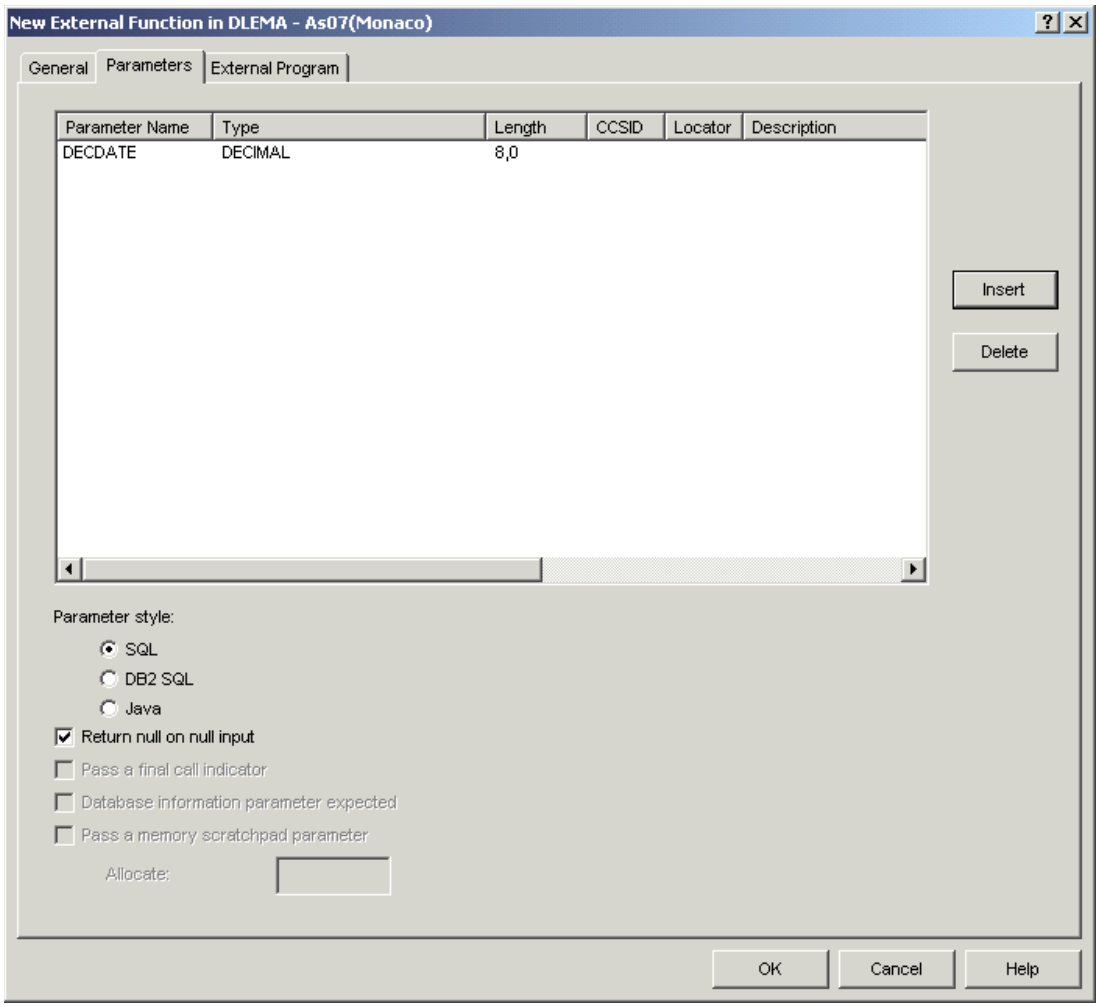


Figure 11-3 Creating an external UDF - Parameters tab

In addition, select the parameter style. Parameter styles are explained in 11.3, “Parameter styles in external UDFs” on page 346.

5. Click the **External Program** tab (Figure 11-4). Enter the characteristics and the location of the external program. In our case, the program is an RPGLE program that returns a DATE.

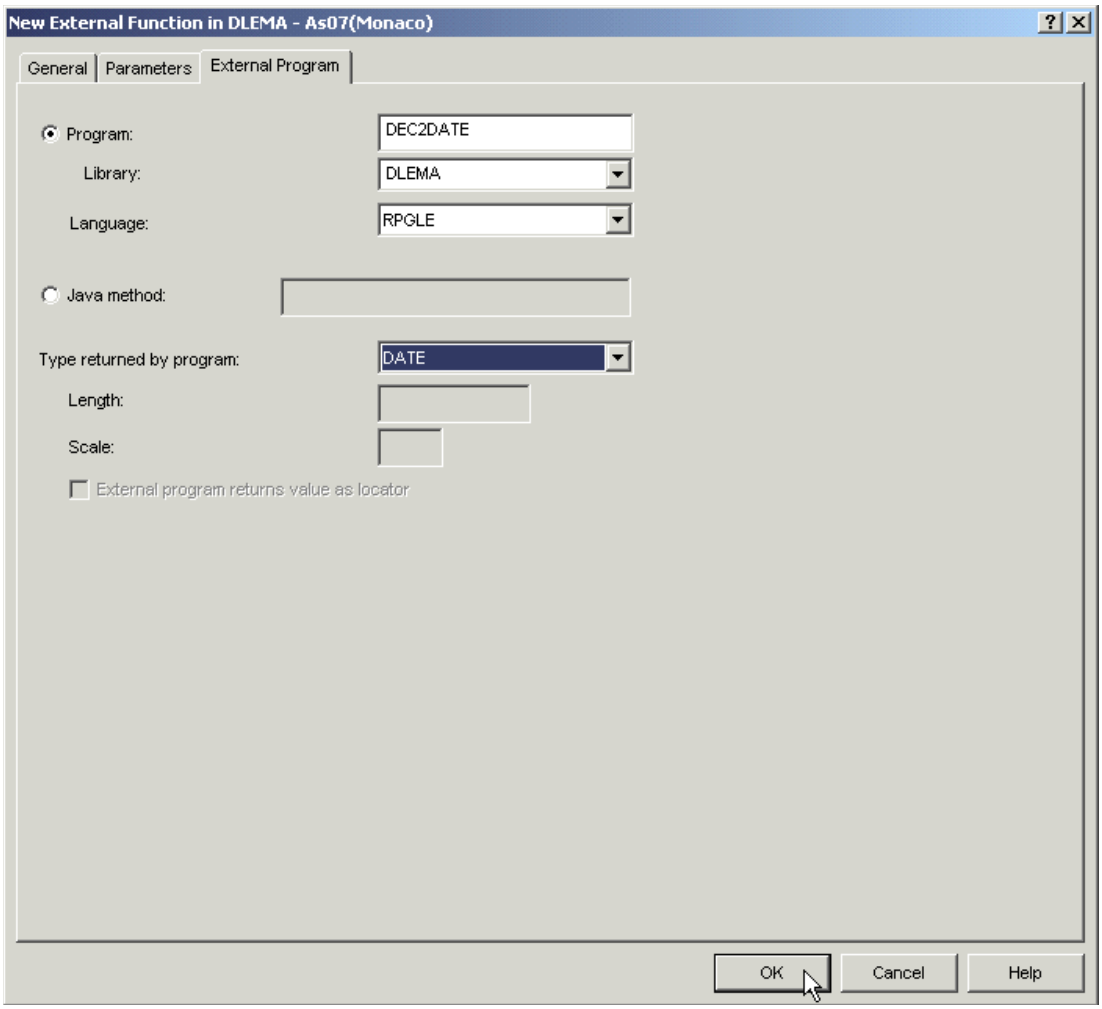


Figure 11-4 Creating an external UDF - External Program tab

Registering an external UDTF

In the following example, we register an external UDTF that is based in a C program. This program reads data from a stream file in the integrated file system (IFS) and returns the result as a table. The source program is in 11.4, “Scratchpad in UDFs and UDTFs” on page 350. Follow these steps:

1. Double-click the **System i Navigator** icon on your desktop. Under My Connections, double-click the IBM i 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 exist. In our case, the name of the library is `SAMPLEDB01`. Select **New** → **Function** → **External**.

3. The New External Function windows opens. Follow these steps:

   a. On the General tab (Figure 11-5), enter a meaningful name for the UDTF 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 **Table**. In our F1Results example, we choose the following columns:

   - DRIVER_NBR
   - DRIVER_NAME
   - GENERAL_POSITION
   - LAST_RACE_POSITION
   - CONSTRUCTOR
   - WHEEL_BRAND
   - DRIVERS_COUNTRY
   - YTD_POOLS
   - YTD_WINS
   - YTD_POINTS

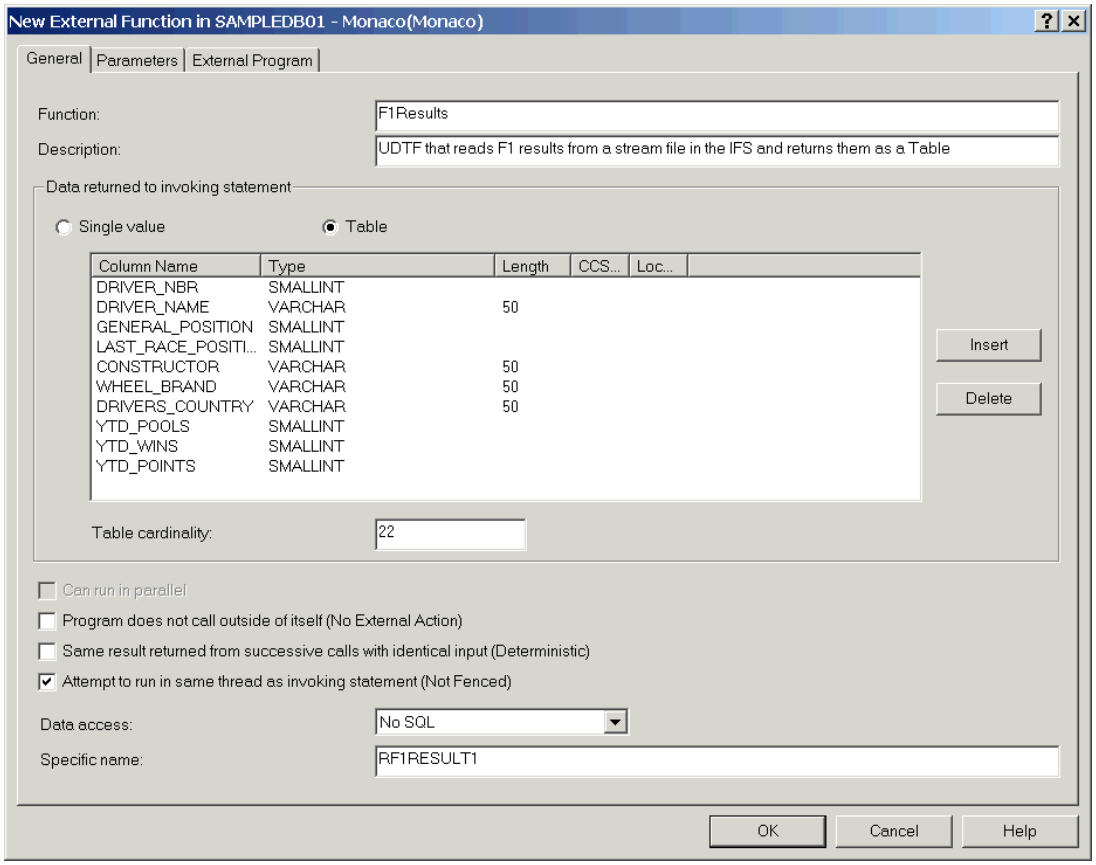


Figure 11-5 Creating an external UDTF - General tab

   b. At the bottom 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). (Default.)

   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.

4. Click the **Parameters** tab (Figure 11-6). Define the input parameters for the UDTF. In our case, the input parameters are the name of the file in the IBM i server IFS.

   Also, define the parameter style. In our case, we choose **DB2SQL** because it is not a Java program. Define an amount of memory for the scratchpad. We define a space of 100 bytes for the scratchpad memory space.

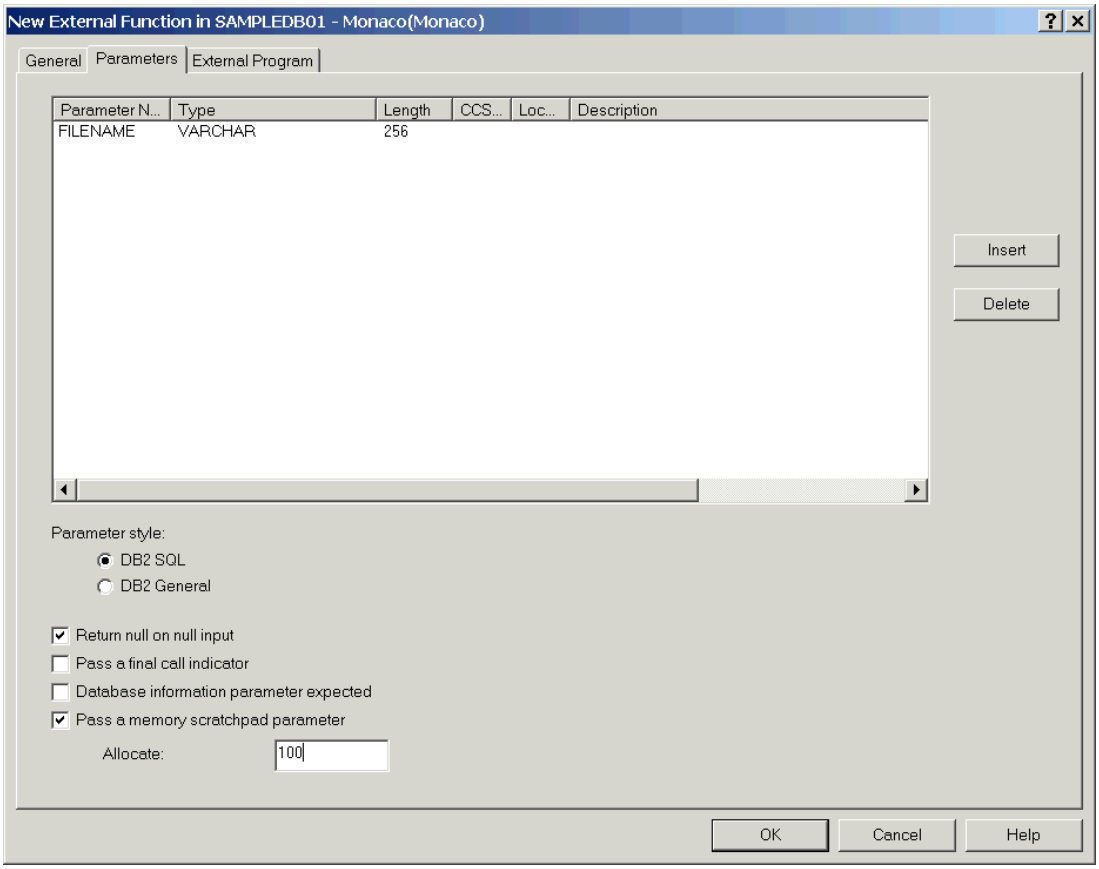


Figure 11-6 Creating an external UDF - Parameters tab

5. Click the **External Program** tab. 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 can contain multiple entry points. When the entry point name differs from the object name, the program is identified by the object name followed by the entry point in parentheses, as shown in Figure 11-7.

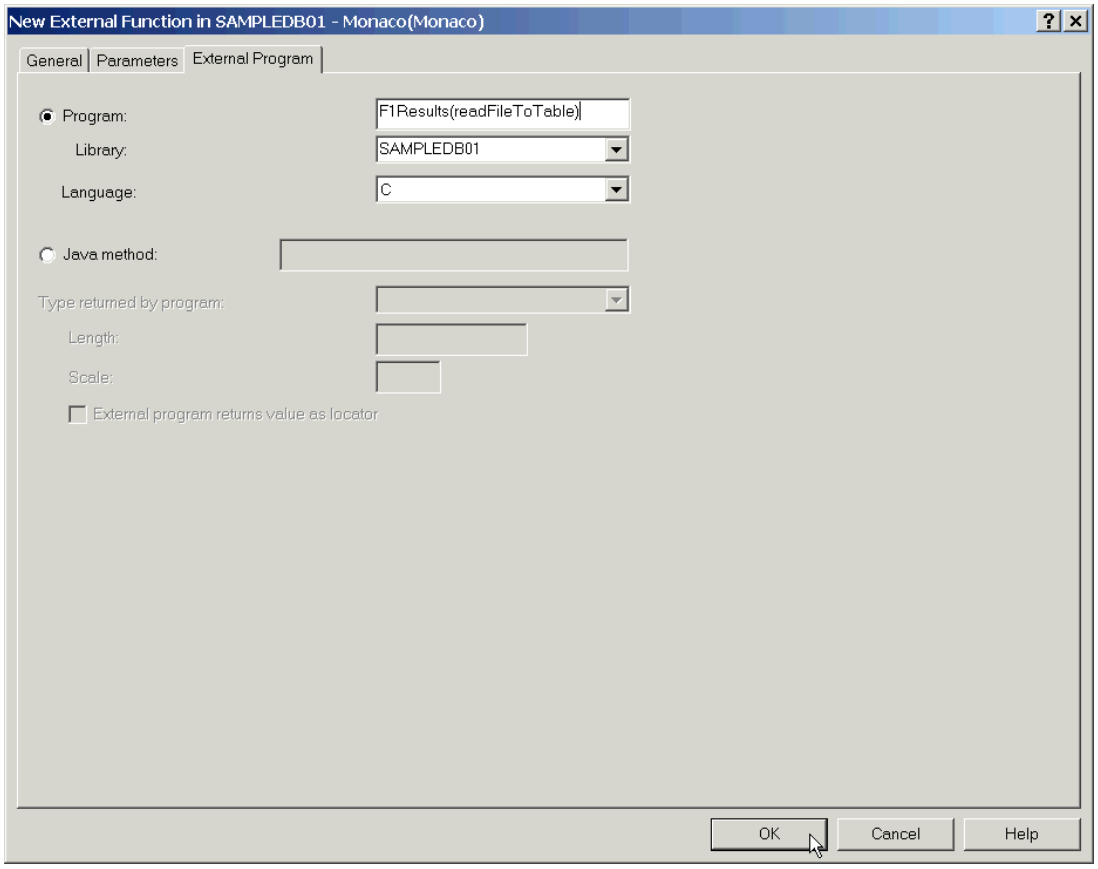


Figure 11-7 Creating an external UDTF - External Program tab

11.2.2 Registering a Java UDF with System i Navigator

The following steps show how to create a Java scalar and table UDF by using System i Navigator.

Registering a Java scalar UDF

In the following example, we register an external UDF that is based in a Java program. The program calculates the number of the working days between two dates. Follow these steps:

1. Double-click the **System i Navigator** icon on your desktop. Under My Connections, double-click the IBM i 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. Follow these steps:

   a. The General tab is shown in Figure 11-8. 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 **Single value**. In our WORKING_DAYS example, we choose **Single value** and select the return type **BIGINT**.

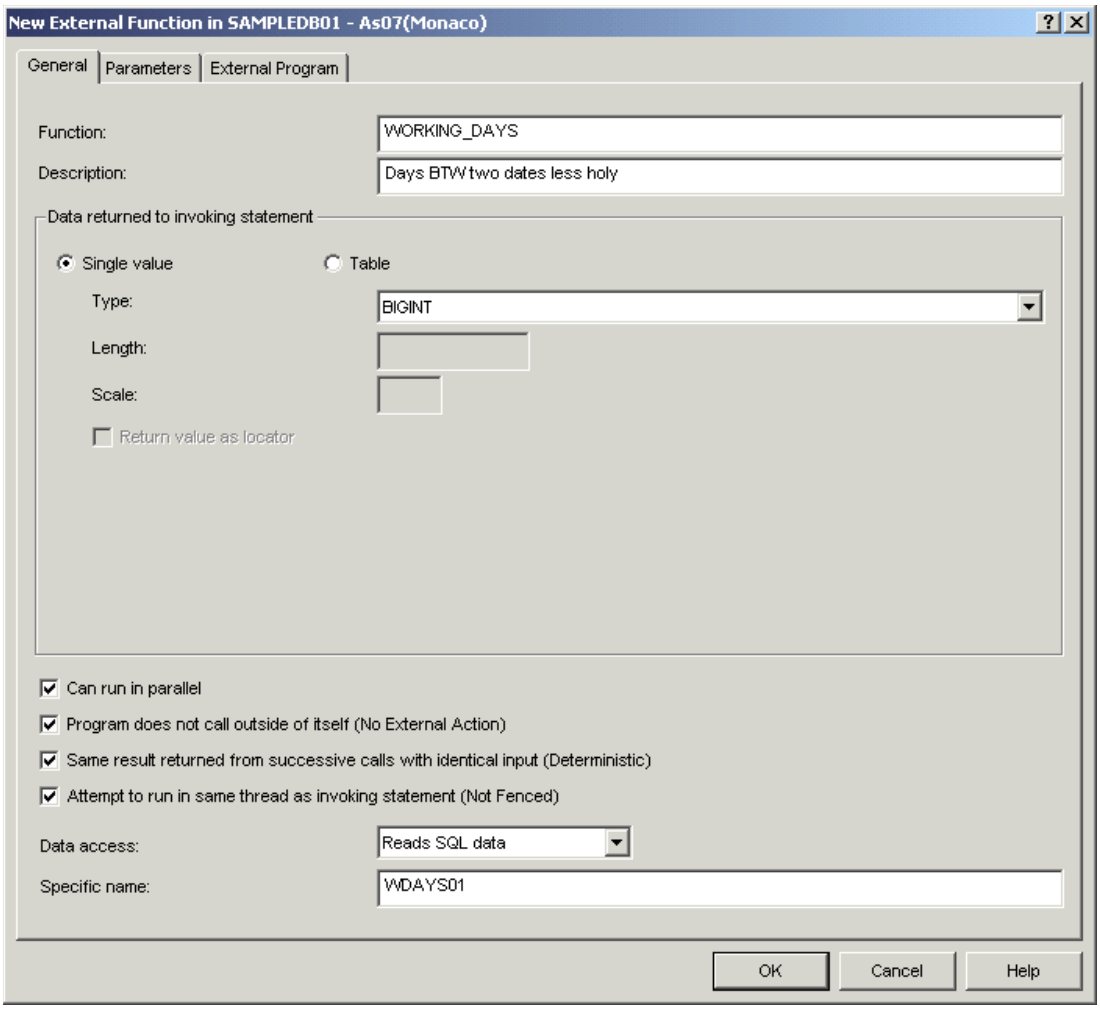


Figure 11-8 Creating External UDF (Java) General tab

   b. At the bottom 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. Even within a country, we might have different calendars for specific purposes.

      The specific name is defined as `WDAYS01`.

5. Click the **Parameters** tab (Figure 11-9). Define the input parameter. For example, we define the initial date, final date, and an identifier (CALENDARID) for a particular calendar.

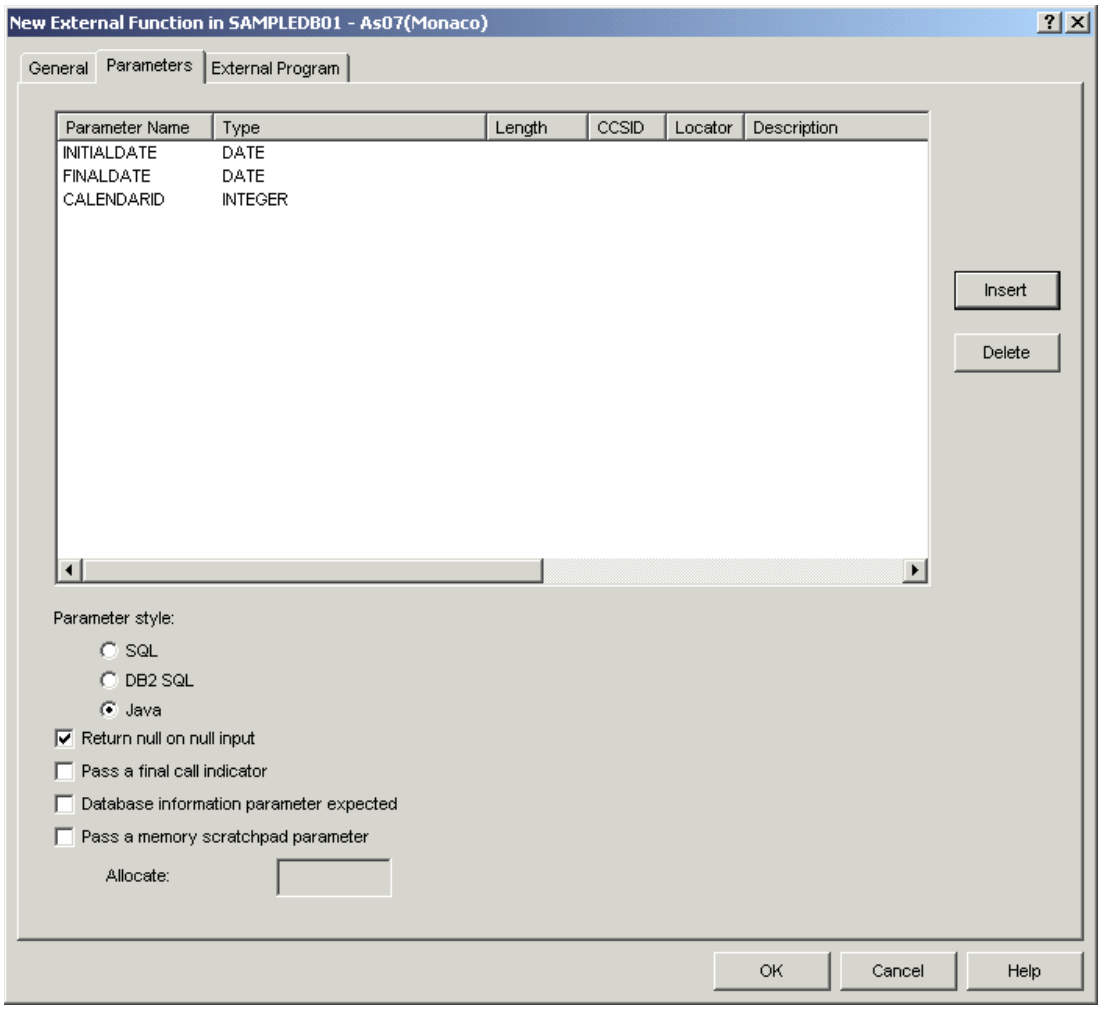


Figure 11-9 Creating an external UDF (Java) Parameters tab

In this case, our program is developed in Java. For Java, the JAVA and DB2GENERAL parameter styles are supported. We choose the **JAVA** parameter style because it is more portable among platforms. The System i Navigator GUI does not allow the DB2GENERAL parameter style. If a Java UDF or UDTF with DB2GENERAL parameter style must be created, you need to use another creation method, such as RUN SQL Statement, to create it.

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 it will return a null.

6. Click the **External Program** tab. 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 can have multiple methods to implement both UDFs and stored procedures. That class must be on the `/QIBM/UserData/OS400/SQLLib/Function` path. The method is identified by the class name followed by a dot (.) or an exclamation point (!) followed by the method name, as shown in Figure 11-10. The Java class and method name are case-sensitive. For performance, we recommend that you use Java archive (JAR) files. DB2 for i provides built-in stored procedures for managing JAR files, as presented in 5.7, “SQLJ procedures to manipulate JAR files” on page 119.

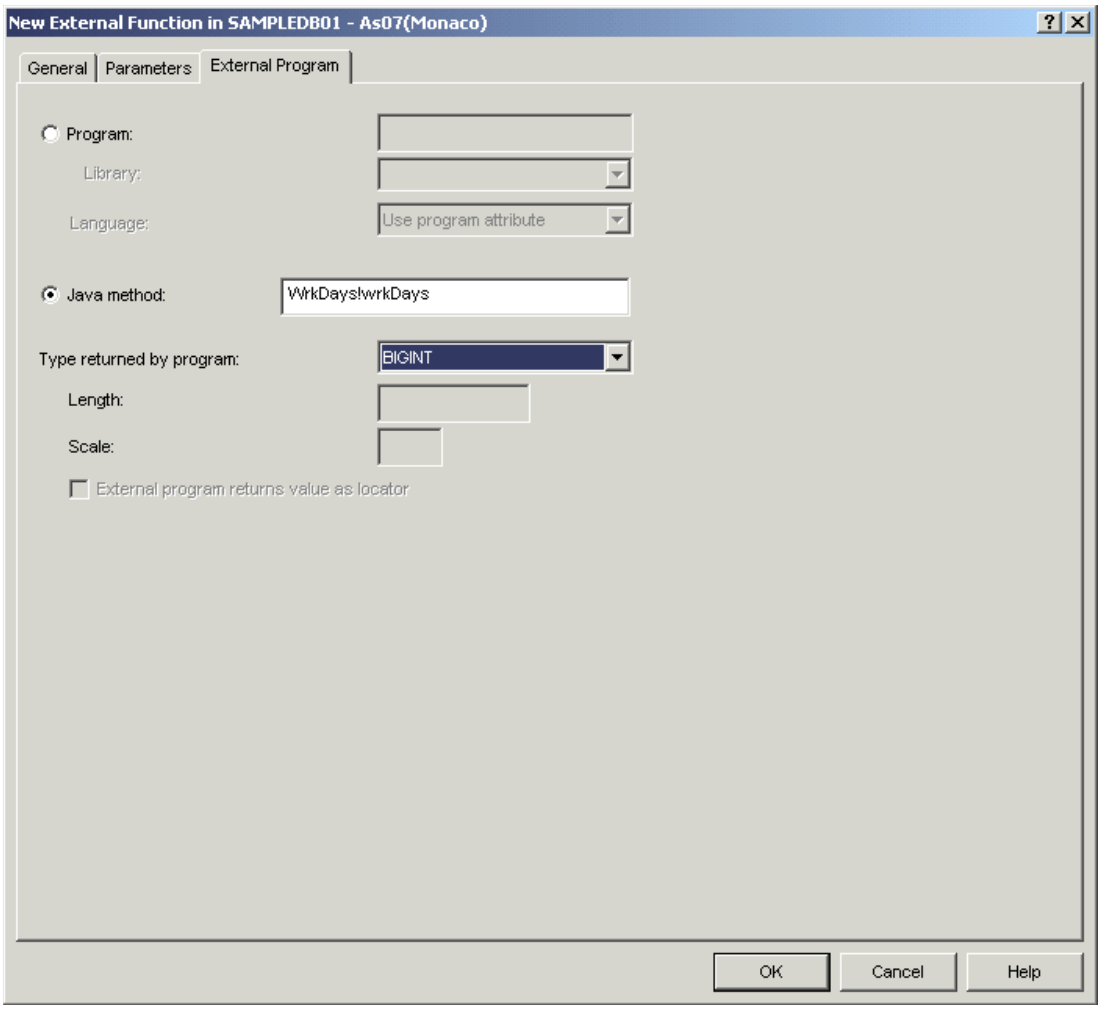


Figure 11-10 Creating an external UDF (Java) External Program tab

Note: Be careful when you define the type that is returned by the program. You can check the data type compatibility in Table 5-1 on page 96.

Registering a Java UDTF

In the following example, we register an external UDTF that is based in a Java program. This program returns a table that exposes the properties, which are set in Java virtual machines (JVMs), that are used for Java stored procedures and Java UDFs.

Follow these steps:

1. Double-click the **System i Navigator** icon on your desktop.
2. Under My Connections, double-click the IBM i 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. On the General tab (as shown in Figure 11-11), 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 **Table**. In our JVMProperties example, we choose a table with two columns, one column that is named Property and the other column with the name Value. Both names are VARCHAR type of length `500`.

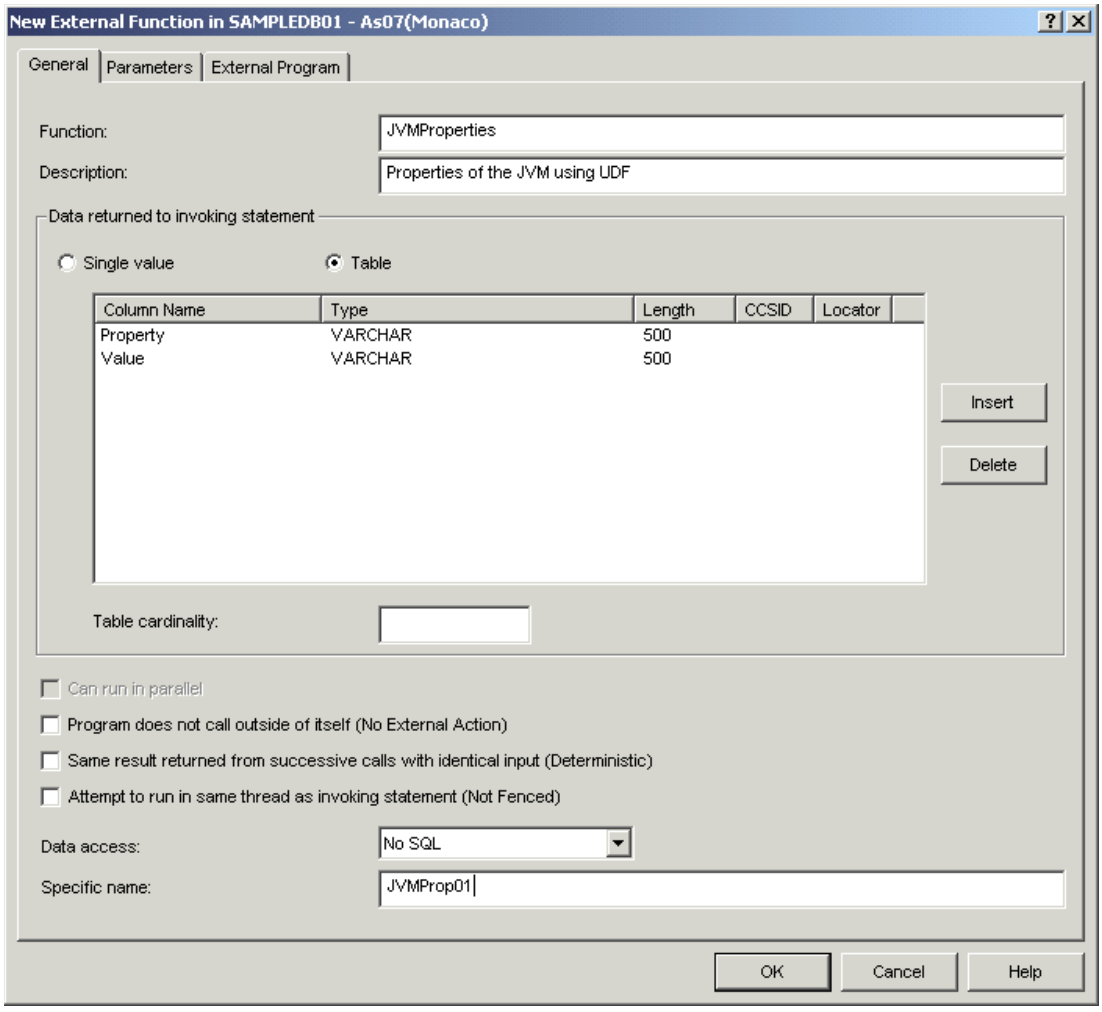


Figure 11-11 Creating an external UDTF (Java) General tab

4. Click the **Parameters** tab, define the parameters to pass to the UDTF. Even if no parameters will be passed to the UDTF, a parameter style must be selected. In this case, we select the parameter style **DB2 General** because it is the only parameter style that is supported by Java UDTFs.

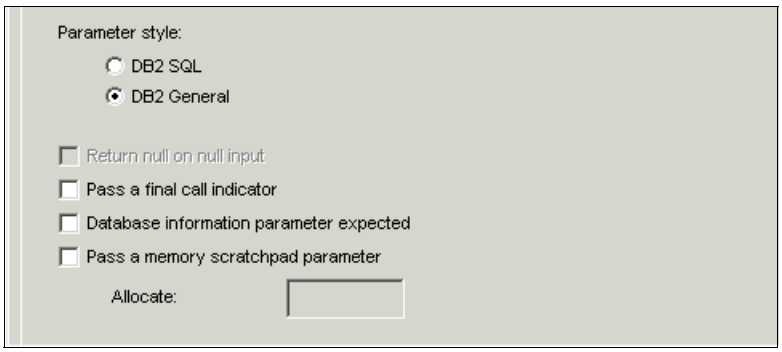


Figure 11-12 Creating an external UDTF (Java) Parameters tab

5. Click the **External Program** tab. Type the characteristics and the location of the external program, as shown in figure Figure 11-13. The rules for identifying the method that implements the UDTF are the same as in the scalar Java UDF and stored procedure that were already presented.

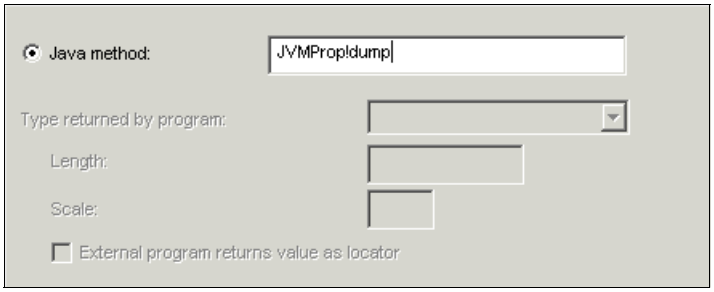


Figure 11-13 Creating an external UDTF (Java) External Program tab

11.3 Parameter styles in external UDFs

The *parameter style* is used to specify the conventions that are 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 the input parameters that you provide. The number and type of extra parameters that are passed by DB2 Universal Database depend on the parameter style. You can specify the required parameter style at the time that the function is created. DB2 for i supports six parameter styles:

- SQL
- DB2SQL
- GENERAL
- GENERAL WITH NULLS
- DB2GENERAL
- Java

These parameter styles are the same as the parameter styles that were presented in 4.2, “Parameter styles in external stored procedures” on page 45, and 5.2, “Coding DB2 for i Java stored procedures” on page 93.

In this section, we describe the parameters that apply to UDFs. We also provide an example for each parameter style.

11.3.1 SQL parameter style

The SQL parameter style conforms to industry standard SQL. This parameter style can be used in scalar UDFs only. The required set of parameters for this parameter style are shown:

```
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 of the SQL parameter style in UDFs compared to stored procedures are listed:

- *Parameters are input parameters*: All parameters are input parameters, except for the last parameter, which is the result.
- *Result: Result value:* This parameter is set by the UDF before it returns to DB2. DB2 UDF for i provides the storage for the returned value.
- *Result indicator*: This parameter must be set by the UDF before it returns to DB2. It is a 2-byte signed integer that is either a positive or negative value. If this parameter is set to a negative value, the UDF result is interpreted as null. If this parameter is set to zero or a positive value, the result is interpreted as not null.
- *Function name*: This parameter is set by DB2 before it calls the UDF. It is a VARCHAR(139) that contains the fully qualified function name, following the SQL naming standard.

11.3.2 DB2SQL parameter style

All of the parameters that are passed to a function for the SQL parameter style are also passed to a function with the DB2SQL parameter style. However, the DB2SQL parameter style allows *additional* parameters to be passed. This parameter style can be used for both scalar and table UDFs. The supported set of parameters for this parameter style are shown:

```
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, which were not covered in the previous section, are explained in the following list:

- **Scratchpad**: This parameter is set by DB2 before it calls the UDF if the SCRATCHPAD clause was specified in the CREATE FUNCTION statement. This parameter can be used by the UDF as an area where temporary values can be saved for use 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 can see the results that are 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 in the scratchpad area by a UDF only during the processing of a specific SQL statement, which can be important for a UDTF that is used in a join or subquery.

  If it is necessary to maintain the content of the scratchpad across OPEN calls, 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* call and a *final* call. These first and final calls can be used for scratchpad maintenance and releasing resources.

- **Call type**: This argument is set by DB2 before it calls 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, it is always present.

  Three values are valid for scalar UDFs:

  -1 First call to the UDF.

  0 Normal call to the UDF. All of the normal input argument values are passed.

  1 Final call to the UDF. No SQL argument or SQL argument indicator values are passed. A UDF must not return any answer by using the SQL result, SQL result indicator, SQL state, or diagnostic message arguments.

  Five values are valid for UDTFs:

  -2 First call to the UDF.

  -1 Open call to the UDF. The scratchpad is initialized if NO FINAL CALL is specified. ALL SQL argument values are passed.

  0 Fetch call to the UDF. DB2 expects the table function to return either a row that consists of the set of the result values, or an end of table condition that is indicated by SQLSTATE '02000'.

  1 Close call to the UDF. This call balances the OPEN call, and it can be used to perform any CLOSE processing and to release resources.

  2 Final call to the 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 parameter is an optional input parameter.

- **dbinfo**: This parameter is for the DBINFO structure if the DBINFO clause is specified on the CREATE FUNCTION statement. See the sqludf.h include file in the QSYSINC library for a detailed definition of this structure.

11.3.3 GENERAL parameter style

The following example shows the supported set of parameters for this parameter style:

```
externalUDF(IN arguments (repeated))
```

When this parameter style is used, the result is the value that the program returns. For this reason, this parameter style can be used with scalar UDFs only.

11.3.4 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 in the following manner:

```
ExternalUDF(IN parameter (repeated),
IN parameter indicator array,
OUT result indicator)
```

Parameter indicator array

The parameter indicator array can be used by the UDF to determine whether one or more parameters are not null. Each entry of the array is set to one of the following values:

`0`	The parameter is present (not null).
`-1`	The parameter is empty or null.

This parameter is treated as input only.

11.3.5 DB2GENERAL parameter style

Parameter style DB2GENERAL is one of two parameter styles that are supported in Java UDFs. In this parameter style, the return value is passed as the last parameter of the function. The parameter style must be set by using a `set` method that is inherited from the `com.ibm.db2.app.UDF` class. When you code a UDF with the DB2GENERAL parameter style, follow these conventions:

- The class that includes the Java UDF must extend or be a subclass of the `com.ibm.db2.app.UDF` Java class.
- The Java method must be a `public void` method.
- The parameters for the Java method must be compatible with SQL. See 5.2.2, “Data type compatibility” on page 96.
- The Java method can test for an SQL NULL value by using the `isNull()` method.
- The Java method must explicitly set the return parameter by using the `set()` method.

The main advantages of the DB2GENERAL parameter style over the JAVA parameter style are shown:

- You can use the DB2GENERAL parameter style to test for null parameters of any data type, including those data types that map to Java data types that do not support null values, such as INTEGER.
- The DB2GENERAL parameter style supports UDTFs.

The DB2GENERAL parameter style has a disadvantage that you must consider. It is not standard, which makes it less portable than the JAVA parameter style.

11.3.6 JAVA parameter style

Consider the following conventions when you create a UDF with the JAVA parameter style:

- The Java method must be `public static`.
- The Java method must return a type that is compatible with SQL. The returned value is the result of the UDF.
- The Java method can test for an SQL NULL for Java types that permit null values.

11.4 Scratchpad in UDFs and UDTFs

The *scratchpad* is a memory area that is provided by DB2 for i 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 contains an 8-byte binary number (equivalent to a C *long* field) that contains the size of the scratchpad followed by a byte array of the specified size.

The size of the scratchpad is established in the SCRATCHPAD clause of the CREATE FUNCTION statement. If a size is not specified, the size is set to the default size, which is 100 bytes.

The following SQL statement is an example:

```
SELECT UDF_W_SCRATCHPAD(ORDERDATE), UDF_W_SCRATCHPAD(SHIPDATE)
FROM SAMPLEDB01.ITEM_FACT
WHERE SHIPMODE = 'EXPRESS'
```

In the previous statement, DB2 for i will reserve two memory areas for scratchpad, one for each reference to the UDF_W_SCRATCHPAD function. In this case, two statement scopes exist for the UDF_W_SCRATCHPAD function. The scratchpad is initialized with binary zeros by the database manager before it calls the UDF for the first time in a statement scope.

For UDTFs, the FINAL CALL clause of the CREATE FUNCTION statement affects how 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 is initialized for each OPEN call.

Scratchpad can be used as a mechanism to allow the UDF to conserve information between calls. In Example 11-7 on page 358, the scratchpad is used to track the next number to generate as an ID. In Example 11-16 on page 369, a scratchpad is used to maintain information about a stream file that was opened at the first call and read among callings to form the return table.

Note: SCRATCHPAD can be used with the DB2SQL or DB2GENERAL parameter style.

11.5 UDF and UDTF calling sequence

Now that you know what UDFs and UDTFs are, it is important to understand when external programs are called and how many times.

The external program that implements a UDF or UDTF is called one time at the beginning of the statement scope with the first row, one time for each of the subsequent retrieved rows in the statement, and one time at the end of the statement.

The following SQL statement is an example:

```
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', our FX_RATE UDF is called 11 times. Depending on the parameter style that is used, the call type can be retrieved by the following methods:

- DB2SQL parameter style: An input parameter that is provided by the database manager, as illustrated in the example that is presented in 11.3.2, "DB2SQL parameter style" on page 347.
- DB2GENERAL parameter style: By using the `getCallType()` Java method, which is inherited from the UDF class. The values that are returned by this method are the same values that are described in 11.3.2, "DB2SQL parameter style" on page 347.

  The UDF class provides constant definitions to make your Java program code more readable. For C programmers, the `sqludf.h` header file defines the same set of constants. Table 11-1 shows the constants for scalar UDFs.

Table 11-1 Predefined constants for call types in scalar UDFs

Constant name	Description
SQLUDF_FIRST_CALL	First call, which is only made if FINAL CALL was specified in the CREATE FUNCTION statement
SQLUDF_NORMAL_CALL	Fetch next row
SQLUDF_FINAL_CALL	Final call, which is only made if FINAL CALL was specified in the CREATE FUNCTION statement

Table 11-2 shows the constants for UDTFs.

Table 11-2 Predefined constants for call types in UDTFs

Constant name	Description
SQLUDF_TF_FIRST	First call, which is only made if FINAL CALL was specified in the CREATE FUNCTION statement
SQLUDF_TF_OPEN	Open table
SQL_TF_FETCH	Fetch next row
SQL_TF_CLOSE	Close table
SQLUDF_TF_FINAL	Final call, which is only made if FINAL CALL was specified in the CREATE FUNCTION statement

11.6 Coding an external UDF

External UDFs are regular HLL programs that are registered in DB2 for i. They do not differ significantly from a regular HLL program that you write, except for certain conventions that you must observe to coordinate the parameter passing between DB2 and your HLL program.

In this section, we present small programs to illustrate each parameter passing technique, error handling, and scratchpad usage.

11.6.1 Coding the SQL parameter style

This section shows examples to code external UDFs with the SQL parameter style. It also demonstrates how the parameters that are passed by DB2 Universal Database for iSeries to the external UDF can be used within the function.

Assume that you want to create a function, DEC2DATE, that converts a nonstandard date that is defined as DECIMAL(8,0) and stored in *YYYYMMDD* format to the DATE data type, which in fact is a common problem in real life.

Examine the CREATE FUNCTION statement for the DEC2DATE external UDF, as shown in Example 11-1.

Example 11-1 CREATE FUNCTION for the RPG SQL parameter style of DEC2DATE

```
CREATE FUNCTION SAMPLEDB01.DEC2DATE2 (
   DECDATE DECIMAL(8, 0) )                          1
   RETURNS DATE                                     2
   LANGUAGE RPGLE                                   3
   SPECIFIC SAMPLEDB01.DEC2DATERPG                  4
   DETERMINISTIC
   NO SQL
   RETURNS NULL ON NULL INPUT
   NO EXTERNAL ACTION
   NOT FENCED
   EXTERNAL NAME 'SAMPLEDB01/DEC2DATE'              5
   PARAMETER STYLE SQL ;                            6

COMMENT ON SPECIFIC FUNCTION SAMPLEDB01.DEC2DATERPG
   IS 'RPG DECIMAL TO DATE' ;
```

CREATE FUNCTION statement explanation

The following notes refer to the numbers in Example 11-1:

1 Qualified procedure name: If qualification is not provided, the implicit qualification rules apply. That is, the unqualified alias, constraint, external program, index, node group, package, sequence, table, trigger, and view names are implicitly qualified by the default schema. This function receives one input parameter that is called **DECDATE** of type DECIMAL(8,0). The parameter name is for documentation only. The parameter name does not need to relate to parameter or variable names that are used in the program. Function overloading rules apply, as explained in 10.5.1, “UDF overloading and function signature” on page 325.

2 Return type of the function. In this case, the return is of the 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 that is required by the programming language.

External UDFs can be written in any of the following languages:

- C
- C++
- COBOL
- COBOLLE
- FORTRAN
- JAVA
- PL/I
- RPG
- RPGLE

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information that is associated with the external program at the time that the function is created. The language of the program is assumed to be C as the default if the following conditions are true:

- The program attribute information that is 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 that is created on the IBM i server must have a specific name. This name must be unique for the specific library. This clause is optional. If you do not specify a specific name for the function, the system generates a specific name. Normally, the specific name is the same as the function's name. However, if a function with the specific name exists, the system generates a unique name.

5 EXTERNAL NAME clause of the CREATE FUNCTION statement. This clause 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 that 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 that 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 does not need to exist at the time that the function is created, but it must exist at the time that 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 that are defined in the CREATE FUNCTION statement, based on the specified parameter style, as described in 11.3, "Parameter styles in external UDFs" on page 346.

Now, examine the external program DEC2DATE, which is referred to in the CREATE FUNCTION statement. We describe the parameters that DB2 Universal Database for iSeries sends to the program and how the program uses these parameters. This program was written in RPG, as shown in Example 11-2. The DEC2DATE external program accepts an 8-digit decimal with a date that must be converted to DATE format as the input argument.

Example 11-2 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'             5
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, which is referred to in the CREATE FUNCTION statement that is shown:

```
EXTERNAL NAME 'DLEMA/DEC2DATE'
```

The following special comments refer to the numbers in the source code in Example 11-2:

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 International Organization for Standardization (ISO) format.

3 The *INDATE* variable is defined as an 8-digit packed decimal.

4 From line 8 to line 11, the parameters that are received by the RPG program are specified according to the parameter style that is used. In this case, they are the SQLSTATE, function name, specific name, and error message parameters. Several of those variables are useful for error management.

5 You do not want *INLR = '1' in your program. *INLR = '1' causes the program to be released from memory after the program finishes. But a UDF, such as this UDF, which is typically called many times, adversely affects performance.

A variation of this code is presented in Example 11-3. The program is not disposed of because of the line in bold. It simply returns without forcing the program to release the memory. Although we did not perform a rigorous test, the second DEC2DATE version outperforms the first version by a factor of three.

Example 11-3 Variation of 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
*-------------------------------------------
```

These DEC2DATE programs are *PGM objects. The program is compiled into the *MODULE object. Then, the *MODULE object is bound into a *PGM object so that we can specify the activation group parameter as *CALLER.

Note: The external program that is coded in any host language needs to be complied with the activation group parameter *CALLER.

The **CRTBNDRPG** control language (CL) command is used to compile and bind the DEC2DATE program, as shown in Figure 11-14.

```
                        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 11-14 Create Bound RPG Program

When you work with UDFs, triggers, and stored procedures, you use service programs more often than programs. Example 11-4 shows the same DEC2DATE code with its *SRVPGM version.

*Example 11-4 Second variation of RPG ILE DEC2DATE UDF as a *SRVPGM*

```
**************************************************************
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 errormsg                      70A   OPTIONS(*VARSIZE) VARYING
pdec2date         B                   export
 *-------------------------------------------
 *PARAMETER DEFINITION
 *-------------------------------------------
ddec2date         pi
d indate                         8P 0
d outdate                         D   datfmt(*ISO)
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 errormsg                      70A   OPTIONS(*VARSIZE) VARYING
 *-------------------------------------------
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
 *----------------------------------------------
pdec2date         E
```

A service source program can contain several different routines or entry points. This way, if we need different versions of our date casting function, one version to convert from integer to DATE, another version to convert from CHAR(8) to DATE, and so on, we can put them all together in the same source so that its management and maintenance are easy. A service source program can contain routines for multiple purposes.

11.6.2 Coding the DB2SQL parameter style

This section describes how to code external UDFs with the DB2SQL parameter style.

Note: The DBINFO option works in a similar way to the DBINFO option in stored procedures, which is explained in 4.3.2, “Coding the DB2SQL parameter style” on page 54.

Assume that you want to massively generate identifiers 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 track the next number to generate, but the application designer was interested in code that can be ported to other platforms, and the application designer asked us not to use the data area.

Then, we thought of a table that tracks the next consecutive number to generate. Because this table might cause a bottleneck, instead of getting the next number in the sequence, we built a UDF that obtains the next group of numbers and uses the SCRATCHPAD to generate individual values until the end of the group is reached.

We examine the CREATE FUNCTION statement for the GENOID external UDF.

Example 11-5 CREATE FUNCTION statement for an external UDF that uses scratchpad

```
CREATE FUNCTION SAMPLEDB01.GENOID (
   CHAR(20) )
   RETURNS DECIMAL(30, 0)
   LANGUAGE RPGLE
   SPECIFIC SAMPLEDB01.GENOID00
   DETERMINISTIC
   MODIFIES SQL DATA
   RETURNS NULL ON NULL INPUT
   SCRATCHPAD 100                                          1
   EXTERNAL NAME 'SAMPLEDB01/GENOID(GENOID)'               2
   PARAMETER STYLE DB2SQL;
```

CREATE FUNCTION statement explanation

The elements that change from parameter style SQL to parameter style DB2SQL (Example 11-5 on page 357) are explained in the following list:

1. This argument is set by DB2 before it calls 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 zeros by DB2 before the first call to the UDF. Here, we defined a length area of 100 bytes. Therefore, the system will reserve 100 bytes of memory for the scratchpad area and send the address of this area to the function program.

2. This name 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 to execute, 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 the program must be created before the function is invoked for the first time. This clause is an optional clause. If it is not specified, the system assumes that the name of the program to execute is the same as the name of the function.

Now, we examine the service program GENOID, which is referred to in the CREATE FUNCTION statement in Example 11-5 on page 357. The program has two parts. The first part (Example 11-6) contains the definition of the RPG module.

Example 11-6 Part 1 - Module definition of 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                                      1
0007.00 D  FunctionName               517A   CONST OPTIONS(*VARSIZE) VARYING    2
0008.00 D  SpecificName               128A   CONST OPTIONS(*VARSIZE) VARYING    3
0009.00 D  DiagMsg                     70A   OPTIONS(*VARSIZE) VARYING          4
0010.00 D  ScratchPad                 104A   OPTIONS(*VARSIZE) VARYING          5
```

The second part of the program (Example 11-7) contains the module code.

Example 11-7 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                                      1
0011.00 D  FunctionName               517A   CONST OPTIONS(*VARSIZE) VARYING    2
0012.00 D  SpecificName               128A   CONST OPTIONS(*VARSIZE) VARYING    3
0013.00 D  DiagMsg                     70A   OPTIONS(*VARSIZE) VARYING          4
0014.00 D  ScratchPad                 104A   OPTIONS(*VARSIZE) VARYING          5
0015.00
0016.00 D ScratchPadDs    DS                  BASED(ScratchPadPtr)               6
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)                          7
0027.00 C                   IF        FirstCall     = 0                        8
0028.00 C                   EVAL      ChkValue      = 0
0029.00 C                   EVAL      Counter       = 0
0030.00 C                   EVAL      FirstCall     = -1
0031.00 C                   ENDIF
0032.00 C                   IF        Counter       = ChkValue                 9
0033.00 C/EXEC SQL
0034.00 C+    SELECT COUNTER INTO :Counter FROM SAMPLEDB01.COUNTER
0035.00 C+    WHERE COUNTER_NAME = :Counter_name
0036.00 C/END-EXEC
0037.00 C                   IF        SQLCOD        = -204                     10
0038.00 C/EXEC SQL
0039.00 C+    CREATE TABLE SAMPLEDB01.COUNTER (COUNTER_NAME CHAR(20) NOT NULL,
0040.00 C+                                     COUNTER      DEC(30,0) NOT NULL
0041.00 C+                                                  WITH DEFAULT,
0042.00 C+                                     PRIMARY KEY ( COUNTER_NAME ) )
0043.00 C/END-EXEC
0044.00 C/EXEC SQL
0045.00 C+    SELECT COUNTER INTO :Counter FROM SAMPLEDB01.COUNTER
0046.00 C+    WHERE COUNTER_NAME = :Counter_name
0047.00 C/END-EXEC
0048.00 C                   ENDIF
0049.00 C                   IF        SQLCOD        = 100                      11
0050.00 C                   EVAL      Counter       = 1
0051.00 C                   EVAL      ChkValue      = 20
0052.00 C/EXEC SQL
0053.00 C+    INSERT INTO SAMPLEDB01.COUNTER
0054.00 C+    VALUES (:Counter_name, 21)
0055.00 C/END-EXEC
0056.00 C                   ELSE
0057.00 C                   EVAL      ChkValue      = Counter + 19
0058.00 C/EXEC SQL
0059.00 C+    UPDATE SAMPLEDB01.COUNTER
0060.00 C+    SET COUNTER = :ChkValue + 1
0061.00 C+    WHERE COUNTER_NAME = :Counter_name
0062.00 C/END-EXEC
0063.00 C                   ENDIF
0064.00 C                   ELSE
0065.00 C                   EVAL      Counter       =  Counter + 1
0066.00 C                   ENDIF
0067.00 C                   EVAL      Output        =  Counter
0068.00 C                   EVAL      Output_ind    =  0
0069.00 P GENOID          E
```

Code sample notes

The following special comments refer to the numbers in the source code in Example 11-6 on page 358 and Example 11-7 on page 358:

1 SQLState parameter that is used for returning the state of the function. The SQLState is a 5-character string that needs a value in one of the following groups:

- `00000`: The UDF completes without errors or warnings.
- `01H`*xx*: The *xx* can be any digits or uppercase letters. The UDF completes without errors but with a warning.
- `38`*yxx*: The *y* is an uppercase letter between I and Z. The *xx* signifies any two digits or uppercase letters. The UDF ends with an error condition.

2 Fully qualified function name. Because the same program might be used in several UDFs, the name of the function, as registered in the DB2 for i catalogs, is received as a parameter and the programmers can use it in their programs.

3 Specific name for the function.

4 Diagnostic text field that might be passed back by the UDF with the warning or error message text.

5 Scratchpad: Input/output parameter with the scratchpad.

6 The scratchpad that is received by the UDF has two values: A long integer that contains the size of the scratchpad data and the data. This value and the following four values redefine the scratchpad to be usable in the program.

7 A pointer to the scratchpad is set so that it is usable in the program.

8 Because the scratchpad is initialized with binary zeros the first time, the UDF is called in an SQL statement scope. When the function is called for the first time, FirstCall is zero.

9 If *Counter* reached the upper value of the group (and at the first call, both Counter and ChkValue are zero), a new upper value must be calculated and recorded on the COUNTER table.

10 SQLCode -204 is fired if the COUNTER table does not exist. In that case, the table must be created and the operation must be tried again.

11 SQLCode 100 is fired if no row exists for the particular counter. A new row is added to the COUNTER table.

The GENOID program was created as a *SRVPGM object. In this case, GENOID is a program that was written in RPG.

The **CRTSQLRPGI** CL command is used to create the module of the program (Figure 11-15). If the program does not contain embedded SQL, the CL command is **CRTRPGMOD**.

```
                        Create SQL ILE RPG Object (CRTSQLRPGI)

Type choices, press Enter.

Object . . . . . . . . . . . . > GENOID        Name
  Library  . . . . . . . . . . >   DLEMA       Name, *CURLIB
Source file  . . . . . . . . . > RPGSRC        Name, QRPGLESRC
  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
                                                                    More...
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
```

Figure 11-15 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')
```

11.6.3 Coding the GENERAL parameter style

The 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. The GENERAL parameter style in UDFs is only supported for external service programs.

In Example 11-8, a UDF is created that is based on an existing COBOL program that returns an exchange rate when it is given an original currency and target currency on a specific date.

Example 11-8 CREATE FUNCTION statement for the GENERAL 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                                          1
   READS SQL DATA
   RETURNS NULL ON NULL INPUT
   EXTERNAL NAME 'SAMPLEDB01/UDF_CBL(GET_FX_RATE)'        2
   PARAMETER STYLE GENERAL ;
```

CREATE FUNCTION statement explanation

By this time, you are already familiar with SQL CREATE statements, but we must highlight the following points in Example 11-8 on page 361:

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 for i will avoid unnecessary executions for the program behind the UDF, which improves performance.
2. Because in UDFs, the GENERAL parameter style is supported for service programs only, the external name must 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, which is in the SAMPLEDB01 library.

Example 11-9 shows a COBOL service program with a GENERAL parameter style UDF.

Example 11-9 COBOL service program for GENERAL parameter style UDF

```
       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  AVALILABLE     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                    1
     BY REFERENCE TARGET_CURRENCY                      1
     BY REFERENCE EXCHANGE_DATE                        1
     RETURNING EXCHANGE_RATE.                          2
 A000-MAIN.
     EXEC SQL
         WHENEVER SQLERROR GO TO E010-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 “UDF0002” 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:

```
CRTSQLCBLI OBJ(SAMPLEDB01/UDF_CBL3) *SRCFILE(SAMPLEDB01/CBLSRC) *SRCMBR(UDF_CBL3)
*OBJTYPE(*MODULE) REPLACE(*NO)
```

The *SRCPGM was created with the following command:

```
CRTSRVPGM SRVPGM(SAMPLEDB01/UDF_CBL3) EXPORT(*ALL)
```

The parameters that are defined for the function are the parameters that the external program receives. Also, the program return will be the UDF return.

11.6.4 Coding the GENERAL WITH NULLS parameter style

The GENERAL WITH NULLS parameter style (Example 11-10) is a variation of the GENERAL parameter style, which was explained in 11.6.3, "Coding the GENERAL parameter style" on page 361.

Example 11-10 CREATE FUNCTION statement for the 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                                          1
   READS SQL DATA
   RETURNS NULL ON NULL INPUT
   EXTERNAL NAME 'SAMPLEDB01/UDF_CBL(GET_FX_RATE)'        2
   PARAMETER STYLE GENERAL WITH NULLS;
```

For the UDF that is defined in Example 11-10, the HLL program has four input parameters and one output parameter. The fourth parameter is a vector of null indicators that correspond to the null state for the first three parameters. The fifth parameter corresponds to the null state for the result that the UDF will return.

11.6.5 Coding the DB2GENERAL parameter style

We examine the CREATE FUNCTION statement for the JVMPROPERTIES external UDF. Example 11-11 shows the CREATE FUNCTION statement for the DB2GENERAL parameter style.

Example 11-11 CREATE FUNCTION statement for DB2GENERAL parameter style

```
CREATE FUNCTION SAMPLEDB01.JVMPROPERTIES ( )
   RETURNS TABLE (
   PROPERTY VARCHAR(500) ,
   VALUE VARCHAR(500) )
   LANGUAGE JAVA                                              1
   SPECIFIC SAMPLEDB01.JVMPROP01
   NOT DETERMINISTIC
   NO SQL
   CALLED ON NULL INPUT
   DISALLOW PARALLEL
   EXTERNAL NAME 'JVMProp!dump'                               2
   PARAMETER STYLE DB2GENERAL ;                               3
```

Note: The following numbered notes refer to Example 11-11:

1. DB2GENERAL applies only for the Java language.
2. `JVMProp` is the name of the Java class that extends the `com.ibm.db2.app.UDF` class, and `dump` is a public method that implements the UDF.
3. The parameter style clause is DB2GENERAL.

Now, we examine the external program JVMProp that is referred to in the CREATE FUNCTION statement. We describe the parameters that DB2 Universal Database for iSeries sends to the program and how the program uses these parameters. This program was written in Java, as shown in Example 11-12.

Example 11-12 Source code JVMProp.java DB2GENERAL parameter style

```
import COM.ibm.db2.app.*;                                     1
import java.util.*;

public class MyUDTFs extends UDF {                            2

   Enumeration propertyNames;
   Properties  properties;

   public void ranking (String property, String value) throws Exception {
      int callType = getCallType();
      switch(callType) {                                      3
         case SQLUDF_TF_FIRST:                                4
            break;
         case SQLUDF_TF_OPEN:                                 5
            properties = System.getProperties();
            propertyNames = properties.propertyNames();
            break;
         case SQLUDF_TF_FETCH:                                6
            if (propertyNames.hasMoreElements()) {
               property = (String) propertyNames.nextElement();
               value = properties.getProperty(property);
               set(1, property);
               set(2, value);
```

```
            } else {
               setSQLstate("02000");                                    7
            }
            break;
         case SQLUDF_TF_CLOSE:                                          8
            break;
         case SQLUDF_TF_FINAL:                                          9
            break;
         default:
            throw new Exception("UNEXPECTED call type of " + callType);
      }

   }
}
```

Code sample notes

The following special comments refer to the numbers in the source code in Example 11-12 on page 365:

1. Package that contains the UDF class.
2. When the DB2GENERAL parameter style is used, the class must extend the UDF class.
3. The `getCallType` method is used to obtain the call timing, as explained in 11.5, “UDF and UDTF calling sequence” on page 350.
4. No action is required at first call in this particular UDTF.
5. At open time, the properties will be read into memory.
6. For each fetch, the next property and value tuple will be returned.
7. When no more properties exist to return, an SQLSTATE “`02000`” is signaled, which means that the last row was reached.
8. No action is required at the close in this particular UDTF.
9. No action at final call occurs in this particular UDTF.

As stored procedures, the Java file that contains the bytecode for the class must be in a specific path in the IFS:

`/QIBM/USERDATA/OS400/SQLLIB/Function`

Note: In DB2 UDF for i, both fenced and unfenced Java stored procedures, UDFs, and UDTFs are in `/QIBM/USERDATA/OS400/SQLLIB/Function` in the IFS. Other DB2 platforms can have different paths for fenced and unfenced Java stored procedures, UDFs, and UDTFs.

WebSphere Development Studio Client for IBM i provides a convenient development environment to develop Java stored procedures and UDFs, and an integrated environment for HLL languages, including CODE/400.

11.6.6 Coding the JAVA parameter style

SQL has date functions so that you can perform operations, such as calculating the number of days between two dates. But in real business applications, we usually need to know the time between two specific dates or time stamps in terms of working days or working hours. The CREATE FUNCTION statement in Example 11-13 creates a UDF that calculates the number of working days between two specific dates for a specific calendar, if the calendars have different holiday dates.

Example 11-13 CREATE FUNCTION for the JAVA parameter style

```
CREATE FUNCTION SAMPLEDB01.WORKING_DAYS (
      INITIALDATE DATE ,
      FINALDATE DATE ,
      CALENDARID INTEGER )
   RETURNS BIGINT
   LANGUAGE JAVA                                              1
   SPECIFIC SAMPLEDB01.WDAYS01
   DETERMINISTIC                                              2
   READS SQL DATA
   RETURNS NULL ON NULL INPUT
   NO EXTERNAL ACTION
   NOT FENCED
   EXTERNAL NAME 'WrkDays!wrkDays'                            3
   PARAMETER STYLE JAVA ;
```

Note: The following notes refer to Example 11-13:

1. The JAVA parameter style is allowed only when the language is Java.
2. Because this function has an SQL SELECT operation inside, it is important to avoid its execution when the same two dates are provided as parameters. For this reason, it was defined as DETERMINISTIC.
3. In this example, `WrkDays` is the name of the class and `wrkDays` is the `public static` method that implements this particular UDF.

Now, we examine the external program WrkDays, which is referred to in the CREATE FUNCTION statement. We describe the parameters that DB2 Universal Database for iSeries sends to the program and how the program uses these parameters. This program was written in Java, as shown in Example 11-14.

Example 11-14 Source code WrkDays.java in the JAVA parameter style

```
import java.sql.*;

public class WrkDays {

   public static long wrkDays(Date initialDate, Date finalDate, int calendarKey)
   throws SQLException {                                                          1
      int count;
      long result;
      Connection con = DriverManager.getConnection("jdbc:default:connection");    2
      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                    3
              - count;
        }
        else
           throw new SQLException("Error reading holiday table", "38ZZZ");        4
        if (rs != null) rs.close();
        if (ps != null) ps.close();
        if (con != null) con.close();
        return result;                                                             5

         }
}
```

Code sample notes

The following special comments describe the numbers in the source code in Example 11-14 on page 367:

1. For the JAVA parameter style, a public Java class must be defined. That class can contain one or more methods, one for each UDF. UDFs and stored procedures can be combined in the same class. The method that implements the UDF must be `public static`.
2. If the Java method that implements the UDF will perform database operations, it must connect to the database manager by using the jdbc:default:connection URL.
3. The difference between the two dates is calculated in milliseconds and then converted to days. The number of holidays is then subtracted.
4. If any SQL error occurs, an `SQLException` is thrown by the method to the caller.
5. The value that is returned by the method needs to be compatible with the SQL types.

11.7 Error handling in external UDFs

UDFs can report errors and warnings by the same mechanisms that are described in Chapter 6, “Stored procedure error handling” on page 143. These mechanisms depend on the parameter passing convention that is used.

When DB2SQL parameter passing is used, the UDF program reports errors and warnings by using the **SQLUDF_MSGTX** and **SQLUDF_STATE** parameters. When DB2GENERAL or JAVA parameter passing is used, the UDF program reports errors by throwing `SQLException` or `SQLWarning` exceptions. If a warning message is sent back to DB2, the SQL statement continues running. When an error message is sent back to DB2, the SQL statement stops. Reported SQL errors and warnings need to follow the same conventions as stored procedures.

11.7.1 Error handling with the DB2SQL parameter style

The following UDTF (Example 11-15 on page 369) receives a file name that corresponds to a stream file in the IFS and returns it as a table. Errors might occur during the execution:

- The file might not exist.
- An I/O error might occur during the processing of the file.

Also, warnings might occur during the execution of the program:

- Rows in the file might have an inappropriate format.
- The end of file is reached.

Example 11-15 shows the CREATE FUNCTION statement for the F1Results UDTF.

Example 11-15 CREATE FUNCTION statement for F1Results 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                                          1
   NOT FENCED
   CARDINALITY 22                                             2
   EXTERNAL NAME 'SAMPLEDB01/CUDF(readFileToTable)
   PARAMETER STYLE DB2SQL;
```

The UDTFs require that DISALLOW PARALLEL is specified in 1. We specified CARDINALITY to 22 at 2 to help the optimizer provide an estimate of the average number of rows that are returned by the UDTF.

The code in Example 11-16 features error handling and pointer arithmetic. Pointer arithmetic was used to align the pointer to the file structure. Pointer arithmetic is explained in 11.8, “Pointer arithmetic and the scratchpad” on page 375.

Example 11-16 HLL UDTF that shows 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 {                                                  1
                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;      2
strSPad = (strScratchPad*)ptrAlignmentPointer;

switch (SQLUDF_CALLT) {
        case SQLUDF_TF_OPEN:
                /* open a text stream file and store the pointer */
                /* on the scratch pad. Check to see if it opened */
                /* successfully                                  */
                if ((strSPad->f = fopen(fileName, "r")) == NULL) {   3
                        strcpy(SQLUDF_MSGTX, "Could not open file ");
                        strncat(SQLUDF_MSGTX, fileName,
                                SQLUDF_MSGTEXT_LEN -
                                strlen(SQLUDF_MSGTX) -1);            4
                        strncpy(SQLUDF_STATE, "38200",
                                SQLUDF_SQLSTATE_LEN);                5
                        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);                6
                        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);                                    7
                        strSPad->f = NULL;
                        strSPad->rowNumber = 0;
                        break;
        }
}
```

In Example 11-16 on page 369, 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 is established in line 4 and the error state 38200 is established in line 5.

When the end of the file that contains the race scores is reached, line 6 signals a warning message that indicates to DB2 for i that the end of the returning table is reached. In line 7, the file is closed.

Note: To obtain access to files in the IFS, the **CRTCMOD** command was issued with the **SYSIFCOPT** option set to *IFSIO:

```
CRTCMOD MODULE(DLEMA/F1UDTF) SRCFILE(DLEMA/CSRC) OUTPUT(*print)
DBGVIEW(*SOURCE) SYSIFCOPT(*IFSIO)
```

11.7.2 Error handling with the DB2GENERAL parameter style

Java UDFs and UDTFs that use the DB2GENERAL parameter style inherit methods for reporting exceptions and warnings:

- setSQLstate
- setSQLmessage

Be careful with the state and size of the returned message. The state code must follow the rules that are explained in 6.1, “Database error reporting strategy” on page 144. Message text must not exceed 70 characters.

Example 11-17 presents a Java version of Example 11-16 on page 369.

Example 11-17 Java version of the F1Results UDTF

```
public class F1RESULTS extends UDF {                                  1


   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 nfe) {
               setSQLmessage("nfe in line " + lines);             2
               setSQLstate("01H04");                              3
            }
         } 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);                 2
         setSQLstate("01H04");                                  3

      }
   }
}

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");                            4
               return;
            }

            String line = f1BufferedReader.readLine();
            if (line == null) {
               setSQLstate("02000");                            4
```

```
                return;
             }
             parseLine(line);
             break;
          case SQLUDF_TF_CLOSE:
             closeFile(); break;
          default:
             setSQLstate("38H06");                                    5
             setSQLmessage("Improper call type");                     5
       }
    }
    catch(IOException ioe) {                                          6
       String message = ioe.getMessage().trim();
       if (message.length() > 70)
          message = message.substring(0,70);
       setSQLmessage(ioe.getMessage());
       setSQLstate("38H01");
       return;
    }
 }
}
```

Code sample notes

The following are some special comments of the source code in Example 11-17 on page 372:

1 Java UDTFs are supported in the DB2GENERAL parameter style only. The DB2GENERAL parameter style Java UDFs must extend the UDF class.

2 When a number format exception is detected, a message that indicates the row in which the problem was identified 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 an SQL state of 02000 to indicate that no more rows exist.

5 When a call type other than open, fetch, or close fires, an SQL error occurs with state 38H06 and message "`Improper call type`".

6 I/O errors cause an SQL error with state 38H01. The program controls the maximum length of the error message.

A simple client program that uses this UDTF is illustrated in Example 11-18.

Example 11-18 Client program that uses the F1RESULTS UDTF

```
public class F1Client {                                               1

   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();                    2

            if (sqlw != null)
               printMessage(sqlw.getSQLState(), sqlw.getMessage());
            // now print the row columns... omitted for simplicity
         }
      }
      catch (SQLException sqle) {                                  3
         printMessage(sqle.getSQLState(), sqle.getMessage());
         System.exit(2);
      }

   }
}
```

Code sample notes

The following special comments refer to the source code in Example 11-18 on page 374:

1 UDTFs can be used from any kind of program, regardless of the original programming language that was used. This sample code is a Java program that receives the driver name, database URL, user, password, and file name as parameters. The sample code assembles the SELECT statement, which includes a UDTF that reads the forecasted race results for next year and prints them. The code does not print it (for simplicity).

2 After the program fetches 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.

11.8 Pointer arithmetic and the scratchpad

Each hardware platform used to have its own requirements for the alignment of certain variables. When you code a UDF with scratchpad, those alignment rules must be followed. In IBM i servers, pointers must be aligned in 16-byte boundaries. The snippet in Example 11-19 gives the definition of the scratchpad structure as it is defined in the include file `sqludf.h`.

Example 11-19 Definition of the scratchpad structure as it is defined in the include file sqludf.h

```
SQL_STRUCTURE sqludf_scratchpad {
   unsigned long length;       /* length of scratchpad data */
   char          data[SQLUDF_SCRATCHPAD_LEN]; /* scratchpad data, init. to all \0
*/
};
```

In Example 11-16 on page 369, we need to store a file pointer in the scratchpad, but the data part of it might not align to a 16 boundary, so we can code a snippet similar to the snippet in Example 11-20 to align that pointer to a 16 boundary.

Example 11-20 Code to align the pointer to a 16 boundary

```
typedef struct {
        char filler[8];
        FILE *f;
        integer rowNumber;
} strScratchpad;
...
strScratchpad *ScratchPadData;
...
ScratchPadData = *sqludf_scratchpad->data;
...
```

However, this code is platform-dependent. If we plan to move the code to another platform or even if the platform architecture changes, the program will not work anymore. In the future, when 64-bit architecture becomes obsolete and the IBM i evolves into a 128-bit machine, the boundary for pointers might change. More platform-independent code is required, such as the code that is shown in the snippet in Example 11-21.

Example 11-21 Platform-independent code

```
typedef struct {
        FILE *f;
        integer rowNumber;
} strScratchpad;
...
        strScratchPad *strSPad;                                          2
        strScratchPad **ptrAlignmentPointer;
...
       ptrAlignmentPointer = ((strScratchPad**)(sqludf_scratchpad))+1;  3
       strSPad = (strSratchPad*)ptrAlignmentPointer;
```

In the previous code snippet in Example 11-21, you see that a structure, which is called strScratchPad, was declared. The variable *f* is a pointer to the input stream file to be read, and rowNumber is an integer to track the line that is read to report warning or error messages that refer to the failing row. We declare a pointer to this structure that is called strSPad at 2. 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 3 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 correctly, an exception is thrown at run time, and the application is terminated.

11.8.1 Debugging external UDFs

When you develop any kind of software, a debugging tool is important. Use *debugging* to detect, diagnose, and eliminate runtime errors in a program. This section shows you debugging alternatives to test SQL/Persistent Stored Modules (PSMs).

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 referenced by an external UDF.

In this example, we debug our TOTSAL UDF. Debugging UDFs might be tricky because 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. 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:

   STRSQL

   The interactive SQL session starts, and the SQL command line is displayed.
3. Switch to session A and type the following command:

   WRKACTJOB

   The Work with Active Jobs panel is displayed in Figure 11-16. This panel displays a list of all jobs that are currently active on the system. The job in session B is listed.

```
                     Work with Active Jobs                      AS07
                                                              09/24/03  10:16:23
CPU %:     .1     Elapsed time:   01:41:42     Active jobs:   167

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...

Opt  Subsystem/Job  User        Type  CPU %  Function        Status
     QBATCH         QSYS        SBS      .0                  DEQW
     QCMN           QSYS        SBS      .0                  DEQW
     QCTL           QSYS        SBS      .0                  DEQW
       QSYSSCD      QPGMR       BCH      .0  PGM-QEZSCNEP    EVTW
     QINTER         QSYS        SBS      .0                  DEQW
5      QPADEV0002   FREDYC      INT      .0  CMD-STRSQL      DSPW
       QPADEV0003   FREDYC      INT      .0  CMD-WRKACTJOB   RUN
     QSERVER        QSYS        SBS      .0                  DEQW
       QPWFSERVSD   QUSER       BCH      .0                  SELW
                                                                      More...
Parameters or command
===>
F3=Exit   F5=Refresh       F7=Find      F10=Restart statistics
F11=Display elapsed data   F12=Cancel   F23=More options   F24=More keys
```

Figure 11-16 Work with Active Jobs panel

4. After you find the job, in our case, QPADEV0002, use option 5 to work with that job. Then, the Work with Job panel is displayed. Choose option 2 and type these characteristics:
   - Job: This field is the name of the job with which you are working.
   - User: This field is the name of the user profile that is using the job.
   - Number: This field is the number that is assigned to the job you are working with. Every job on the OS/400 system is assigned a 6-digit unique job number.

   In our case, we use this data:

   ```
   Job:   QPADEV0002     User:   FREDYC         Number:   016859
   ```

5. Start a service job for the session B job. Enter the following command on the command line:

   STRSVRJOB 016859/FREDYC/QPADE0002

6. Start a debug session for the service program that is used in the TOTSAL function. Type the following command on the command line:

   STRDBG UPDPROD(*YES) SRVPGM(SAMPLEDB01/TOTSAL)

7. You see the debug session on your panel with the source code loaded into the debugger. In this example, we analyze the ILE C function that the UDF generates. Go to the function SQLPROC1, which is the “IF” in SQL that is transformed in ILE C. Put a breakpoint in one of the executable statements in the program. In our case, we chose this statement:

   ```
   if (SQL_STRUCT_HV.SQL_DATA_RETURNED == '1')
   ```

   To create a breakpoint, place the cursor on the line of code at which you want to place the breakpoint and press F6.

   You see the following message at the bottom of the window:

   ```
   Breakpoint added to line 396.
   ```

   This procedure is shown in Figure 11-17.

```
                              Display Module Source

Program:   TOTALSAL       Library:   SAMPLEDB01     Module:   TOTALSAL
   391         } SQL_STRUCT_HV;
   392       4   memcpy(SQL_STRUCT_HV.SQL_VAR_1,&(*TOTALSAL_x).LASTNAME,17);
   393       5   SQL_STRUCT_HV.SQL_VAR_4[0]=(*TOTALSAL_x).SQLP_I4;
   394       6   sqlca.sqlerrd[5] = -9;
   395       7   QSQROUTE ((SQLCA * )&sqlca,&SQL_STRUCT,&SQL_STRUCT_HV);
   396       8   if (SQL_STRUCT_HV.SQL_DATA_RETURNED == '1')
   397             {
   398       9        SQLP_INT_VAR = SQL_STRUCT_HV.SQL_VAR_3;
   399             }
   400      10   SQLCODE=SQLCADE;
   401      11   memcpy(SQLSTATE,SQLSTOTE,5);
   402         #if (__OS400_TGTVRM__>=510)
   403             #pragma datamodel(pop)
   404         #endif
   405         }
                                                                        More...
Debug . . .

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
Breakpoint added to line 396.
```

Figure 11-17 Adding a breakpoint to the debug session

8. Press F12 to resume, which takes you back to the command line. You must invoke the UDF from the interactive SQL that is run in session B.

9. 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 executes. The TOTSAL (DECIMAL, DECIMAL, DECIMAL, VARCHAR) UDF is invoked. The following message is displayed at the bottom of the panel:

   ```
   Query running. 0 records selected, 1 processed.
   ```

   This message is shown in Figure 11-18. However, the result of the query does not show. Instead, the session busy cross sign stays at the bottom of the panel.

```
                                 Enter SQL Statements

 Type SQL statement, press Enter.
      Current connection is to relational database MONACO.

 ===> SELECT FIRSTNME,LASTNAME,TOTSAL(SALARY,BONUS,COMM,LASTNAME) AS TOTAL
      FROM EMPLOYEE


Bottom
 F3=Exit   F4=Prompt   F6=Insert line   F9=Retrieve   F10=Copy line
 F12=Cancel            F13=Services     F24=More keys
 Query running. 0 records selected, 1 processed.
```

Figure 11-18 Invoking the TOTSAL SQL UDF

10. Switch back to session A. You see the source code of the PICTCHECK service program displayed on the panel. The line of source code that will be executed is highlighted in white on the panel. In our case, this line is the line at which you set the breakpoint in step 7 on page 378.

11. Press F10 to execute the highlighted line of code. The line is executed, and it is no longer highlighted. The next line of code to execute is highlighted. Each time that you press F10, the next line of code in sequence is executed.

12. You can check the value that is contained in any of the program variables in one of two ways:
   – Press F11 after you place the cursor over the variable for which you want to check the value.
   – Type 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 panel, as shown in Figure 11-19.

```
Display Module Source
Current thread:   00000010     Stopped thread:   00000010
Program:   TOTALSAL       Library:   SAMPLEDB01     Module:   TOTALSAL
   392       4   memcpy(SQL_STRUCT_HV.SQL_VAR_1,&(*TOTALSAL_x).LASTNAME,17);
   393       5   SQL_STRUCT_HV.SQL_VAR_4[0]=(*TOTALSAL_x).SQLP_I4;
   394       6   sqlca.sqlerrd[5] = -9;
   395       7   QSQROUTE ((SQLCA * )&sqlca,&SQL_STRUCT,&SQL_STRUCT_HV);
   396       8   if (SQL_STRUCT_HV.SQL_DATA_RETURNED == '1')
   397             {
   398       9        SQLP_INT_VAR = SQL_STRUCT_HV.SQL_VAR_3;
   399             }
   400      10   SQLCODE=SQLCADE;
   401      11   memcpy(SQLSTATE,SQLSTOTE,5);
   402          #if (__OS400_TGTVRM__>=510)
   403              #pragma datamodel(pop)
   404          #endif
   405          }
   406      12 if (sqlca.sqlcade == 100) {
                                                                          More...
Debug . . .

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
SQLCODE = 100
```

Figure 11-19 Checking the value of the program variables by using F11

13. Continue to press F10 until you step through the entire program. At any time, you can run the program to completion by pressing F12.
14. After you finish debugging the code, you return to the Work with Job display. On the command line, type the following CL commands:

   ENDDBG
   ENDSRVJOB

These commands end the debug mode and the service job that was run to debug the service program.

11.9 Coding example for an external user-defined table function

In this section, we provide sample coding for an external UDTF. Because SQL does not have a function that is equivalent to the **DIR** command in Windows or **LS** in UNIX, we write a program to perform this work and create an external UDTF for this program. This external UDTF receives one parameter of the OS/400 integrated file system (IFS) directory path name (fewer than 128 characters in length) and returns a list of all of the subdirectories of the specified path.

We give this UDTF a name of IFSDIR and store it in a schema that is named SQLLIB. The external program is coded in RPG IV.

Example 11-22 shows the code of the RPG IV program.

Example 11-22 RPG IV code of the IFSDIR program

```
 *=================================================================
 *  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                    1A
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'
D ENDOFTABLE      c                   '02000'
D UDTF_ERROR      c                   'U5001'
 *
 *  NULL Constants
D ISNULL          c                   -1
D NOTNULL         c                    0
 *
 *  Prototypes for all procedures
D GetAttr         pr              n
D  QPathName                    128    Const
D  CountryID                      2    Const
D  LangID                         3    Const
 *
 *  Get IFS Object Attributes
DQp0lGetAttr      pr             10i 0 ExtProc('Qp0lGetAttr')
D  pPath                          *   Value
D  pAttrArr                       *   Value
D  pBuffer                        *   Value
D  pBufferSize                  10u 0 Value
D  pBufferSizeN                   *   Value
D  pNoBytes                       *   Value
D  pSymLinkUse                  10u 0 Value
 *
 *  Directory Prototype Entries
 *
 *  Open an IFS Directory
DOpenDir          PR              *   EXTPROC('opendir')
D  dirname                        *   Value
 *
 *  Read IFS Directory Contents
DReadDir          PR              *   EXTPROC('readdir')
D  dirhndl                        *   Value
 *
 *  Close IFS Directory
DCloseDir         PR             10i 0 EXTPROC('closedir')
D  dirhndl                        *   Value
 *
 *  IFS Entry Handles
D dirName         s              128
D dirHandle       s                *
D DirPtr          s                *
D dsDirEntry      ds                   Based(DirPtr)
D  MiscInfo                      40
D  DirEntryCCSID                 10i 0
```

```
D  DirEntryCntry                2
D  DirEntryLang                 3
D  DirEntryNLSR                 3
D  DirEntryL                   10u 0
D  DirEntry                   640
 ******************************************************************
 * C API Error Retrieval
 ******************************************************************
dGetErrNo         pr              *   Extproc('__errno')
dStrError         pr              *   Extproc('strerror')
d                                10i 0 Value
 *
dErrNo            s               10i 0 Based(pErrNo)
dErrStr           s              128    Based(ptrStrError)
dptrStrError      s                 *
 *
dErrDesc          s              128
dStatus           s              128
dNULL             s                1    Inz(X'00')
 *
 *  Parameters for DB2SQL parameter style
 *
 *  SQL State  -  Input/Output
D SQL_State       s                5
 *  Function Name    Schema.Def name - Input only
D Function_Name   s              517
 *  Function Specifc Name - Input Only
D Specific_Name   s              128
 *  Message Text - Input/Output
D Msg_Text        s               70    Varying
 *
 *  Error Code Data Structure for API Calls
DdsErrCode        DS
D BytesPrv                       10i 0       Inz(%Size(dsErrCode))
D BytesAvl                       10i 0       Inz(%Size(dsErrCode))
D ExcID                           7
D Reserved                        1
D ErrData                       128

C     *Entry        PList
 *
 * Function input parameters
C                   Parm                    Folder
 *
 * Function output parameters
C                   Parm                    FileName
C                   Parm                    CreateStamp
C                   Parm                    AccessStamp
C                   Parm                    ModStamp
C                   Parm                    Type
C                   Parm                    DataSize
C                   Parm                    AllocSize
C                   Parm                    Owner
 *
 * Function NULL input parameter indicators
C                   Parm                    Folder_NI
 *
 * Function NULL output parameter indicators
C                   Parm                    FileName_NI
C                   Parm                    CreateStamp_NI
```

```
C                   Parm                    AccessStamp_NI
C                   Parm                    ModStamp_NI
C                   Parm                    Type_NI
C                   Parm                    DataSize_NI
C                   Parm                    AllocSize_NI
C                   Parm                    Owner_NI
 *
 *  DB2SQL Style Parms
C                   Parm                    SQL_State
C                   Parm                    Function_Name
C                   Parm                    Specific_Name
C                   Parm                    Msg_Text
 *
 *  UDTF CallType flag parm  (Open,Fetch,Close)
C                   Parm                    CallType
 *
 * 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
    //          inifinte 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;
             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;
EndSl;
Return;

BegSr *PSSR;
    SQL_State=UDTF_ERROR;
    Msg_Text='General Program Error';
    *InLR=*On;
    Return;
```

```
   EndSr;
/End-Free
*================================================================
*  GetAttr (call the QpOlGetAttr 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                                      1B
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 QpOlGetAttr
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.  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_exclude                    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                     10O    Const
D   BufferLen                   10u 0 Const
D   NoConsts                    10u 0 Const
D   ConstList                   10    Const
D   ErrorStuc                        Like(dsErrCode)

D   Buffer        S            10O
C/Free
    //
    // Retrieve UTC Offset for this machine, the first time through
    //
    // 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*36OO + Minutes*6O;
        If Sign='-';
            UTCAdjust=-UTCAdjust;
        EndIf;
    EndIf;
    //-----------------------------------------------------------------
    // Call API
    //-----------------------------------------------------------------
    pPathName=qPathName;
    pPathNameLen=%Len(%Trim(qPathName));
    pCountryID=CountryID;
    pLangID=LangID;
    //
    // Retrieve Object Attributes
    //
    ReturnCode=QpOlGetAttr(%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;
                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_AllcSize;
                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;
 /END-FREE
P                 E
```

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 that is named IFSDIR with the following SQL statement:

```
CREATE FUNCTION SQLLIB.IFSDIR (IFSFOLDER VARCHAR(128))
       RETURNS TABLE (FILENAME    VARCHAR(128),       1C
                      CREATESTAMP TIMESTAMP,
                      ACCESSSTAMP TIMESTAMP,
                      MODSTAMP    TIMESTAMP,
                      TYPE        CHAR(10),
                      DATASIZE    BIGINT,
                      ALLOCSIZE   BIGINT,
                      OWNER       CHAR(10))
       EXTERNAL NAME 'SQLLIB/IFSDIR'               2
       LANGUAGE RPGLE
       DISALLOW PARALLEL
       RETURNS NULL ON NULL INPUT
       PARAMETER STYLE DB2SQL ;
```

The following special comments refer to the previous SQL statement:

- The column definition of the returned table at 1C must match the column definitions at 1A and 1B in Example 11-22 on page 381.
- The parameter of EXTERNAL NAME 2 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 11-20 shows a sample result of the IFSDIR UDTF.

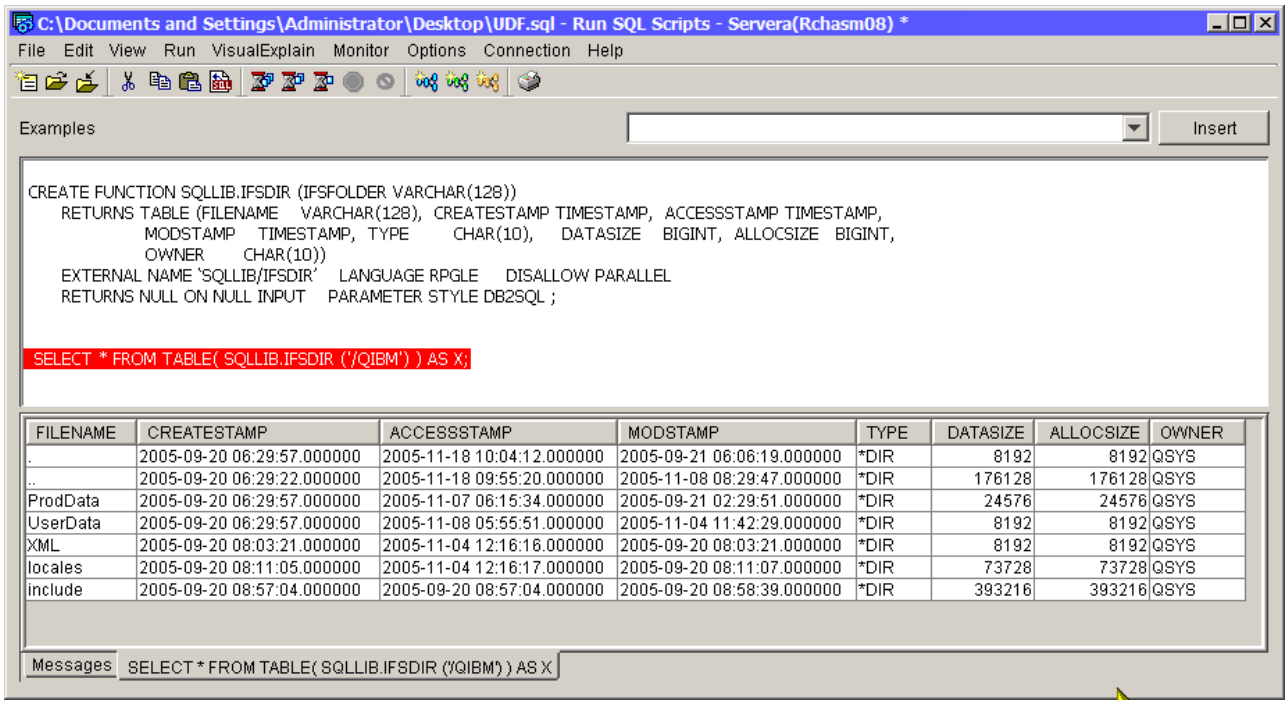


Figure 11-20 Result of the IFSDIR UDTF

A

Sample ILE C program that uses the QDBRTVFD API

This appendix shows code for a sample Integrated Language Environment (ILE) C program that uses the Retrieve File Description application programming interface (API). This program shows how to obtain information about the triggers that are associated with a physical file. The program can also be modified easily to retrieve information about the referential integrity constraints.

Sample ILE C program that uses the QDBRTVFD API

Example A-1 shows a sample ILE C program where the Retrieve File Description API is used.

Example A-1 ILE C program that uses the QDBRTVFD API

```
#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;
 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,
         &in_parms->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..\n");
     }
     else         /* Else the file is not a physical file.        */
```

```
        printf("The file is not a physical file...\n");
  }
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...\n",
            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 pooointer 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 ...*/
     {
      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 that are associated with a database file. The list is retrieved by using the QDBRTVFD APIs. A complete description of this API is in *System API Programming*, SC41-5800, which is available at the following website:

https://publib.boulder.ibm.com/iseries/v5r1/ic2924/books/c4158000.pdf

This utility can be created and called in the following way:

```
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.

B

Order Entry application: Detailed flow

This appendix provides detailed flow charts of each module that is included in the Order Entry application scenario. The following modules are described:

- Program flow for the Insert Order Header program
- Program description for the Insert Order Header program
- Program flow for the Insert Order Detail program
- Program description for the Insert Order Detail program
- Program flow for the Finalize Order program
- Program description for the Finalize Order program

Program flow for the Insert Order Header program

DB2 Universal Database for iSeries functional highlights in this program include the following topics:

- Referential integrity constraints for the Order Header table
- Insert trigger on the Order Header file

Figure B-1 shows a functional description of the various components of this application scenario.

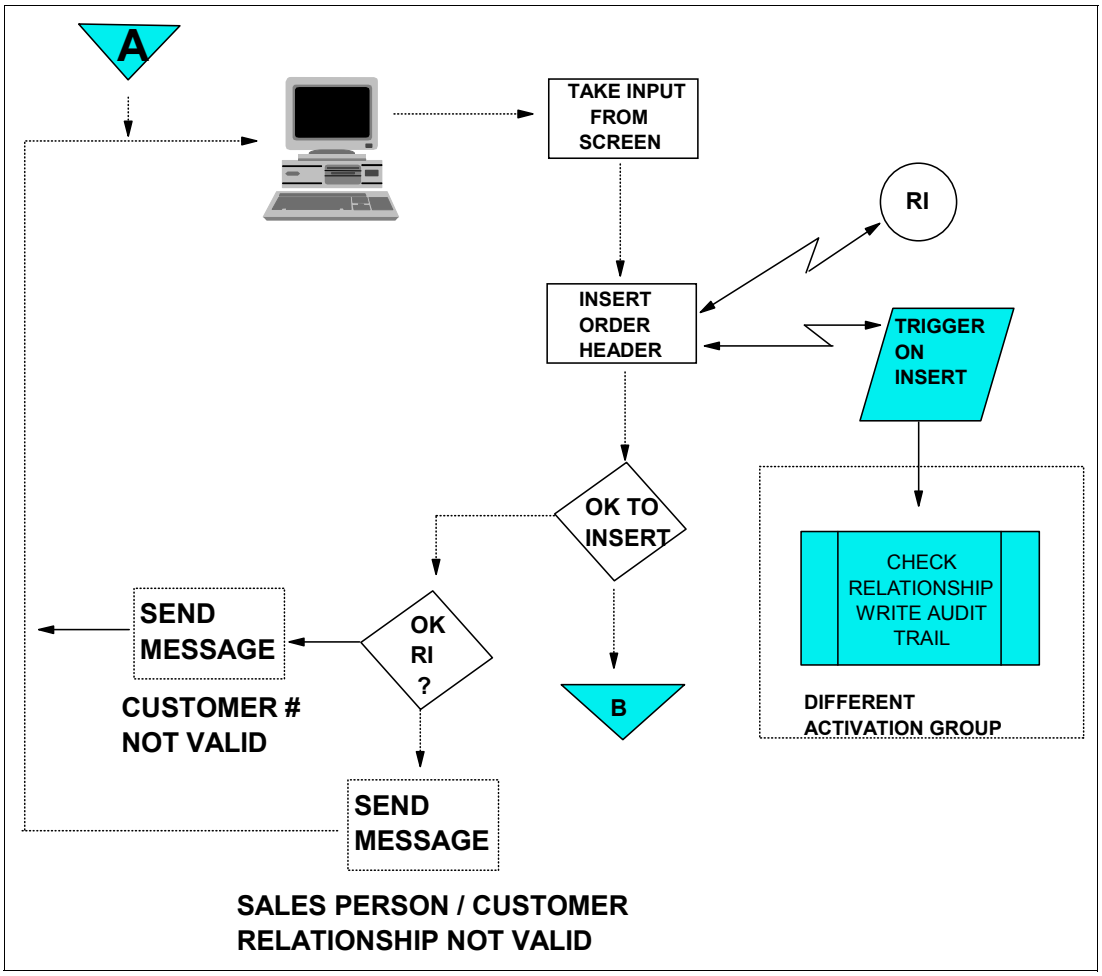


Figure B-1 Insert Order Header program flow

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 completes, the database manager activates a program that can verify whether 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, the order number is entered by the sales representative.

A more detailed flow of this program is described:

1. The program inserts a row into the Order Header table.
2. If the database referential constraint enforcement detects a customer number that is not defined in the Customer table, a program message is sent to explain 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. Because an insert trigger is defined on this table, a program is automatically triggered by the database manager.
6. The trigger program checks whether the current user profile is associated with the customer in the Sales/Customer table. If no match exists, 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 for i functional highlights in this program include the following topics:

- Referential integrity constraints for the Order Detail table
- Referential integrity constraints for the Stock table (on the remote system)
- Two-phase commit and the second stage of Distributed Relational Database Architecture (DRDA-2) or DRDA Level 2
- Remote stored procedure

The program flow for Insert Order Detail is shown in Figure B-2.

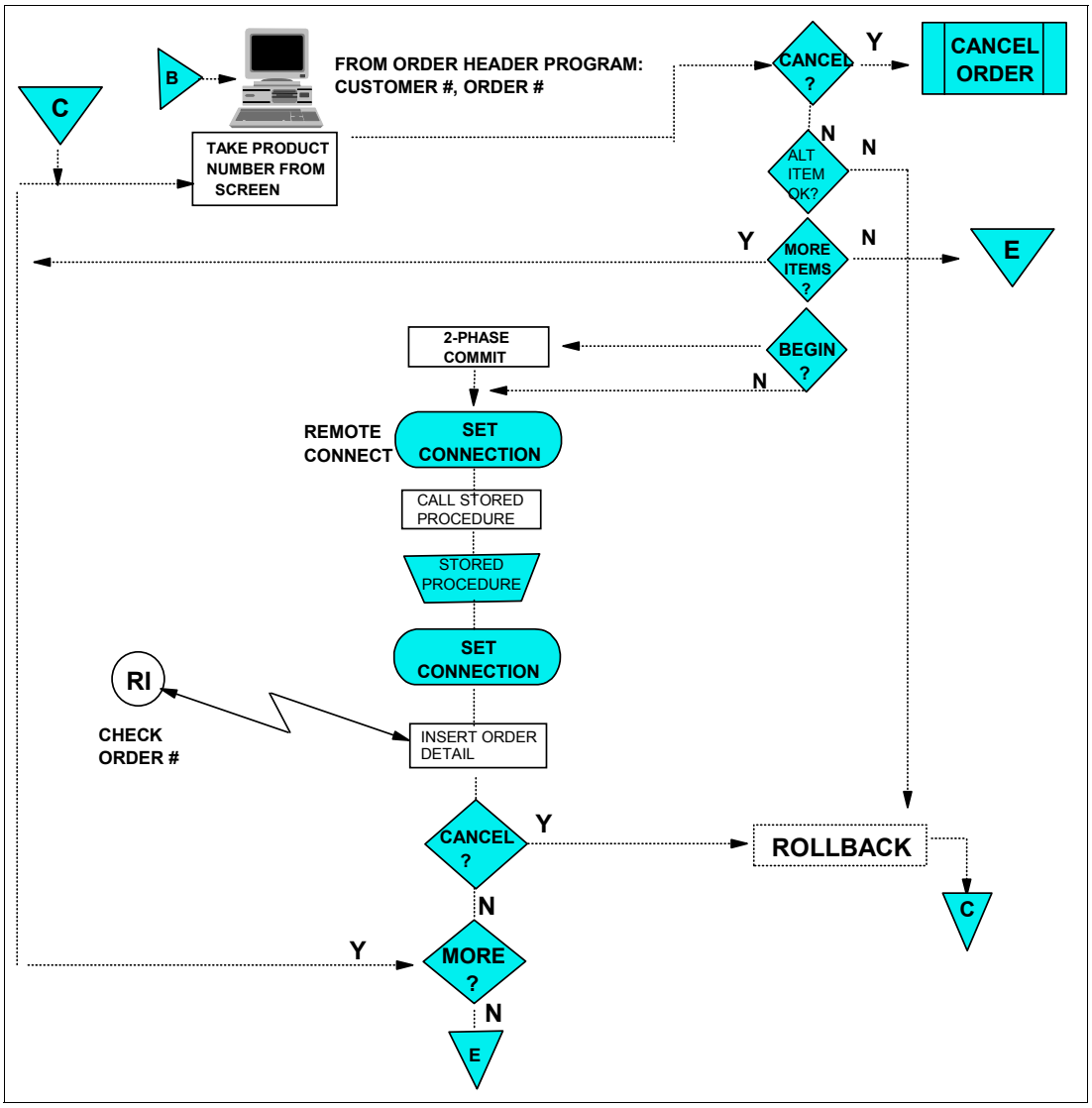


Figure B-2 Insert Order Detail program flow

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 can choose to accept or reject the substitute item.

A more detailed flow of this program is described:

1. Get the customer number and the order number from the Insert Order Header program.
2. Get the product number and quantity for every item from the display.
3. Issue a SET CONNECTION statement to the remote system. All of the necessary CONNECT statements are performed by the main program.
4. Call a stored procedure at the remote system to perform the following tasks:
   - 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 product is out of stock, and update the corresponding quantity.
   - Pass the product information back to the calling program.
5. The stored procedure then passes control back to the calling program.
6. 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.
7. A rollback is also performed if the referential integrity checking on the Order Detail table fails. The checking fails if you insert the record with the wrong order number.
8. Optional: The user can cancel the whole order. In this case, a Cancel Order program is called.
9. The program maintains 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 for i functional highlights in this program include the trigger on the Update Order Header row. See Figure B-3 for the program flow.

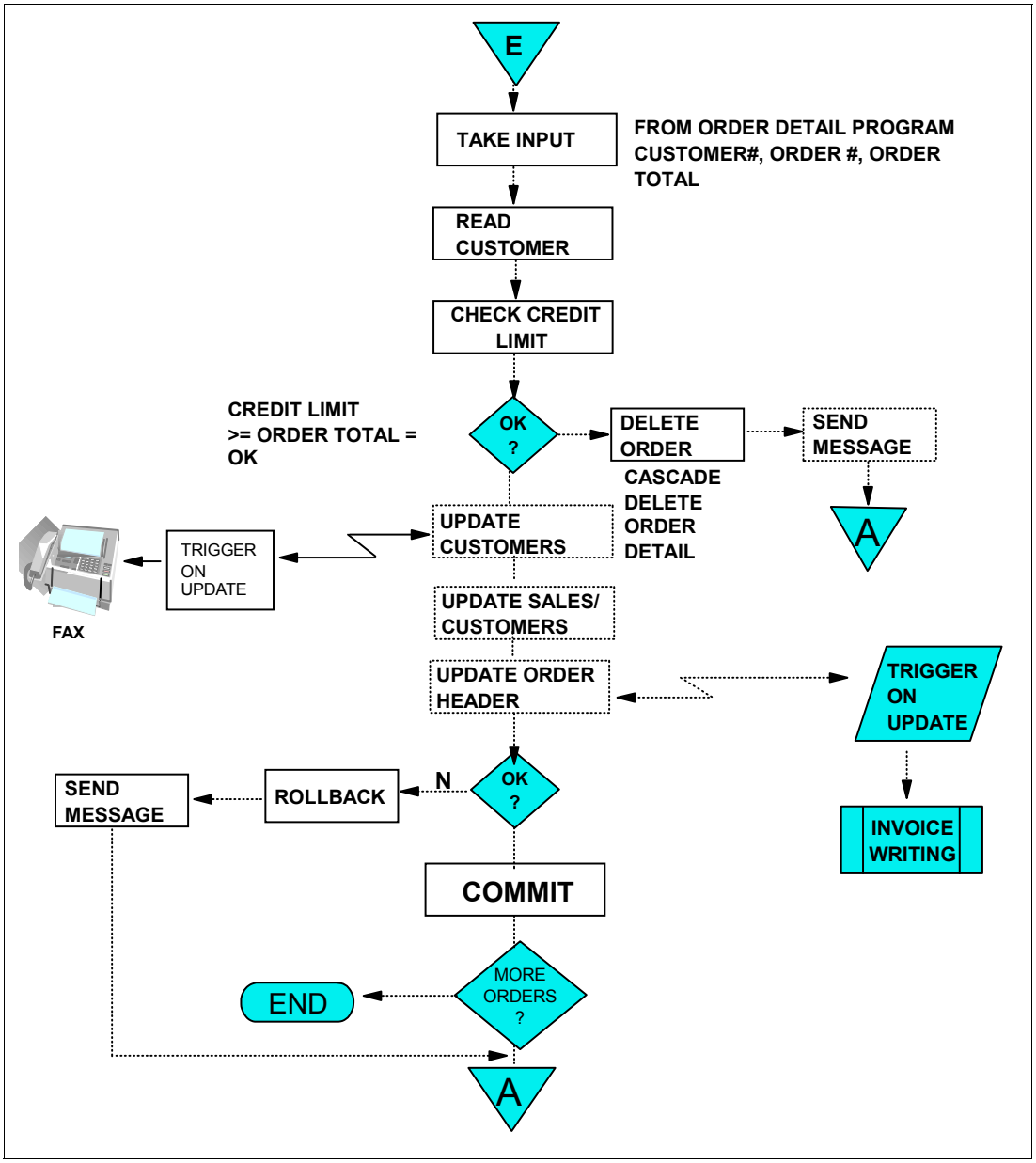


Figure B-3 Finalize Order program flows

Program description for the Finalize Order program

The idea of this program is to show how to use the *database trigger* function in a real application. 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 is complete.

Also, the program updates the credit limit on the customer file. If the current balance exceeds 90% 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).

A more detailed flow of this program is described:

1. Get the customer number and the order number from the previous process with the order grand total.
2. Check the customer record. If the credit limit is exceeded, the order is canceled. 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 that was reserved for this order. When this process is complete, the Order Header is deleted, and all of the order detail disappears as a result of the *CASCADE constraint on the Order Header file. Finally, the entire transaction is committed. Again, the two-phase commit support ensures that the local database and the remote stock file are kept synchronized.
3. If the credit limit is OK, this program updates the following fields:
   - The total amount in the customer file to track 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
4. 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.
5. After the preceding updates are complete, COMMIT is executed.
6. If more orders exist, the Insert Order Header program starts again.
7. If no more orders exist, this Order Entry application ends.

C

Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the web material

The web material that is associated with this book is available in softcopy on the Internet from the IBM Redbooks web server. Point your web browser at this website:

`ftp://www.redbooks.ibm.com/redbooks/SG246503`

Alternatively, you can go to the IBM Redbooks website:

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

Select **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG246503.

Using the web material

The additional web material that accompanies this book includes the following files:

File name	Description
`dbadvfun.zip`	IBM i and client source image
`readme.zip`	Readme documentation

System requirements for downloading the web material

The web material requires the following system configuration:

- IBM i requirements:
  - OS/400 Version 5 Release 1
  - 5722-ST1 - DB2 Query Manager and SQL Development Kit
  - 5722-SS1 - Host servers
- Personal computer software:
  - Microsoft Windows 95/98, Windows NT, or Windows 2000
  - iSeries/IBM i Access Express for Windows
  - PC5250 Emulation

Downloading and extracting the web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material (a compressed file) into this folder.

The `readme.txt` file contains the instructions for restoring the IBM i libraries and directories and installing the PC clients and runtime notes.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks publications

The following IBM Redbooks publications provide additional information about the topic in this document. Several publications referenced in this list might be available in softcopy only.

- *A Fast Path to AS/400 Client/Server Using AS/400 OLE DB Support*, SG24-5183
- *SQL Procedures, Triggers, and Functions on DB2 for i*, SG24-8326
- *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
- *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402

Other resources

These publications are also relevant as further information sources:

- *Backup and Recovery*, SC41-5304
- *Database Programming*, SC41-5701
- *DB2 UDB for AS/400 SQL Programming*, SC41-5611
- *SQL Reference*, SC41-5612
- *System API Programming*, SC41-5800:

  https://publib.boulder.ibm.com/iseries/v5r1/ic2924/books/c4158000.pdf
- Conte, Paul, *Database Design and Programming for DB2/400*, 29th Street Press, April 1997, ISBN 1-8824190-65
- Conte, Paul and Cravitz, Mike, *SQL/400 Developer's Guide*, 29th Street Press, September 2000, ISBN 1-882419-70-7

Referenced websites

These websites are also relevant as further information sources:

- IBM Toolbox for Java driver:

  https://ibm.biz/Bd4c5V
- *SQL Reference for Cross-Platform Development*:

  ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/cpsqlrv11.pdf

- *DB2 Universal Database for iSeries SQL Messages and Codes*:
  https://publib.boulder.ibm.com/iseries/v5r1/ic2924/info/rzala/rzalamst.pdf
- IBM i Information Center:
  http://publib.boulder.ibm.com/html/as400/v5r1/ic2924/index.htm

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Redbooks

External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i

SG24-6503-03
ISBN 0738441597

IBM

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

SG24-6503-03

ISBN 0738441597

Printed in U.S.A.